# System Level Memory Size Estimation

Gabi Grun
Nikil Dutt
Florin Balasa

Technical Report 97-37
September 3, 1997

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
(714) 824-7063

## Abstract

DSP applications, such as image, voice, and video processing are characterized by large multi-dimensional arrays, accessed from multiple nested loops. The cost and performance of the final implementation of such designs is dominated by the background storage. To optimize the memory size and the memory accesses, they have to be considered at an early design stage. We propose a fast memory size estimation method for a parallel procedural representation, which can trade-off computation time against accuracy. This technique is presented in the context of a Hardware/Software exploration environment.

# 1. Introduction.

DSP systems, such as mobile telecommunication, video, image, and voice processing routines can be characterized as computation-intensive systems, where the data is represented most often by large multi-dimensional arrays. In order to perform these computations, the design implementation typically contains a hierarchy of memory components that satisfy both the storage and performance requirements of the design.

In order to achieve a low cost implementation, while still satisfying the performance constraints of the design, the trade-off between parallelism and chip area has to be considered. Traditionally, the relationship between the degree of parallelism and the cost of the functional units is considered. Additionally, in memory-dominated applications, the trade-off between parallelism and memory size has to be accounted. Program structure (such as the dependences, and memory access pattern) will determine the performance of the implementation, as well as the functional and storage requirements.

Early design exploration for DSP systems can be viewed as the task of generating multiple behavioral descriptions, which yield different throughputs, with different storage requirements. By changing the loop structure or the order of instructions, a lower storage can be obtained, while possibly deteriorating the performance. The goal is to find the lowest cost implementation, while satisfying the time constraints. Alternatively, the goal may be to generate several candidate designs with varying storage and performance characteristics. Memory size estimation is responsible for determining the total size of storage requirements.

In this report we present an approach to high-level memory size estimation, in the context of a Hardware/Software design environment, for parallel behavioral descriptions including multi-dimensional arrays and multiple nested loops.

An overview of our entire system is presented in Figure 1. The design process starts from an english specification, from which the developers create an initial procedural code (represented by the C description). The performance and cost of the final design are very much influenced by this initial representation of the system. The first step in the system generates different algorithmic representations, which will provide a set of starting points for the subsequent design stages. Each of these representations have a different cost/performance characteristic, and they allow the exploration of a larger design space.

The Hardware/Software partitioning task chooses, based on cost and performance criteria, what parts of the system will be implemented by software, running on an embedded processor, and what parts will be implemented as an on-chip ASIC. The hardware branch of the design process performs the High-Level Synthesis tasks, generating a controller unit (CU), and a datapath (DP). The tasks performed in the software compilation branch produce the code for the target processor.

(Re)scheduling, virtual memory assignment, register assignment, memory exploration and physical memory assigment/address generation tasks have similar meaning in the hardware and software branch.

(Re)scheduling is the process of reordering the instructions within basic blocks or between basic blocks, to explore the design space in terms of parallelism, functional and storage resources, and time constraints.

Virtual Memory Assignment is the mapping of all the program variables (both scalars and arrays) to a common virtual memory address space. This virtual address space will be subsequently translated to a physical memory space, respectively register files, on-chip and off-chip memories. Register Assignment selects the mapping of variables (scalars or array elements) to the on-chip registers. We assume a register file (RF) is present both in the processor, as well as in the ASIC. Memory Hierarchy Exploration considers different hierarchycal memory organizations (eg. register files, on-chip SRAM, caches, off-chip DRAMs, etc) and the logical assignmnet of the scalar and array variables into these hierarchycal memory organizations [PaD97]. Physical Memory Assignment/Address generation maps the individual variables to specific memory locations in the main memory or the scratch-pad memory, and generates the address circuitry/code.

In the software part, Code Generation produces the code for the target embedded processor. In the hardware part, allocation is responsible for allocating resources from a library, and binding is responsible for assigning these resources to the actual operations in the code.

The target architecture model we use is presented in Figure 1. We assume a processor core, running the software part, and an ASIC, comprised of a controller and a datapath, implementing the time critical parts of the system. Each of these have a register file (RF), which offers very fast access. We also have an on-chip local memory structure, composed of a scratch-pad memory, data cache, and instruction caches [PaD97], offering a slower access time than the RF, but still much faster than the background memory (1 to 2 clock cycles). The off-chip memory has the slowest access time (10-20 clock cycles), but has the largest size. We assume the ASIC and the processor share a common memory address space, and the communication is done through a bus.

# 2. System-Level Memory Size Estimation.

The goal of memory size estimation is: given a procedural algorithm specification (e.g., in C), to estimate the total number of memory locations necessary. Our work is addressing the early stage of the design process, but can be used in later points as well. We predict the size of memory needed for the given instruction ordering, assuming that in-place mapping is used during the array to memory assignment step.

Our goal is to provide a fast and accurate estimate. Since this is not allways possible, we allow trade-off between runtime, and the quality of the estimate. For the case when the estimation is not exact we provide bounds (upper, and lower) to alow early design decisions to be made.

The total memory size is dependent on the final schedule. Therefore, an exact estimation corresponds to a specific schedule. Nevertheless, if the loop structure remains the same, and only the order of instructions within the basic blocks changes, the variation in memory size is small. The reordering of operations within the loop boundaries, whenever it is possible, does not produce a multiplicative change in memory size over the size of the loops. Thus, the memory size estimation is close to the exact value for any schedule within the same loop boundaries.

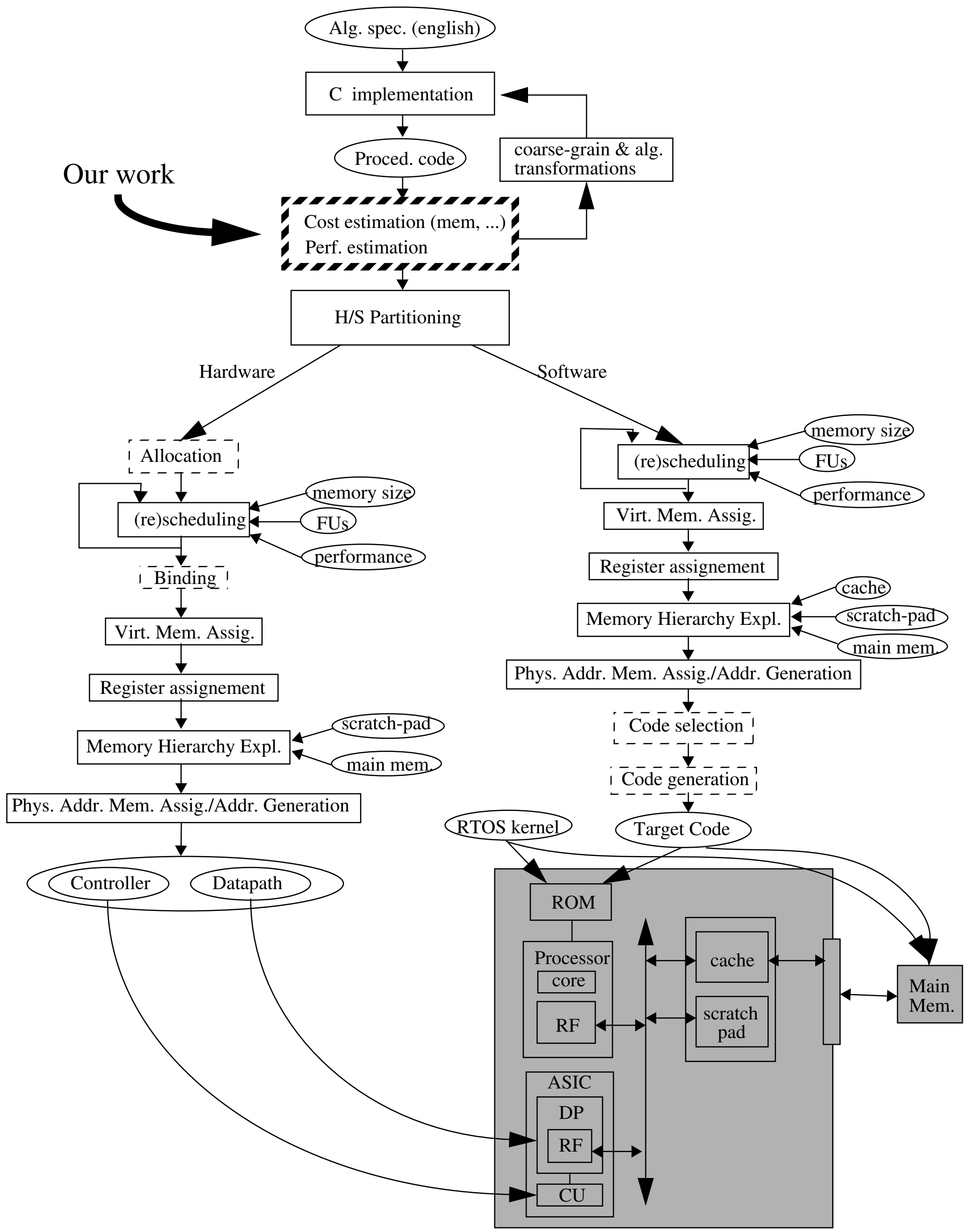


Figure 1: System overview.

The final memory size is also depending on the actual virtual memory assignment. We assume that this will be done at a later stage in the design, using in-place mapping. Therefore the predicted memory will be minimal for that given schedule.

## 2.1 Problem Description.

Given a procedural static single assignment (SSA) specification, comprising multiple nested loops and multi-dimensional array accesses, with the array indexes and loop boundaries linear functions of the outer loop variables, and parallelism explicitly described in the code, derive the number of memory locations necessary for the computation.

Knowing that not all elements of the arrays are alive at the same time, it is possible to share the memory locations between different elements or arrays. Thus, the number of memory locations necessary is less than the declared size of the arrays. Due to the presence of multiple loops, and arrays with multiple dimensions with complicated index functions, it is usually very hard to determine the storage requirement by hand (see e.g., the code in Figure 2). Our method will provide an estimate of the memory size needed to store the arrays in the code, taking the sharing of memory locations into account, and allowing the designer to take early design decisions.

```
optDelta[0,0]:=0;
for j:=-N to N do begin
  for k:=-m to m-1 do begin
    for l:=-n to n-1 do begin
        ad[M,2*N+j,M+k,2*N+j+l]:=a[1,M+k,2+N+j+l]@1;
    end;
  end;
  delta[M,2*N+j,0,0]:=0;
  delta[M,2*N+j,0,1]:=delta[M,2*N+j,0,0]+a[0,M,2*N+j]-
                                         ad[M,2*N+j,M-m,2*N+j-n];
  for l:=-n+1 to n-1 do begin
       delta[M,2*N+j,0,l+n+1]:=delta[M,2*N+j,0,l+n]+
                     a[0,M,2*N+j]-ad[M,2*N+j,M-m,2*N+j+l];
  end;
  for k:=-m+1 to m-1 do begin
       delta[M,2*N+j,k+m,1]:=delta[M,2*N+j,k+m-1,2*n]+
                     a[0,M,2*N+j]-ad[M,2*N+j,M+k,2*N+j-n];
       for l:=-n+1 to n-1 do begin
            delta[M,2*N+j,k+m,l+n+1]:=delta[M,2*N+j,k+m,l+n]+
                             a[0,M,2*N+j]-ad[M,2*N+j,M+k,2*N+j+l];
       end;
  end;
  optDelta[0,j+N+1]:=delta[M,2*N+j,2*m-1,2*n]+optDelta[0,j+N];
end;
```

Figure 2: Motion Detection algorithm kernel.

We assume that all variables and arrays in the input code are of the same bitwidth. We consider that each instruction is executed in one time slot. An instruction may comprise more than one operation (e.g., a[i] = b[j] + c[j] * d[i] contains a multiplication and an addition). All the operations in the instruction are considered to be executed at the same time. Hence, the time slots represent a symbolic concept. They do not show clock cycle level information on a real time axis, but rather a relative ordering between the instructions.

To show the importance of a fast memory size estimation technique, we present in Figure 2 part of the Motion Detection algorithm kernel, used in immage processing. This example contains multiple nested loops, and multi-dimensional arrays, and it is very hard to determine by hand the memory usage pattern and the storage requirement.

## 2.2 Previous Work.

The related problems of memory estimation, memory allocation, and signal-to-memory assignment have recently gained attention in the EDA community. The early works were scalar-oriented, solving the traditional problem of *register* allocation and assignment: the scalar variables in an already scheduled register-transfer level (RTL) specification can be optimally assigned to a minimum number of registers by means of a “left-edge” algorithm [KuPa87]. This strategy is mainly due to the fact that applications targeted in conventional high-level synthesis contain a relatively small number of signals. Under the (usually implicit) assumptions mentioned above, binary ILP formulations [BMBL88], graph coloring [StJe92] and clique partitioning [TsSi86] techniques have all provided satisfactory results.

When dealing with multidimensional signal processing applications, scalar-oriented techniques are no longer applicable due to the enormous number of scalar signals in the algorithmic specifications. In particular, an exact evaluation of the memory requirements is usually not possible any more, and the possibility of efficiently estimating the necessary storage has to be investigated.

One of the earliest approaches of handling arrays of signals is based on clustering the memory modules storing the individual arrays in the larger memory modules such that a cost function (based on layout and performance estimations) is minimized [RaGC91]. 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, Schmit and Thomas [ScTh97] proposed a more refined array clustering (based on vertical and horizontal concatenation), along with a technique for binding groups of arrays to memory modules drawn from a given library. However, it seems that the technique has the same drawback as the one mentioned above. Jha and Dutt [JhDu95] have also studied the binding of a logical memory to physical memory modules from a library, while Panda and Dutt [PaDu97] have tackled the problem of array mapping so as to minimize the transition activity on the address buses. In addition, Bakshi and Gajski [BaGa95] proposed a methodology to explore different memory organizations for pipelined DSP specifications.

The memory related works at IMEC (Belgium) assume the behavioral specifications are non-procedural. Van Swaaij, Franssen et. al. [SFCM93] 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. Balasa et. al. (e.g., [BaCM95]) consistently employed data-flow analysis of the non-procedural source code to estimate and allocate the memory: they proposed a polyhedral data-flow model operating on groups of scalars defined by linearly bounded lattices. Verbauwhede [VeSR94] employed a similar view but her memory estimation techniques works well for specifications with constant boundary loops and with limited non-procedurality (due to a set of assumptions on the loop nests); her technique is less general, but faster.

A memory synthesis approach presented in [JLVW95] is targeted to dynamic data structures representing C++ abstract data types. As a consequence, the level of specification is lifted to a higher level than currently used in behavioral synthesis. In [SWCJ97], the memory management for dynamic data structures is combined with a flow graph balancing step to optimize the memory bandwidth, and thus decrease the necessary number of memory modules and ports.

The PHIDEO system (Phillips) [LMVW93] employs a hierarchical stream model in order to handle multidimensional signals with complex affine indexes; their model is unable to handle affine indexes in loop nests with non-constant boundaries.

## 2.3 Our Approach.

We present an algorithm (called MemoRex) to estimate the memory size for a procedural SSA description containing multi-dimensional arrays, allowing the possibility to trade-off computation time against estimation accuracy.

The memory size estimation algorithm derives an estimate of the memory trace, and the upper and lower bounds of the memory size. The lower and upper bound represent the range within which the exact memory size lies. With more computation time available, tighter upper and lower bounds are computed, bringing them closer together. Since the memory size lies between the two bounds, narrowing the gap between the bounds results in a better estimate. When the upper bound equals the lower bound the estimate is exact.

The run time for the exact evaluation depends on the nature of the input algorithm. In some instances we may obtain the exact estimate with very little computation time. However, usually the memory access patterns are very complex, and therefore the estimation time can be large. The user can then choose to allow for longer computation time, and determine a narrower range for the estimation, or alternately, he may terminate the estimation procedure when a satisfactory error margin is reached.

We start by explaining intuitively the memory estiamtion algorithm. Figure 3 shows the flow of our algorithm. In order to determine the memory size for a given input code, we first approximate the memory trace. 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 usually cannot determine the exact memory trace (the continuous line in the graphic from Figure 3) due to the high computational effort required. If this is the case, a bounding area encompassing the memory trace - the shaded rectangles from the graphic in Figure 3 - is determined instead.

The storage requirement of the 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 - shown in Figure 3 as the range between the two dotted lines - represents the result of our estimation tool.

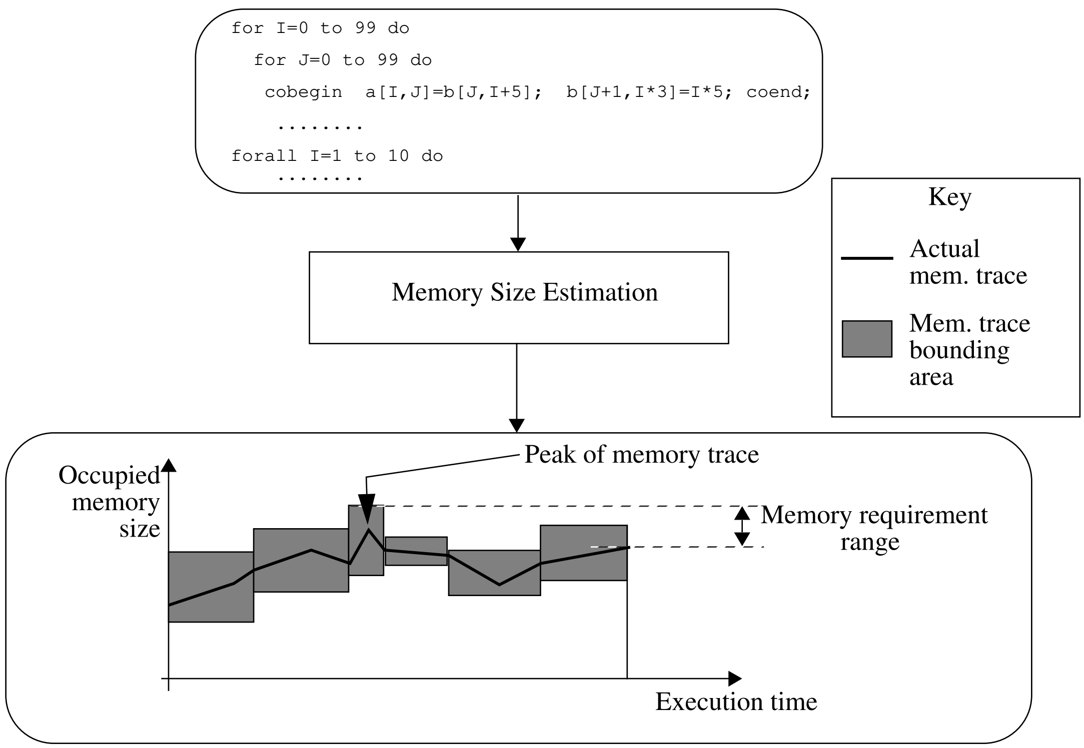


Figure 3: The flow of the MemoRex algorithm.

Employing the terminology introduced in [SFCM93], the algorithm computes the number of scalars produced by each definition domain and consumed for the last time by each operand domain. The definition/operand domains are the array references in the left/right hand side of the assignments.

Then, we determine 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 3. In fact, this determines a set of points on the memory trace. To determine or approximate the unknown parts of the trace, Next, we determine a set of covering bounding rectangles, represented as shaded rectangles in Figure 3.

Based on the bounding rectangles computed earlier, and the memory size between the loop nests, in the next step we approximate the trace peak, determining the range for the memory requirement, as a lower and an upper bound for the memory requirement.

Finaly, we refine 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. These steps can be repeated until a convenient accuracy is reached.

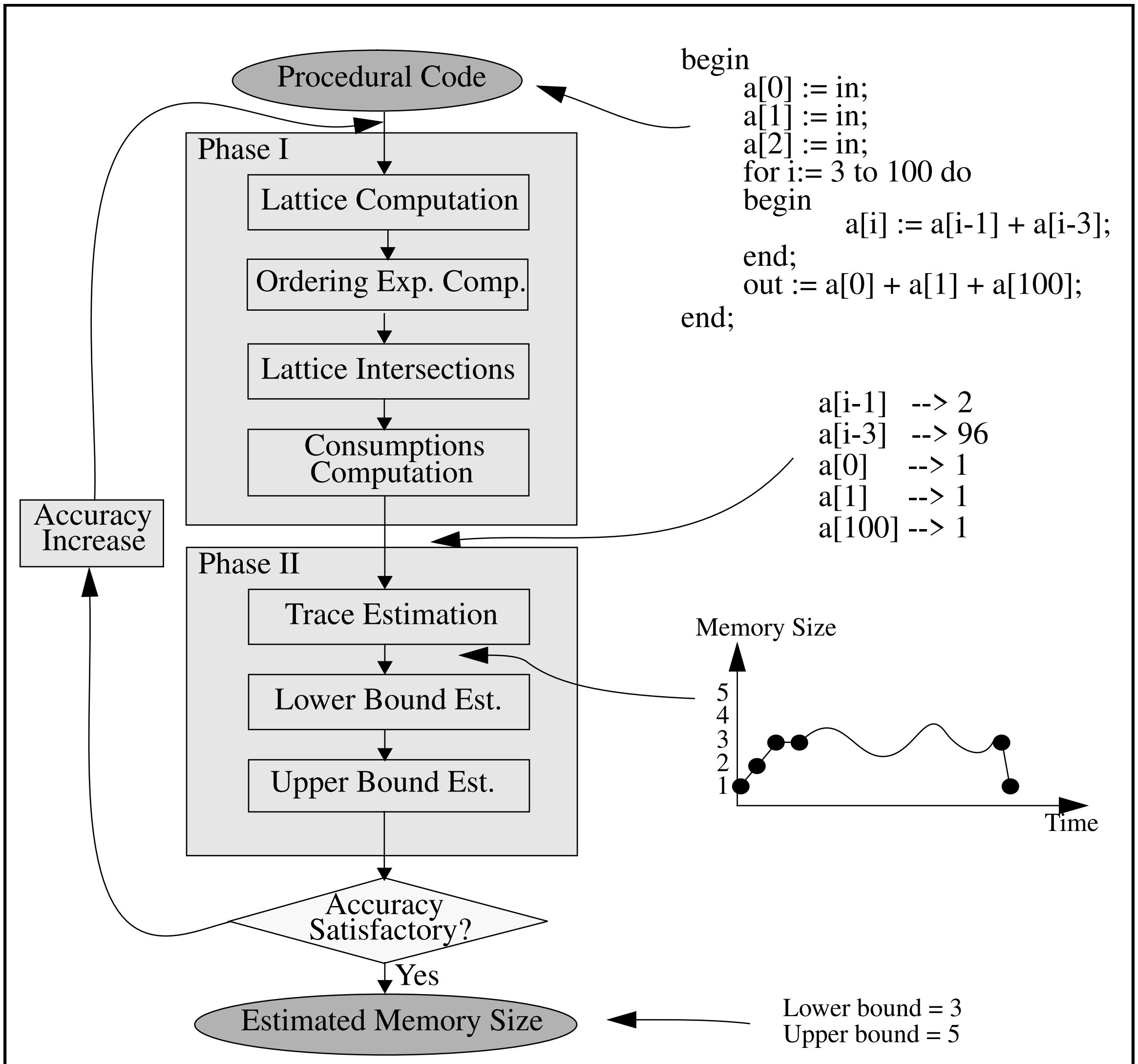


Figure 4: Algorithm outline.

Figure 4 shows an outline of our algorithm. The input is represented by a procedural code (e.g., C-like program). The output is an upper and lower bound for the storage requirements for that program. The algorithm is composed of two parts. The Phase I computes for each array read access from the code, the number of elements consumed. We say that a value is consumed when it is read for the last time, or its lifetime ends. First, for each array read access, we compute a lattice representing the total set of elements referenced by that array read. Next, for each assignment in the code, an ordering expression is computed, representing the moment when the instructions are executed on a relative (as opposed to clock cycle based) time axis. By performing intersection of these lattices, the consumptions are determined. This part is based on the work by Balasa et. al., adapted for a procedural representation. Besides the dataflow information used in the lattices and lattice intersections as presented in [Bala95], we generalized the model to capture also procedural ordering information, by extending the lattice intersection to account also for the relative ordering between instructions.

Phase II uses the consumptions generated by Phase I, in order to compute an approximate memory trace, and the upper and lower bounds for the storage size. If the error margin (given by the percentile gap between the lower and upper bound) is satisfactory, then the estimation is complete. Otherwise, this process can be repeated for an increased level of accuracy. Note that the upper and lower bounds converge towards the exact memory size. For some applications they can be equal for the first level of accuracy, in which case the estimation is exact from the first pass.

We begin by defining a set of terms which will be used later in the presentation. An *integral polytope* is a bounded set of points $P \subset Z^n$ satisfying a finite set of linear inequalities. In the following we will call it simply a *polytope*. The set $\{y | y = Ax, x \in Z^n\}$ represents a *lattice* generated by the matrix A. A *linearly bounded lattice* (LBL) is a lattice where x belongs to a polytope.

The *iterator space* of an assignment is the polytope defined by the surrounding loop variables. The surounding loop bounds, as well as the surounding conditionals represent the set of linear inequalities of the polytope, and each tuple of possible values for the loop variables, represent an integer point in the polytope.

The array references from an assignment statement are called the *definition domain* (left hand side access), and *operand domain* (right hand side accesses). Each domain can be represented by one or several lattices. Without loss of generality, in the following we consider that each domain is represented by only one lattice. The set of inequalities defined by the surrounding loops and conditionals represents the polytope, and the index function represents the mapping matrix A.

In the following we present first the mathematical model used throughout our algorithm. Next, we continue with Phase I and Phase II of our method, as well as the accuracy increase technique.

## 2.3.1 Mathematical Model.

We now present the mathematical model underlying our estimation approach. The traditional representations of procedural code as a hierarchy of loop statements, conditionals and assignments (e.g., the program graph type representation) is not well suited for our purpose of memory size estimation. Generally in a dataflow analysis problem, a dataflow graph is constructed. For the case of multi-dimensional arrays, [Bala95] uses a polyhedral dataflow graph (PDG). The PDG is very good for capturing the pure dataflow information, comprised of the data dependencies between instructions. Thus, the PDG is well suited for analysis of non-procedural representations. For our case though, where the input is a procedural specification, it is not satisfactory, because it doesn't capture the control dependencies between instructions, i.e. the relative ordering between instructions for which there is no data dependency.

The mathematical model we present here, the Polyhedral Control DataFlow Graph (PCDFG), is suited for capturing both data dependence information, as well as procedural ordering information. It is based on the PDG presented in [Bala95], generalized to describe the relative ordering between instructions. An analogy with the traditional Control DataFlow Graph (CDFG) [GDWL92] can be made, by noting that just like the CDFG, the PCDFG captures the control and dataflow information from the code, but using a polyhedral representation. (names: Polyhedral Procedural Dependence Graph, Polyhedral Ordering Dependence Graph, Polyhedral Procedural DataFlow Graph, Procedural DataFlow Graph)

```
begin
    a[0] := in;
    a[1] := in;
    a[2] := in;
    for i:= 3 to 100 do
    begin
        a[i] := a[i-1] + a[i-3];
    end;
    out := a[0] + a[1] + a[100];
end;
```

Figure 5: Explanatory example.

Throughout the following sections, we will use two examples to show the concepts. The first one is a simple explanatory example described in Figure 5. This is not a realistic example, but allows for clarity of the explanations. We will refer to it as the "explanatory example". The second one is a parallel example, and we use it to show the effect of parallelism on the memory estimation technique. The parallel example will be presented in the section for Parallel Computation.

In our model, the program is represented as a set of assignment statements. Each assignment has a corresponding iterator space, as well as a set of array accesses. Each array access has an index space. Besides this information, we attach to each assignment statement an order expression.

The order expression is a symbolic expression, depending on the set of loop variables from the scope of the assignment. It does not represent the exact moment on a real time axis when the instructions are executed, but rather the relative order of execution of the assignment statements.

Each assignment statement is a tuple, comprised of the iterator space of that assignment (I), the ordering expression (OE), the index space of the array write access (AW), and the index spaces of the array reads ($AR_i$).

$A = (I, OE, AW, AR_1, AR_2, ...)$
I = iterator space formed by the surrounding loop variables
OE = order expression
AW = array write = index space of the array write access
$AR_i$ = array read = index space of the array read access

Figure 6 shows in detail this tuple for the assignment a[i] := a[i-1] + a[i-3]. The iterator space corresponding to the loop is represented by the possible values of the loop variable. The loop variable has values between 3 and 100. Therefore the iterator space is {3..100}. The definition domain a[i] has the index space {3..100}. The operand domains a[i-1], a[i-3] have the index spaces {2..99} and {0..97}. The order expression corresponds to the ordering of the instructions in the procedural code, and is a symbolic expression of the surrounding loops' variables.

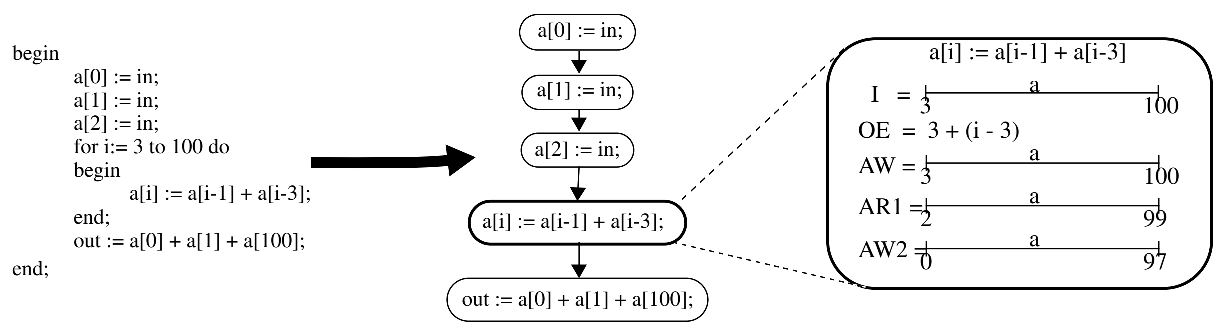


Figure 6: Mathematical Model.

Such a representation is produced for each assingment statement from the code. Considering that each statement represents a node, the edges represent the implicit order of the instructions in the initial code. The order of execution of the statements is given by the OE field of each node.

The iterator space represents the n-dimensional space of the loop variables, where n is the number of loop nests surrounding the assignment, and is represented as a polytope. The index spaces of the array accesses are represented by linearly bounded lattices. Note that if the assignment is not surounded by any loop, the iterator space is an empty set, and the index spaces are constants.

### 2.3.2. Phase I: Computation of Array Productions and Consumptions.

To compute the amount of memory needed, we need to know the number of array elements produced and consumed. Since each assignment statement in the code represents a set of individual instructions (depending on the surounding loops) the array write access from the assignment may produce array elements, whereas the array read access may consume elements. In order to determine the memory size, the number of productions and consumptions for each array access is needed.

Since the input code is in SSA form, each array write will produce a new array element. On the other hand, an array read access will not consume a value every time. A value is consumed, only when its lifetime ends, i.e. it is read for the last time. Therefore, a subset of the array elements referenced by a read, are consumed for the last time (i.e. these elements are dead). We call this subset the array consumptions of that array read access (operand domain).

Since each array write produces a new value, the number of productions is easy to determine. It is the size of the lattice representing the write access. To compute the array consumptions, we need to know which elements are read again later in the schedule, therefore a dataflow analysis is required.

An operand domain consumes only the array elements which are not read again later in the code. Therefore, the consumptions of an operand domain can be computed by removing from the domain's index space all the elements which are read at a later point.

First, we compute the lattice representing each array read. Then, we intersect each such lattice with all the other lattices which occur at a later moment. The intersections represent the elements which are *not* consumed in that read access. Therefore, the elements from the operand's lattice which are not in any intersection, represent the consumptions of that operand domain.

The elements consumed in an operand domain are the remaining elements in the domain's lattice, after subtracting all the intersections computed as stated above. The number of elements consumed by the domain represent the number of array consumptions of that domain. These numbers are used in the Phase II to computate the upper and lower bound.

The dataflow analysis is based on the work by F. Balasa. In [Bala95] a method to determine the dependences between groups of elements in an array, using Polyhedral Dataflow Analysis, is presented. Each array access is represented as a lattice, and by intersecting the lattices corresponding to different array reads, the correponding dependences are determined. Nevertheless, the work in [Bala95] is targeted towards non-procedural representations. We extended their method to cover procedural code, by including the order of execution which is implicit in a procedural language. The intersection of two lattices is restricted to those elements which satisfy the ordering condition: the elements in the first lattice are read before the elements in the second lattice. Therefore, the intersection will give the elements which are read in the first lattice, but consumed in the second one.

## Lattice computation

The lattice coresponding to an array access is determined by its index function, and by the surounding loops and conditionals. The set of linear inequalities generated by the loop bounds and the conditions in the if statements represent the polytope (the iterator space) of the assignment. The index function (or set of index functions for a multi-dimensional array) of the array reference generates the mapping matrix of the lattice.

A linarly bounded lattice can also be represented in the following form:

$$L_1 = \{x = T_1 \times I_1 + u_1 | A_1 \times I_1 \geq b_1\},$$

where $T_1$, $u_1$ represent the mapping matrix (the array index functions), and $A_1$, $b_1$ represent teh set of inequalities of the polytope (iterator space).

E.g., for the a[i-1] array read from the explanatory example (Figure 5), the lattice representing the referenced array elements is:

$$L_{a[i-1]} = \left\{ x = \begin{bmatrix} 1 \end{bmatrix} \times \begin{bmatrix} i \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix} \middle| \begin{bmatrix} 1 \\ -1 \end{bmatrix} \times \begin{bmatrix} i \end{bmatrix} \geq \begin{bmatrix} 3 \\ -100 \end{bmatrix} \right\}$$

## Ordering expression computation.

In order to compute the intersections between lattices we need to know the corresponding ordering expressions (OE). Each statment has its own OE, which depends on the surounding loops and if statements structure.

The ordering expression corresponding to an assignment statement is a symbolic linear equation, having as variables the surrounding loop variables. For any set of values for the loop variables, the order expression has a specific value, which shows the time slot when the assignment takes place. This time slot is not referring to a real time axis. It rather captures the relative ordering between the instructions. E.g., in our explanatory example from Figure 5, the assignment a[i] := a[i-1] + a[i-3] has the ordering expression OE(i) = 3 + (i-3)*1 = i.

The OE for each statement can be computed recursively, based on the control structure of the program. For two consecutive assignments a1 and a2, we have the following relation: OE(a2)=OE(a1)+1, which shows that consecutive assignments are executed in consecutive time steps. The ordering expression of the assignments in a loop is computed based on the current iteration number, and the lenght of one iteration of the loop. We cosider the loops normalized. The ordering expression for the assignments within a conditional is computed just like in the case of sequential instructions. We insure the mutual exclusivity of the two branches by adding the if condition to the set of inequalities describing the index space. In the case of parallel instructions, the ordering expression is the same for the instructions executed in parallel, and is equal to the expression immediately before the instructions, incremented by one.

## Lattice intersection.

Considering two lattices, each with a corresponding ordering expression, we want to determine the elements common to the two lattices for which the ordering expression from the first lattice is less than the ordering expression from the second lattice. This condition represents the fact that the elements from the intersection have to be read in the first lattice earlier than in the second lattice. This leads to the fact that the elements from the intersection are not consumed in the first lattice.

The intersection between two lattices represents another lattice. In particular, by adding the ordering condition, the intersection of the two lattices will be still a lattice, since the condition is just an additional linear inequality.

Generally, the intersection between two lattices is determined by computing a solution for the diophantine equation generated by the equality between the mapping functions of the two lattices. This solution will provide the mapping function of the resulting lattice. The union of all the inequalities from the two lattices forms the set of inequalities of the resulting lattice.

Let $L_1$ and $L_2$ be the two lattices to be intersected, and $O_1(I_1)$ and $O_2(I_2)$ be the two ordering expressions depending on the loop variables (denoted by $I_1$, $I_2$). The ordering functions $O_1(I_1)$ and $O_2(I_2)$ can be represented also as matrix multiplications, with $O_1$ and $O_2$ being the coefficient matrixes.

$$L_1 = \{x = T_1 \times I_1 + u_1 | A_1 \times I_1 \geq b_1\}$$

$$L_2 = \{x = T_2 \times I_2 + u_2 | A_2 \times I_2 \geq b_2\}$$

Intersecting these two lattices [Bala95] means first solving the linear diophantine system:

$T_1 I_1 - T_2 I_2 = u_2 - u_1$.

If the system has no solution, the intersection is empty. Otherwise, let the solution of the diophantine system be:

$$\begin{bmatrix} I_1 \\ I_2 \end{bmatrix} = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} \times I + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

The union of the linear constraints of the two lattices along with the ordering condition, represent the linear inequalities of the resulting intersection lattice:

$$A_1 \times I_1 \geq b_1$$

$$A_2 \times I_2 \geq b_2$$

The ordering condition is added to these inequalities:

$$O_1 \times I_1 \leq O_2 \times I_2$$

The resulting intersection is:

$$L = \{x = T \times I + u | A \times I \geq b\}, \text{ where}$$

$T = T_1V_1, \quad u = T_1v_1 + u_1$

$$A = \begin{vmatrix} A_1V_1 \\ A_2V_2 \\ -O_1V_1 + O_2V_2 \end{vmatrix}, b = \begin{vmatrix} b_1 - A_1v_1 \\ b_2 - A_2v_2 \\ O_1v_1 - O_2v_2 \end{vmatrix}$$

Considering the array reads a[i-1] and a[i-3] from the explanatory example, we compute the corresponding lattices as:

$$L_{a[i-1]} = \left\{ x = \begin{bmatrix}1\end{bmatrix} \times \begin{bmatrix}i\end{bmatrix} + \begin{bmatrix}-1\end{bmatrix} \middle| \begin{bmatrix}1 \\ -1\end{bmatrix} \times \begin{bmatrix}i\end{bmatrix} \geq \begin{bmatrix}3 \\ -100\end{bmatrix} \right\}$$

$$L_{a[i-3]} = \left\{ x = \begin{bmatrix}1\end{bmatrix} \times \begin{bmatrix}i\end{bmatrix} + \begin{bmatrix}-3\end{bmatrix} \middle| \begin{bmatrix}1 \\ -1\end{bmatrix} \times \begin{bmatrix}i\end{bmatrix} \geq \begin{bmatrix}3 \\ -100\end{bmatrix} \right\}$$

To find the elements which are common to $L_{a[i-1]}$ and $L_{a[i-3]}$, but are read in a[i-1] before a[i-3], we intersect them, considering the ordering expression condition $i_1 <= i_2$, where $i_1$ is the index from the first lattice, and $i_2$ is the index from the second lattice, and the result is:

$$L = \left\{ x = \begin{bmatrix}1\end{bmatrix} \times \begin{bmatrix}i\end{bmatrix} + \begin{bmatrix}-1\end{bmatrix} \middle| \begin{bmatrix}1 \\ -1 \\ 1 \\ -1 \\ 0\end{bmatrix} \times \begin{bmatrix}i\end{bmatrix} \geq \begin{bmatrix}3 \\ -100 \\ 1 \\ -98 \\ -2\end{bmatrix} \right.$$

The last row in the constraints matrix of the intersection lattice results from the ordering condition $i_1 <= i_2$, and is represented by a tauthology (0>=-2). This lattice represents all the elements between a[2] and a[97].

## Consumptions computation.

Once the lattice intersections have been computed, we can determine the number of elements in each lattice which do not belong to any intersection. This is done by constructing an Inclusion Graph ([Bala95]), and determining the size of the union of all the intersections. The elements which are in the original lattice, but not in the union of the intersection, are the consumptions of the array access.

In Figure 7 we present the number of array elements consumed in each of the operand domains from the example. We only represented the array consumptions, since the productions are computed as the size of the original lattices, and do not involve any dataflow analysis.

cons(a[i-1]) = 2
cons(a[i-3]) = 96
cons(a[0]) = 1
cons(a[1]) = 1
cons(a[100]) = 1

Figure 6: Consumtions for the operand domains.

## Lattices with non-injective mappings.

An injective function f: X -> Y is the one which for any two different parameters $x_1$, and $x_2$ in X, will result in different values of the function: $f(x_1) \neq f(x_2)$. When the index function of an array reference is non-injective, i.e. for different iterations of the loop the same array element is referenced, the computation of intersections of lattices does not give valid results.

In a non-injective array read, in different time-slots we may refer to the same array element. A value may be referenced twice by that read, at different moments. If some other array read references the same value sometime between the two references from the first read, then the intersection of the two lattices with the condition that the first read is done first, and the second occurs later, will include the common value. Nevertheless, this common value is actualy consumed by the first read access, at the moment when it is referenced the second time. Thus, the intersection is incorect, since it should include only the values which are *not* consumed by the first read access.

In order to avoid this problem, the lattices have to be reduced to an injective subset before computing the intersections. The injective subset has to be comprised of all the elements referenced from the initial lattice, but the elements which are accessed more than once, only the last access has to be included. Therefore, even though the lattice will contain the same elements, the polytope which generates it will be only a subset of the original polytope.

An example when non-injective lattices appear is when an array access does not depend on all the surounding loop variables. This case can be easily solved, by considering only the highest value for the loop variables which don't appear in the index function. Note that the actual code is not changed. The input description remains the same throughout our algorithm.

## Parallel Computations.

Our method allows the estimation of memory size for a parallel specification. The language construct to describe a parallel isntruction is a parallel block, with all the operations inside the block executed in parallel. The parallelism can be described at any granularity level (e.g., we can describe two loops, or two assignments to be executed in parallel). We consider a synchronous parallelism model (the parallel operations are executed synchronously in the same time steps). Figure 7 shows an example of such a parallel specification. We use this example to show the effect of parallelism on the memory estimation technique.

In Figure 7, the arrays c and b are created in parallel. The first 5 iterations of the cretion of c are executed in parallel with the creation of b. Since we know the time steps when each of the instructions are performed, we determine the ordering expressions for the parallel operations in the same manner as for sequential operations. We determine the number of elements consumed for the last time in each operand domain by performing lattice intersections. We show the number of elements produced and consumed for the last time in the right hand side of the Figure 7. We use these numbers in the second phase of the algorithm, to determine the upper and lower bound for the memory size.

Once we determined the ordering expressions for the parallel instructions, we treat the parallel and the sequential instructions in a similar fashion. The number of array elements consumed and produced in an assignment is determined in the same way, regardless of wheather it was accessed from sequential or parallel instructions.

```
for i:=1 to 10 do begin
        a[i]:=in;                       prod(a[i])=10
end;
parallel begin
        for i:=1 to 10 do begin
                c[i]:=in;               prod(c[i])=10;
        end;
        for i:=1 to 5 do begin
                b[i]:=a[i]+a[i+5];      prod(b[i])=5, cons(a[i])=5, cons(a[i+5])=5
        end;
end;
for i:=1 to 5 do begin
        out[i]:=b[i];                   prod(out[i])=5, cons(b[i])=5
end;
for i:=6 to 15 do begin
        out[i]:=c[i];                   prod(out[i])=10, cons(c[i])=10
end;
```

Figure 7: Example for parallel description

When different assignments are executed in parallel, or when there are multiple accesses in the same assignment, we may have multiple simultaneous accesses to the same array element. If the same array element is consumed twice at the same time, there is an element of nondeterminism. It is unknown which of the two domains reads it for the last time (consumes it). If there are no other reads to the same element later in the code, one of the two domains will consume it. In fact it is not important which of the two will read it for the last time, but it is important to count the consumption in one and only one of the two accesses. If besides these two accesses, there are also other references to the same element later in the code, none of the two domains will consume the element. The latter case is straight forward. The former has to be further considered.

If two domains read the same element at the same time, and one of them consumes it, we have to consider an ordering between the two reads. Only one of the two domains will actually consume the element. Whenever intersecting two lattices having the same ordering expression (their domains are executed in parallel), an additional ordering between the accesses has to be considered. By generating an order between the instructions executed in parallel, we avoid counting a value as consumed by both its parallel references. Note that this ordering does not change the actual order of instructions in the code. It is only needed for the correct computation of array consumptions.

### 2.3.3 Phase II: Upper and Lower Bound Computation.

Based on the array consumptions corresponding to each operand domain, an estimate of the memory trace is computed, as well as an upper and lower bound for the memory size.

In each time slot during the execution of the code, the memory need will increase or decrease depending on the number of elements produced, and the number of elements consumed in that time slot. The number of memory locations needed in each time slot during the execution of the input algorithm is called the memory trace. The highest value reached by the memory during this trace, gives the storage requirement for the algorithm. If the memory trace is known for each moment during the execution of the input code, then the exact memory need can be determined. If the memory trace is known only in some points during the execution of the input code, then only an upper and a lower bound for the memory requirement can be determined. Depending on the complexity of the input code, we will determine the exact memory requirement, or a set of bounds.

Our algorithm treats the parallel computations and the sequential computations in a unifrom manner. In each time step the array elements can be accessed either from one assignment with multiple reads, or from multiple parallel assignments. Since at this stage we are interseted only in the array accesses (not in the functional resources), the two cases are similar.

The memory size is determined by first computing the memory trace for the boundary points of the outermost loops in the code (and similarly for the assignments which are outside any loop nest). Then, knowing the memory trace at the beginning of a loop and at the end of a loop, we compute a min and max bound for the memory requirement for that loop. By knowing the bounds for each outer loop, we determine a set of bounds for the entire code.

## Memory trace computation.

The memory trace at outer loop boundaries can be computed by traversing the code. Assuming we know the size of input variables for the given code, we compute the memory size after each outer statement (assignment or loop), by adding to the current memory size, the number of elements produced in that statement, and subtracting the number of elements consumed. The total number of consumptions from each outermost statement is computed by adding the consumptions corresponding to all the operand domains inside that statement. The total number of array elements produced is also computed as the sum of the sizes of the definition domains inside it.

In Figure 8 we show the memory trace estimate for the explanatory example from Figure 5. The exact memory need at each moment is known at the loop boundaries and outside the loop, but is not known inside the loop. Therefore, the known points form an estimate of the complete memory trace. The straight lines between known points show the exact trace, whereas the curves show the unknown parts of the trace. The memory size needed for the computation of the whole procedural code is given by the maximum memory needed at any moment. It is the maximum value reached during the trace. The upper and lower bounds we compute, represent an interval where the exact value of the memory needed for the entire code is found. They show the bounds where the unknown part of the trace lies. The element of non-determinism will be therefore confined to a narrow error margin.

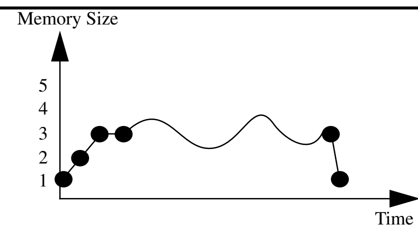


FIgure 8: Memory trace estimate.

The total memory needed for the entire algorithm is equal to the maximum value reached during the memory trace. Since we know the memory trace only ouside the loops, for the trace inside the loops we will compute a pair of bounds which result in the min and max memory requirements. Intuitively, if we assume that in a loop the consumptions are executed ASAP, then the memory trace will first show a decrease, and then an increase (after all consumptions are over). Conversely, if the consumptions are executed ALAP, then the trace will show first an increase, and then a decrease (once the consumptions begin). The ASAP case will result in the lowest memory need for any ordering of the instructions inside the loop, whereas the ALAP case will result in the highest memory need. Thus, for any actual order of instructions in the loop, the actual amount of memory required will lie between the upper and lower bounds computed.

Each array consumption from an operand domain is executed in a different time slot. The number of consumptions in an operand domain is less or equal to the number of productions in the corresponding definition domain. If they are equal, means that each time the assignment is executed, a value is consumed in that operand domain. On the other hand, if they are not equal, there are instances of the assignment (individual instructions) when no value is consumed in that operand domain. If the consumptions occur at an early moment, the total memory necessary will be less than in the case when the consumptions occur later. Early consumptions are better from the point of view of the memory. Since the productions of values occur in every time slot corresponding to an assignment, by sliding the consumptions of one operand domain earlier or later in the time slots corresponding to that assignment, we determine the best and worst case from the point of view of memory size. In the following we show how this upper and lower bound is computed.

## Lower bound computation.

Assuming we have several operand domains in a loop, the consumptions corresponding to each of these may occur earlier, or later in the time slots of the loop (the moments when such accesses occur are completely determined by the procedural description, but for computation time reasons we do not consider them known). If they all occur ASAP, but still satisfying the constraint that in every time slot only one of the consumptions of a domain can occur, the memory will be the lowest possible for that loop. Since all the consumptions are as early as possible, the memory trace will start by decrasing, due to the consumptions. After reaching a minimum point, it will increase. Therefore, the maximum value attained during the loop is either the starting point (the memory size at the beginning of the loop), or the ending point (the memory size at the end of the loop).

The size of memory for the ASAP case is actually equal to the maximum of the memory amounts at the boundaries of the loop.

Figure 9 shows the ASAP case for the explanatory example from Figure 5. In Figure 9 we show the memory trace. The shaded segments at the top of the figure represent the consumptions and productions of the array accesses. The length of the segments corresponds to the number of elements produced or consumed for the last time by each access. These segments can slide along the time steps corresponding to the loop, which is equivalent to scheduling the accesses earlier or later in the slots corresponding to the loop. By sliding them ASAP, we create a memory trace for the period of the loop. In the time/memory-size diagram, the circles represent the points where the memory trace is known, and the dotted line represents a possible trace for the period when the exact trace is not known.

Knowing that during the 98 assignment instances performed in the loop, the two operand domains consume 2, respectively 96 array elements, we can determine the lower bound. Here the consumptions occur early, each domain accessing at most one element in each time slot. The shape of the trace in this case engenders a low memory size (3 in this example). The memory size for this trace represents the lower bound.

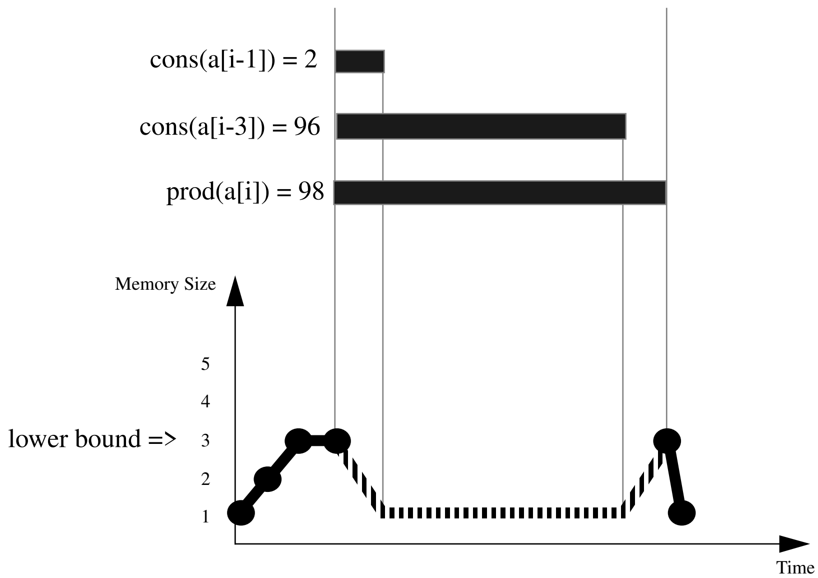

Figure 9: Memory trace for early consumptions.

## Upper bound computation.

The memory for the ALAP case is determined by first ordering the consumptions, and then traversing them in reverse order, starting from the last time slot corresponding to the loop, towards the beginning of the loop. The number of elements consumed in each time slot decreases while we advance towards the beginning of the loop. The slots which have an equal number of consumptions form an interval. We traverse the time slots of the loop in reverse order (from the end of the loop towards the beginning), and in each interval we compute the total number of consumptions and the total number of productions. By subtracting them we determine the total number of elements by which the memory increases or decreases during that interval. By computing this for each interval we determine the exact memory trace for the ALAP case of the whole loop. Thus, we can find the maximum value reached by the memory during the execution of the loop, which represents the upper bound corresponding to that loop.

Figure 10 shows the case when the consumptions occur late (ALAP) in the execution of the loop. This generates a trace where the memory size (the dotted line) first increases, then decreases. The memory need for this trace will be higher (5 in this example).

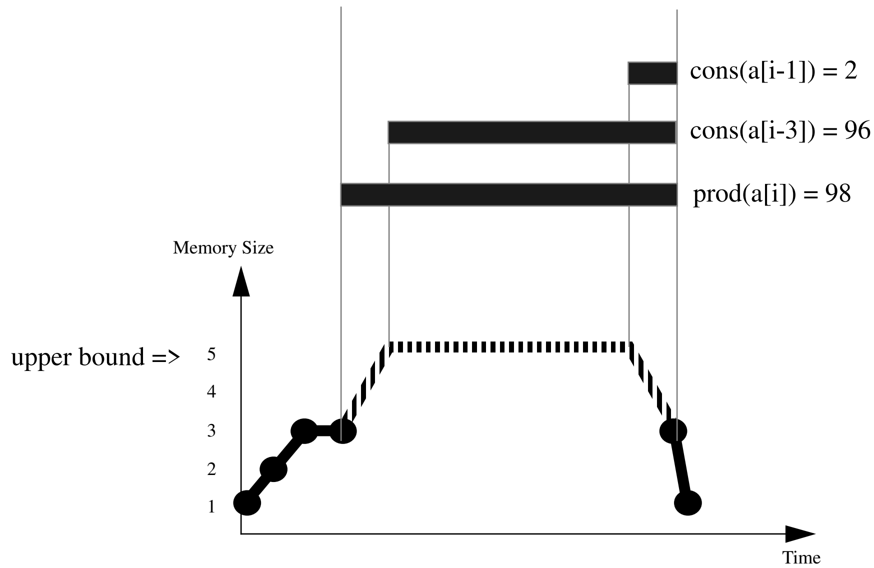

Figure 10: Memory trace for late consumptions.

At this point we know the exact memory needed for both the ASAP and ALAP cases for each loop. The maximum of the amounts for the ASAP cases for all the loops represents the memory needed for the ASAP case for the entire algorithm. The maximum of the amounts needed for the ALAP cases for all the loops represents the figure for the entire algorithm for ALAP.

The memory needed for the entire algorithm for ASAP represents the lower bound, and the amount needed for the algorithm in the ALAP case represents the upper bound for the actual memory required. Since the actual order of execution of the consumptions is somewhere between the ASAP and ALAP for each loop, the size of memory needed is in fact somewhere between the upper and lower bounds.

In our example, the upper and lower bounds computed so far (3 and 5 ) represent the range of values where the exact size of the memory is found. This value could be known exactly, if the complete information about the moment when each array element is consumed inside the loops were considered. Though, to consider separately each element can be very time consuming. The next section presents a method to trade off computation time against accuracy of the estimation.

Note that the ASAP and ALAP versions are not actual schedules. They are used only to determine the bounds on the memory size.

### 2.3.4 Binary Refinement for Improved Accuracy.

Since determining the exact size of memory might be computationally expensive, our method offers the possibility to trade-off computation time against estimation accuracy.

The accuracy of the estimation is high if the upper bound is very close to the lower bound. If they are equal, the estimation is exact. In many instances (as shown in the experiments) this case happens. If they are not equal though, the accuracy can be improved by splitting the outermost loops into equal parts, and then recomputing the bounds. By doing this, we determine more accurately the moment when the array consumptions occur. The memory trace is more accurate, since the memory size is known in more points. Thus the estimation will be more accurate. The new upper and lower bounds are placed between the old bounds, therefore the error margin is decreased after each loop splitting.

By splitting the loop, we mean splitting the iterator space of the loop conceptually into two equal parts. E.g., if the loop is from 1 to 100, then we split it into two loops, one from 1 to 50, and the other one from 51 to 100. Note that the actual behavior is not modified. The splitting is done only conceptually, to determine more accurately the bounds.

For each of the two loops resulted after splitting, we recompute the number of consumptions and productions for each array access. Since the range of time slots where these consumptions can slide in each loop is smaller, the ASAP and ALAP cases for each loop will be closer, therefore the gap between the bounds will be narrower, and the error margin smaller. By recomputing the upper and lower bounds, a better estimation of the memory size is obtained.

By splitting repeatedly the outer loops, the accuracy can be improved until the exact value is determined. Depending on the availability of time, and the level of accuracy required, the user can select a higher number of loop splittings.

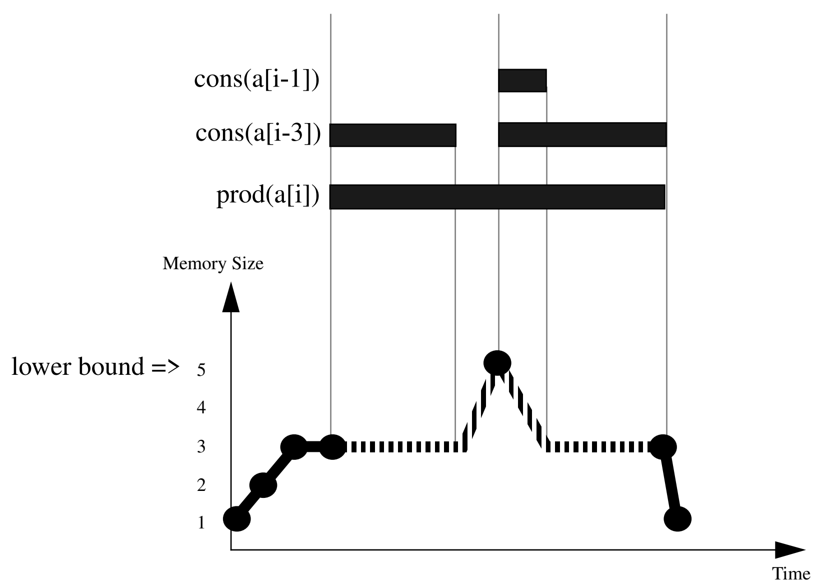


Figure 11: Memory trace after splitting, for early consumptions.

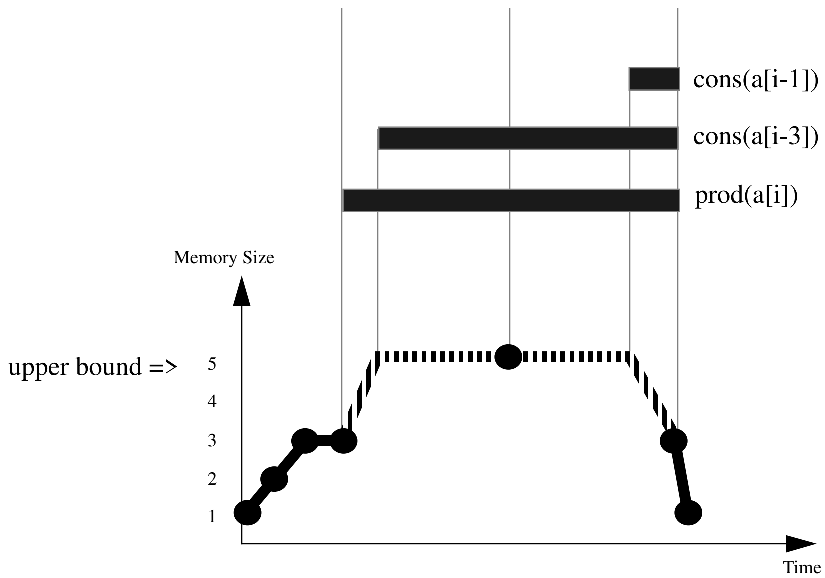


Figure 12: Memory trace after splitting, for late consumptions.

Figures 11 and 12 show the ASAP and ALAP memory traces after the loop in the code from the explanatory example was split once. The resulting loops have 3 to 51 and 52 to 100 as bounds. Since the exact trace is known at the loop boundaries, an extra point in the trace is known here (at the boundary between the two newly created loops). In our example the new upper and lower bounds are both equal to 5. Therefore the exact value of memory needed for the entire code is equal to 5.

## 3 Experimental Results.

We implemented our algorithm using C++ in a UNIX environment. We ran our experiments on a Sun SPARCStation5. The examples are taken from both multimedia applications (such as image, voice, video processing), and scientific applications. The figures are compared to a brute force (BF) implementation, which represents a symbolic execution of the code, and provides the exact value of the memory size, but requires a very high computation time. In some cases it is not even feasible. The brute force approach is used only for the purpose of comparison.

Table 1 shows the exact memory size, as computed by the brute force approach (column 3), as well as the upper and lower bounds generated by our algorithm (columns 5 and 6). Column 4 shows the amount of time needed for the BF algorihtm, whereas column 8 shows the computation time for our tool. Column 7 presents the error margin for each run of our algorithm, as 100*(upper-lower)/upper. The succesive rows for a specific application represent consecutive levels of accuracy. The first row corresponds to the first level of accuracy (where no loop splitting has been done). The second row represents the results after the loops have been split once, and so on. The #declared array elements (column 2) represent the total size of the arrays declared in the code. Since the input is considered to be in SSA form, the number of declared variables is very large, but the overhead introduced by SSA is eliminated by in-place mapping.

Our approach provides good results, with very close upper and lower bounds, for a reasonable computation time. For many of the examples we ran (Compress, Linear, Laplace, SOR), the upper and lower bounds are equal, which means the estimation is exact from the first pass (first level of accuracy).

In the case that the estimation is not exact from the first pass (e.g., for the Wavelett examples), the accuracy can be increased progressively, trading-off more computation time against closer bounds. The computation time for the applications we ran ranges between less than 0.01 sec, and 64 sec. For each input algorithm, the user can choose the computation time and accuracy level which best suits his application, and design process.

In some examples (such as IDCT) the computation time does not vary uniformly with the level of accuracy. Sometimes, for a higher level of accuracy it decreases (instead of increasing), due to the way the splitting was done. The complexity of the memory access pattern may decrease when the outermost loops are split, and the time needed to compute the intersections between lattices decreases, even though the total number of lattices increases.

In some examples (e.g., IDCT, Wavelett1) the brute force has a very small computation time (under 0.01s), whereas our algorithm has a more significant delay (tenths of a second, or even a second). This happens when the application has a complicated loops structure, but very small loop sizes (small number of iterations). If this is the case, the brute force algorithm has a low number of instructions to execute, whereas our approach has the overhead of lattice computation. Nevertheless, in the examples where the loop sizes are significant (e.g., Wavelett2), the overhead in our algorithm is still in the range of seconds, whereas the brute force algorithm is unfeasible.

In all the applications our algorithm generated reasonable memory size estimations in a computation time under 1 second. More exact estimations were obtained even for examples with very large size arrays, and loops (e.g., Wavelett2) within 5 seconds.

| Application | #declared array elements | Brute Force | | Our Algorithm | | | |
|---|---|---|---|---|---|---|---|
| | | Exact memory size | Time (s) | Lower bound | Upper bound | % Error (100*(u-l)/u) | Time (s) |
| ex1 | 10 | 3 | <0.01s | 3 | 3 | 0.0 | <0.01s |
| Compress | 20000 | 10000 | 112.90s | 10000 | 10000 | 0.0 | 0.02s |
| Linear | 402 | 202 | 0.05s | 20002 | 20002 | 0.0 | <0.01s |

**Table 1: Experimental results.**

| Application | #declared array elements | Brute Force | | Our Algorithm | | | |
|---|---|---|---|---|---|---|---|
| | | Exact memory size | Time (s) | Lower bound | Upper bound | % Error (100*(u-l)/u) | Time (s) |
| IDCT | 256 | 136 | 0.58s | 128 | 256 | 50.0 | 0.08s |
| | | 136 | 0.52s | 128 | 192 | 33.3 | 0.30s |
| | | 136 | 0.55s | 128 | 160 | 20.0 | 1.07s |
| | | 136 | 0.57s | 128 | 144 | 11.1 | 0.67s |
| | | 136 | 0.57s | 132 | 140 | 5.7 | 1.07s |
| | | 136 | 0.55s | 134 | 138 | 2.9 | 2.88s |
| | | 136 | 0.55s | 136 | 136 | 0.0 | 2.55s |
| Laplace | 20201 | 10404 | 73.70s | 10404 | 10404 | 0.0 | 1.15s |
| ex6 | 352 | 101 | 0.08s | 100 | 126 | 20.6 | 0.02s |
| | | 101 | 0.05s | 101 | 101 | 0.0 | <0.01s |
| ex7 | 10 | 4 | <0.01s | 3 | 4 | 25.0 | <0.01s |
| | | 4 | <0.01s | 4 | 4 | 0.0 | 0.02s |
| ex8 | 101 | 5 | <0.01s | 3 | 5 | 40.0 | <0.01s |
| | | 5 | <0.01s | 5 | 5 | 0.0 | <0.01s |
| SOR | 900 | 393 | 0.05s | 393 | 393 | 0.0 | 0.02s |
| Wavelett1 | 96 | 51 | <0.01s | 50 | 74 | 32.4 | 0.02s |
| | | 51 | <0.01s | 50 | 62 | 19.4 | 0.05s |
| | | 51 | 0.02s | 50 | 56 | 10.7 | 0.12s |
| | | 51 | 0.02s | 50 | 53 | 5.7 | 0.42s |
| | | 51 | <0.01s | 51 | 52 | 1.9 | 0.77s |
| | | 51 | <0.01s | 51 | 51 | 0.0 | 0.12s |
| Wavelett2 | 960,000 | infeasible (more than one day) | infeasible (more than one day) | 480002 | 760002 | 33.3 | <0.01s |
| | | | | 480002 | 600002 | 20.0 | 0.03 |
| | | | | 480002 | 540002 | 11.1 | 0.12 |
| | | | | 480002 | 510002 | 5.9 | 0.38 |
| | | | | 480002 | 495002 | 3.0 | 1.32 |
| | | | | 480002 | 487502 | 1.5 | 5.10 |
| Lowpass | 30404 | 10100 | 319.79s | 10099 | 10297 | 1.9 | 0.20s |
| | | 10100 | 320.47s | 10099 | 10199 | 1.0 | 0.73s |
| | | 10100 | 319.87s | 10099 | 10149 | 0.5 | 1.88s |
| GSR | 19011 | 13870 | 161.28s | 13870 | 13917 | 0.3 | 4.25s |
| IDCT2 | 256 | 136 | 0.92s | 128 | 768 | 83.3 | 0.08s |
| | | 136 | 0.90s | 128 | 448 | 71.4 | 0.30s |
| | | 136 | 0.93s | 128 | 288 | 55.6 | 1.12s |
| | | 136 | 0.90s | 128 | 208 | 38.5 | 0.72s |
| | | 136 | 0.93s | 132 | 172 | 23.3 | 1.20s |
| | | 136 | 0.90s | 134 | 154 | 13.0 | 3.45s |
| | | 136 | 0.95s | 136 | 144 | 5.6 | 4.15s |
| | | 136 | 0.93s | 136 | 140 | 2.9 | 12.47s |
| | | 136 | 0.95s | 136 | 138 | 1.4 | 22.72s |
| | | 136 | 0.98s | 136 | 136 | 0.0 | 23.47s |
| ex20 | 35 | 15 | 0.02s | 15 | 15 | 0.0 | <0.01s |
| ex21 | 35 | 20 | <0.01s | 20 | 20 | 0.0 | <0.01s |
| ex22 | 35 | 20 | <0.01s | 20 | 20 | 0.0 | <0.01s |
| ex23 | 35 | 20 | 0.02s | 20 | 20 | 0.0 | 0.02s |

**Table 1: Experimental results.**

| Application | #declared array elements | Brute Force | | Our Algorithm | | | |
|---|---|---|---|---|---|---|---|
| | | Exact memory size | Time (s) | Lower bound | Upper bound | % Error (100*(u-l)/u) | Time (s) |
| from paper | 20200 | 200 | 210.44s | 100 | 200 | 50.0 | 0.02s |
| | | 200 | 208.72s | 150 | 200 | 25.0 | <0.01s |
| | | 200 | 211.92s | 175 | 200 | 12.5 | 0.03s |
| | | 200 | 206.98s | 188 | 200 | 6.0 | 0.02s |
| | | 200 | 209.31s | 194 | 200 | 3.0 | 0.03s |
| | | 200 | 200.98s | 197 | 200 | 1.5 | 0.03s |
| | | 200 | 207.66s | 200 | 200 | 0.0 | 0.03s |
| Motion Detection (G=H=256, M=N=4, m=n=2) | 234 | 59 | 0.07 | 46 | 271 | 83.0 | 0.10s |
| | | 59 | 0.07 | 46 | 171 | 73.1 | 0.20s |
| | | 59 | 0.07 | 46 | 121 | 62.0 | 0.60s |
| | | 59 | 0.07 | 49 | 96 | 49.0 | 0.80s |
| | | 59 | 0.07 | 59 | 59 | 0.0 | 0.70s |

**Table 1: Experimental results.**

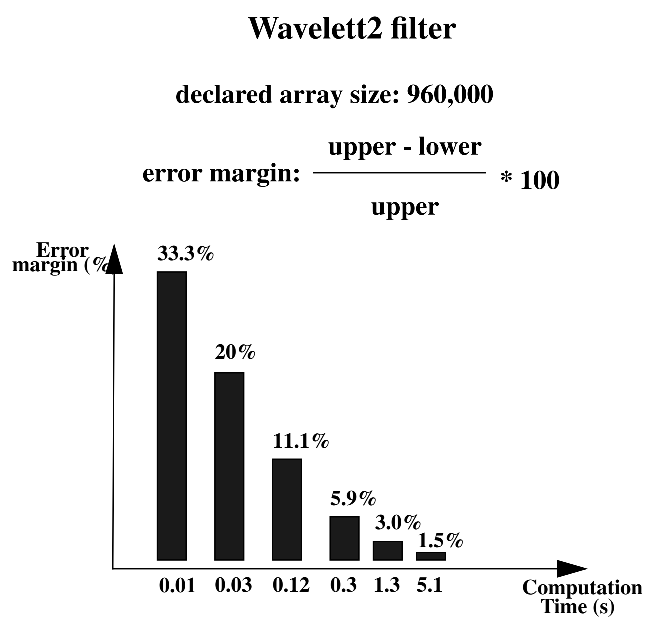


**Figure 13: Accuracy against time for the Wavelett filter.**

Figure 13 shows in detail the tradeoff between computation time and accuracy for an example with large size arrays. Our algorithm is run on a scaled up version of the Wavelett filter (Wavelett2 in the table), where there are 960,000 declared array elements. The brute force algorithm is unfeasible (it didn't finish in one day, on a SPARCStation 5). With our tool, for a very limited computation time (0.1 seconds), an estimation within 11% error margin can be computed. If the computation time constraint is relaxed to 5.1 seconds, an estimate within 1.5 % error margin is computed. We believe that this error margin is sufficiently accurate for an early estimation regarding the memory cost.

## 4. Conclusions.

We presented a new memory size estimation method, for a parallel procedural bahavioral input containing very large size arrays and loops. Our technique is based on polyhedral dataflow analysis, and allows the computation of an upper and a lower bound for the storage requirements. Depending on the amount of time available, the estimation accuracy can be increased, by narowing the gap between the bounds computed. This permits trading off computation time against the accuracy of estimation. The experiments have shown that for a running time in the range of 0.1 seconds, reasonable estimations can be generated. Even for very large examples, the error margin can be decreased to 1% within seconds.

Our memory size estimation provides early characterization of algorithmic descriptions, allowing good design decisions at the source level.

## 5. References.

[BaGa95] S. Bakshi, D. Gajski, “A Memory selection algorithm for high-performance pipelines,” *Proceedings of European Design Automation Conference (EURO-DAC’95)*, pp. 124-129, Brighton, UK, Sept. 1995.

[BMBL88] M. Balakrishnan, A.K. Majumdar, D.K. Banerji, J.G. Linders, J.C. Majithia, “Allocation of multiport memories in data path synthesis," *IEEE Trans. on Comp.-Aided Design*, Vol.CAD-7, No.4, pp. 536-540, April 1988.

[BaCM95] 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.

[Bane93] U. Banerjee, “*Loop Transformations for Restructuring Compilers: the Foundations,*” Kluwer Academic Publishers, Boston, 1993.

[BENP93] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, “Automatic Program Parallelization,” *Proc. of IEEE*, Vol. 81, No. 2, Feb. 1993.

[GDWL92], D. Gajski, N. Dutt, A. Wu, S. Lin, “*High-Level Synthesis: Introduction to Chip and System Design,*” Kluwer Academic Publishers, 1992.

[JhDu95] P. Jha, N. Dutt, “Library mapping for memories,” *Proc. of European Design and Test Conference (ED&TC’95)*, pp. 288-292, Paris, France, March 1997.

[KuPa87] F. Kurdahi, A. Parker, “REAL: A program for register allocation,” *Proc. 24th ACM/IEEE Design Automation Conf.*, pp. 210-215, June 1987.

[JLVW95] G. de Jong, B. Lin, C. Verdonck, S. Wuytack, F. Catthoor, “Background memory management for dynamic data structure intensive processing systems,” *Proc. IEEE Int. Conf. Comp.-Aided Design,* pp. 515-520, San Jose CA, Nov. 1995.

[PaDu97] P. Panda, N. Dutt, “Behavioral array mapping into multiport memories targeting low power,” *Proc. 10th Intn’l Conf. on VLSI Design*, pp. 268-272, Hyderabad, India, Jan. 1997.

[LMVW93] 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.

[PTVF92] 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.

[RaGC94] L. Ramachandran, D. Gajski, V. Chaiyakul, “An algorithm for array variable cluster-

ing," *Proc. European Design and Test Conf.*, pp. 262-266, Paris, France, March 1994.
[ScTh97] 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.
[SWCJ97] P. Slock, S. Wuytack, F. Catthoor, G. de Jong, "Fast and extensive system-level memory exploration for ATM applications," to appear in *Proc. of 10th Intn'l Symp. on System-Level Synthesis,* Antwerp, Belgium, Sept. 1997.
[StJe92] L. Stok, J. Jess, "Foreground memory management in data path synthesis," *Int. Journal on Circuit Theory and Appl.,* Vol. 20, pp. 235-255, 1992.
[SFCM93] 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.
[TsSi86] C.J. Tseng, D. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. on Comp.-Aided Design,* Vol. CAD-5, No. 3, pp. 379-395, July 1986.
[VeSR94] 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.
[Wolf96] M. Wolfe, *"High Performance Compilers for Parallel Computing,"* Addison-Wesley, Redwood City, CA, 1996.
[ZMSM94] V. Zivojnovic, J. Martinez, C. Schlager, H. Meyr, "DSPStone: A DSP-oriented benchmarking methodology," *Proc. of ICSPAT'94*, Dallas TX, Oct. 1994.