An Integrated Algorithm for Memory Allocation and Assignment in High-level Synthesis

Jaewon Seo
Dept. of EECS, KAIST, Korea

Taewhan Kim
Dept. of EECS, KAIST, Korea

Preeti R. Panda
Synopsys Inc., USA

Abstract – With the increasing design complexity and performance requirement, data arrays in behavioral specification are usually mapped to fast on-chip memories in behavioral synthesis. This paper describes a new algorithm that overcomes two limitations of the previous works on the problem of memory-allocation and array-mapping to memories. Specifically, its key features are *(1) a tight link to the scheduling effect*, which was totally or partially ignored by the existing memory synthesis systems, and supporting *(2) non-uniform access speeds* among the ports of memories, which greatly diversify the possible (practical) memory configurations. Experimental data on a set of benchmark filter designs are provided to show the effectiveness of the proposed exploration strategy in finding globally best memory configurations.

Categories and Subject Descriptors : B.5.2 [Register-Transfer-Level Implementation]: Design Aids

General Terms : Algorithms, Performance, Design

Keywords : memory allocation, memory assignemt, memory design, scheduling effect

1. INTRODUCTION

Behavioral synthesis, which is the process of automatically generating an RTL design from an algorithmic description, is an important research area in design automation. Many behavioral descriptions for manipulating large amounts of data computations use *array variables* to represent data storages. Consequently, behavioral synthesis is required to allocate memory modules for implementing the array variables. However, many synthesis systems either do not support this process or simply rely on designer's specification of particular memory modules. Thus, for memory intensive applications, it is necessary to automate the process of finding the best memory configuration in terms of both total memory cost and design performance.

There are number of research works which have addressed the problem of memory exploration in behavioral synthesis. Shiue *et. al* proposed an ILP model and heuristic-based algorithm for solving the problem of determining memory configuration with minimum area under power constraint or minimum power consumption under area constraint [1]. However, the approach does not take into account the scheduling effect, which could lead to a major limitation in the design space exploration. An exploration strategy under the assumption of a fixed hierarchical model (with different access speeds) of memories was proposed by Holmes and Gajski [2]. However, since the memory granularity is determined by the memory hierarchy used, its application to general (diverse) types of memories in library is not supported. Ramachandran *et. al* proposed a bottom-up cluster-based memory synthesis algorithm that produces an array variable configuration [3]. They assumed that for each array group, any memory module can be allocated only if it satisfies the array size requirement. This clearly limits the exploration of memories with different (read/write) access speeds and different number of ports. Panda proposed a memory bank exploration algorithm which exploits the memory access sequences to minimize page misses in a bank [4]. One limitation is that the exploration has excluded the memories with multiple ports. Wuytack *et. al* addressed the problem of storage bandwidth optimization, by which subsequent memory allocation and assignment tasks produce an architecture with less memories and memory ports [6]. Panda *et. al* provides a full survey on memory and array optimization in high-level synthesis [7].

In this paper, we propose a novel memory exploration strategy. Specifically, our key contributions are *a tight link to the scheduling effect*, which was hardly taken into account by the existing approaches, and supporting *non-uniform access speed* among the ports of memories, which greatly diversifies the possible memory configurations. We integrate the two features in one framework of our algorithm to explore memory configurations more fully and effectively.

2. MOTIVATING EXAMPLES

In this section, we motivate the key features of our algorithm with examples. Our algorithm is based on the following observations.

- *Scheduling effects* during memory exploration can significantly enhance the quality of the memory configuration. (Example 1)
- *Different speed of memory (read/write) accesses* during memory exploration significantly extends the scope of the exploration of the memory configurations. (Example 2)

Example 1: (scheduling effects)
Figure 1(a) shows a segment of behavioral description that manipulates four arrays where each size is 16 (bits) $\times$ 1024 (words). Suppose we have a library of memory modules available to use, as summarized in Figure 1(b). It specifies the number of (read-only, write-only, read/write) ports and the area[1] (i.e., cost) of each memory. In the library, we temporarily assumed that every memory access operation takes one clock cycle and all the operations in Figure 1(a) should be completed within three clock cycles. Figure 2(a) shows a schedule for the DFG, called *schedule-1*, and its possible allocations and assignments of memory modules using the library in Figure 1(b). We can easily find that arrays *A*, *B* and *D* cannot be grouped to allocate one memory module because three (read) access operations must be executed in parallel at clock step 1 and none of the memories support the parallelism. The highlighted configuration in the table of Figure 2(a) indicates the one with minimum total memory cost. However, when we reschedule the read operation $read\_D[i]$ from clock step 1 to 2, as shown in Figure 2(b), we can further reduce the total memory module cost, that is, from 26.55 to 22.46. This simple example indicates that scheduling is one of the major factor which drastically affects the memory exploration and the quality of the resulting memory configuration.

[1]In our work, the area (also energy consumption) of each memory was calculated according to the model proposed in [1].

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

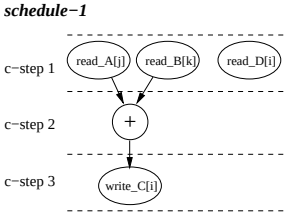
```
reg [15:0] A[1023:0];
reg [15:0] B[1023:0];
reg [15:0] C[1023:0];
reg [15:0] D[1023:0];
...
C[i] = A[j] + B[k];
t = D[i];
...
```

(a)

mem.	size (bits×words)	num. of ports r	w	rw	area (mm^2)
M1	16×1024	0	0	1	7.94
M2	16×1024	1	1	0	8.66
M3	16×2048	0	0	1	11.23
M4	16×2048	1	1	0	12.25
M5	16×2048	1	0	1	15.32

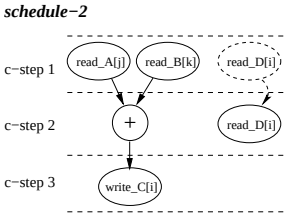
(b)

Figure 1: (a) A segment of input behavioral description, (b) A memory module library.



mem. configuration (array group : memory)	area (mm^2)
A:M1, B:M1, C:M1, D:M1	31.76
A:M1, B:M1, CD:M3	27.11
A:M1, C:M1, BD:M5	31.20
...	...
AB:M5, CD:M3	**26.55**
...	...

(a) *schedule-1* for Figure 1(a) and its possible mem. configurations



mem. configuration (array group : memory)	area (mm^2)
A:M1, B:M1, C:M1, D:M1	31.76
A:M1, B:M1, CD:M3	27.11
A:M1, C:M1, BD:M5	31.20
C:M1, D:M1, AB:M3	27.11
...	...
AC:M3, BD:M3	**22.46**
AB:M5, CD:M3	26.55
...	...

(b) *schedule-2* for Figure 1(a) and its possible mem. configurations

Figure 2: An example of showing the effect of scheduling on the exploration of memory configuration.

Example 2: (non-uniform speeds of memory accesses)
Table 1 shows a list of memory modules with detailed description of the number of clock cycles taken by each type of memory access. For example, memory module $M2$ has one read-only and one write-only ports, each consuming 2-clock cycle time, whereas $M2'$ has one read-only port having 2-clock cycle time and one write-only port having 3-clock cycle time. Clearly, integrating the non-uniform access times of memory modules and scheduling in one framework of memory exploration is essential to find globally best memory configurations.

3. THE MEMORY EXPLORATION ALGORITHM

The proposed algorithm accepts an unscheduled *data-flow graph* (DFG) as input. Let L be the memory library, and T be the number of clock cycles (i.e., latency constraint) allowed to execute all the operations in DFG. Then, the optimization problem is to select memories from L and assign them to arrays in DFG so that the total memory cost[2] is minimized while satisfying the latency constraint.

Our algorithm consists of three major steps : In Step 1, an initial solution is generated. In Step 2, the previous solution is refined iteratively. In Step 3, array clustering is performed.

Array grouping is a partition on the set of arrays, where arrays that reside in a same memory module are grouped together. Initially in Step1 each array is mapped to a separate group, which means that each array resides in a distinct memory. During Step 2 the grouping of arrays is not altered. After completion of Step 2, in Step 3 the grouping is modified by merging any two groups, and then with the resultant grouping, Step 2 is performed again. The whole process is repeated until a single large cluster is formed and the memory configuration and corresponding schedule with the minimum cost is chosen as a result. This decoupling of memory allocation and array grouping is based on the fact that the considering them together is computationally very expensive and it also enables the manual specification of the grouping by the designer, if needed.

[2]The cost usually refers to the total area of the allocated memories. However, our approach can easily change the cost to the 'total energy' consumed by the memories. (We provide a set of data on the energy consumptions in Table 3 of section 4.)

mem.	size (bits×words)	number of ports r	w	r/w	access cycles r	w	r/w	area (mm^2)
M1	16×1024	0	0	1	-	-	2	7.94
M1′	16×1024	0	0	1	-	-	3	7.94
M2	16×1024	1	1	0	2	2	-	8.66
M2′	16×1024	1	1	0	2	3	-	8.66
M3	16×2048	0	0	1	-	-	2	11.23
M3′	16×2048	0	0	1	-	-	3	11.23
M4	16×2048	1	1	0	1	2	-	12.25
M4′	16×2048	1	1	0	2	2	-	12.25
M5	16×2048	1	0	1	2	-	2	15.32
M5′	16×2048	1	0	1	3	-	3	15.32

Table 1: An example of memory library with access timings.

• Step 1 (*Generation of an initial solution*): We generate an initial schedule and a corresponding memory configuration that will be refined iteratively during Step 2. We use Force-Directed Scheduling (FDS) algorithm [8] to balance memory operations across clock steps for minimizing the requirements for memory read/write ports. FDS is one of the most representative scheduling algorithms whose objective is to minimize the hardware resources under a certain latency constraint. In our case, we use FDS to schedule the memory operations so that the maximum number of concurrent reads/writes is minimized. From the schedule obtained by FDS, we determine, for each array, the maximum concurrent accesses. Then, among the memories in L that supports this concurrent accesses, we select the one with the minimum cost, and assign it to the array. During FDS, access time of each memory operation is specified by the designer or simply assumed to be the minimum in the library to avoid the requirement for excessive number of ports

• Step 2 (*A simultaneous rescheduling and reassignment*): We refine the schedule and memory configuration obtained in Step 1. Our objective is to reduce the total cost of memory configuration by attempting to iteratively perform rescheduling (operations) and reassignment (arrays to memories) together. To support a large number of iterations, the task of rescheduling and reassignment performed at each iteration should be fast.

In scheduling, we place our primary importance on the optimization of memories to be used. Consequently, we extract all the memory access operations from DFG, and construct a graph, called the Memory Dependency Graph (MDG), which maintains a partial order between the memory access nodes in the DFG.

MDG has two dummy nodes called *source* and *sink*. An arc is created from *source* to every memory access node with no predecessor, and from every memory access node with no successor to *sink*. We assume *source* and *sink* are scheduled at clock step 0, T+1, respectively. The MDG concept is similar to that proposed in [4], but ours is more general in usage, including a minimum delay information between nodes.

Definition (*MI: minimum interval*): An arc $v \rightarrow w$ with *minimum interval* $MI(v \rightarrow w)$ in the MDG indicates the minimum number of clock cycles required for the execution of arithmetic/logical operations along the arc between the two memory access nodes v and w.

For example, $v \stackrel{MI=3}{\longrightarrow} w$ indicates that at least three clock cycles are required for the period between the completion of the execution of v and the start of w. We calculate the MI values for each arc during the process of generating an MDG by using a depth-first graph traversal procedure. For a schedule S, a memory configuration M for

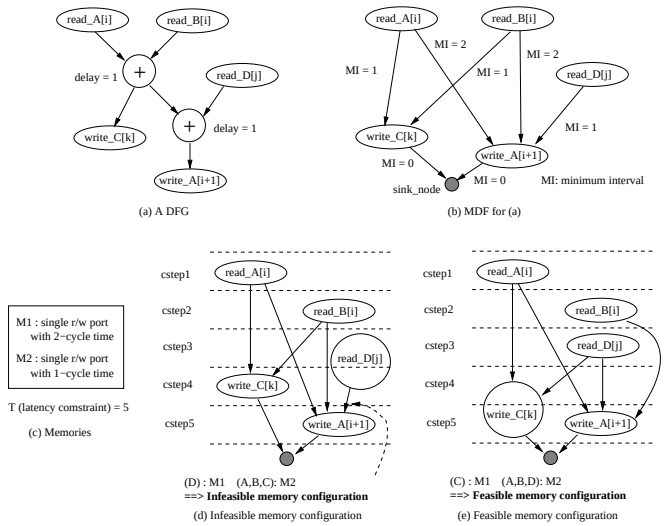


Figure 3: An example showing MDG construction and infeasible/feasible memory configurations.

S is called a *feasible* memory configuration if for every arc $v \rightarrow w$ in MDG, it satisfies

$$cstep(v) + delay(v) + MI(v \rightarrow w) \leq cstep(w) \quad (1)$$

where $cstep(v)$ is the clock step at which memory access operation v starts to execute and $delay(v)$ is the execution delay (in the number of clock cycles) of v. Consequently, *our goal is to find a schedule with latency constraint of T that leads to a feasible memory configuration with minimum total memory cost.*

An example of DFG and its MDG are shown in Figures 3(a) and (b), respectively. The number assigned to each arc in Figure 3(b) indicates the MI value. Figure 3(d) shows the memory operations for a schedule of the MDG when arrays *A*, *B* and *C* each uses a memory of type *M2* in Figure 3(c) and array *D* uses *M1*. Consequently, as indicated with dotted arrow in Figure 3(d), the arc from $read\_D[j]$ to $write\_A[i+1]$ violates the inequality relation in Eq.(1), resulting in an infeasible memory configuration for the schedule. On the other hand, Figure 3(e) shows a feasible memory configuration for the same schedule (i.e., all arcs satisfying the inequality in Eq.(1)), where *A*, *B* and *D* each uses a memory of type *M2* while *C* uses a type *M1* memory.

The example in Figure 4 shows how operation rescheduling and memory remapping are performed at an iteration of our algorithm. Suppose we have an initial schedule and its memory configuration shown in Figure 4(b) using the memories in Figure 4(a). For each operation, we determine *schedule_zone*, which is the set of clock steps at which the operation can be scheduled for execution without causing any timing violation of Eq.(1). The heavy lines next to the operations represent *schedule_zone*. For example, *schedule_zone* for $op1$ in Figure 4(c) is $\{cstep1\}$ since scheduling $op1$ to *cstep2* causes MI violations between $op1$ and $op4$ and between $op1$ and $op5$, while *schedule_zone* for $op4$ is $\{cstep3, cstep4\}$.

Then, we check for each memory in L, if the selected array can be scheduled within its *schedule_zone* without causing any MI violation. As a result, we obtain a new schedule for that array. We use a simple list scheduling heuristic for the checking using the selection guideline that the most timing-critical operation is scheduled first. Then, among the memories that lead to feasible memory configurations, we choose the one with the smallest memory cost. For example, the symbol attached to each interval of *schedule_zone* in Figure 4(c) indicates the selected memory.

Let $M(a_i)$ be the memory assigned to array a_i for a certain memory configuration R, and M_j be the memory selected according to

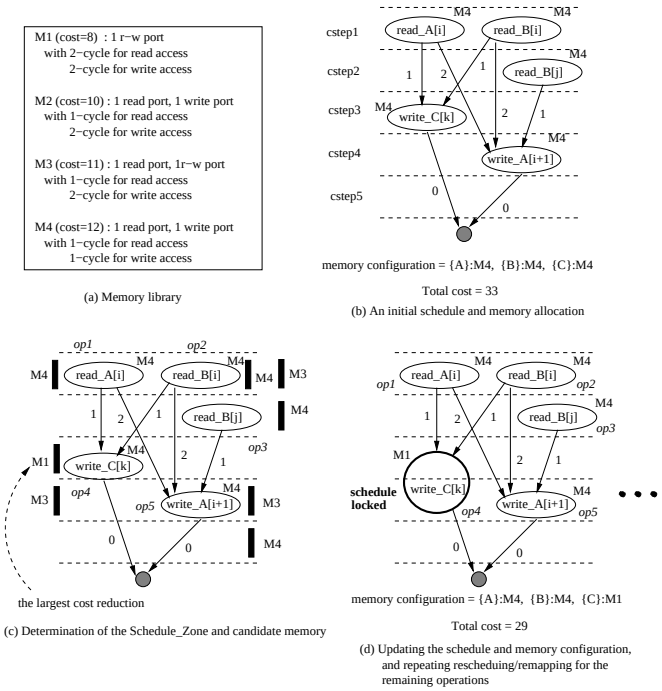


Figure 4: An example illustrating our incremental step of simultaneous rescheduling and remapping.

the procedure described above when all operations of a_i are rescheduled. Then, among all array-memory pairs (a_i, M_j), we choose to reschedule the one that maximizes the quantity

$$\Delta C = C(M(a_i)) - C(M_j) \quad (2)$$

where $C(M)$ represents the memory cost of type M. From the example in Figure 4(c), we selected op_4 and rescheduled (actually the same clock step as before) it at clock step 3 because its cost reduction in Eq.(2) (i.e.,*C(M4)-C(M1)*=12-8=4) is the largest. op_4 is then locked for rescheduling (but may change the assigned memory by subsequent iterations), and the process of rescheduling and reassignment for the remaining operations is repeated until all operations are rescheduled and reassigned.

The overall procedure of Step 1 and Step 2, called MEM-exp, works as follows. First, we generate an initial schedule for an input DFG and a memory configuration for the schedule using a memory library L. We refine the initial solution iteratively in the loop of our algorithm. At each iteration, we select a pair of array and memory that lead to a maximum decrease of memory cost (i.e., the value in Eq.(2)) without violating the latency constraint of T and causing an infeasible memory configuration. We then reschedule all the operations of that array and at the same time, reassign the memory determined using the procedure described in the previous paragraph to the selected array. Once the rescheduling and remapping are done, any attempt to reschedule the operation is disallowed for the rest of the execution of the inner loop of our algorithm. The loop continues if we find a new configuration whose total cost is less than the minimal cost found so far or there is an overall reduction of the memory cost in the current inner loop (i.e., $\sum \Delta C < 0$).

• Step 3 (*Array Clustering*): The exploration of memory configurations in Step 2 has been performed with the restriction that the grouping of arrays is fixed. In Step 3, the grouping is changed to further reduce the total memory cost, by bottom-up array clustering. Arrays are clustered by merging two different groups into a single group and reallocating a new memory module to that group. Storing two or more arrays into the same memory module may require additional operations for address translation. After adding proper operations

design	$\|L\|$	MEM-exp		*Exhaustive*		diff.
		cost (mm^2)	time (sec)	cost (mm^2)	time (sec)	(%)
kalman	180	10.772	0.3	10.718	77	0.5
sor	200	371.134	0.1	371.134	20	0.0
phoneme	180	132.050	0.7	-	-	-
heat	180	17.103	0.2	17.103	27	0.0
ddpoly	200	8.349	0.1	8.349	32	0.0
tridag	200	24.816	0.1	23.309	24	6.1
avg.						1.32

Table 2: Comparisons of memory area for the memory configurations produced by MEM-exp **and an exhaustive searching method.**

into DFG, rescheduling is performed in the same way as in Step 2. Among all possible merging of two groups the one that reduces the memory cost maximally is chosen. After completion of Step 3, current memory configuration is refined iteratively without changing the grouping of arrays (Step 2). The whole process terminates when only a single large group remains.

Extension to DRAM Memories : When L includes DRAM memories, MEM-exp is extended to exploit one important feature of DRAM architecture. The architecture enables a significant speed-up in executing subsequent memory accesses that refer to the words in the same page because additional steps of row decoding and precharging are not needed. Such memory access mode is called *page mode* [4].

We extend MEM-exp to exploit the page mode accesses: If two memory access operations in MDG are executed successively and they use the same DRAM memory, we check the MI violation between the two operations and *schedule_zone* violation of the operations by determining the number of clock steps required for executing them based on the page mode accesses, and comparing the number with the MI value and *schedule_zone*.

It should be noted that our memory exploration method can easily incorporate other types of DRAM access modes (e.g., those described in [5]) with similar ways.

4. EXPERIMENTAL RESULTS

The proposed memory exploration algorithm MEM-exp was implemented in C++ and executed on a Sun Sparc20 workstation. We tested a set of memory intensive applications to demonstrate the effectiveness of the proposed memory exploration algorithm. kalman, sor and heat are the kalman filter design, the successive over relaxation design, and the differential heat release computation design respectively [9]. phoneme is the phoneme recognition [10]. trida is the tridiagonal systems of equations and ddpoly is the polynomial and its derivative evaluation [11].

The experiments were performed in two respects: (1) to check the effectiveness of memory exploration for minimizing the total memory area and (2) to check the effectiveness for minimizing the total energy consumption by memories.

The latency constraint in each experiment is set to the median value in the range between the lower and upper bound of latency constraints. With latency constraint which is lower than lower bound, no feasible solution can be obtained, while, with latency constraint higher than upper bound, no more improvements can be achieved.

• **Effectiveness of minimizing total memory area**: Table 2 shows a comparison of results produced by our algorithm and an exhaustive searching method (*Exhaustive*), in which we used a numberof pruning techniques to speed up the exhaustive exploration without losing optimality. $|L|$ is the number of memories in memory library L. In terms of total memory area, our solutions are on average 1.3% more than the optimal solutions. In terms of run time, our method is significantly fast. *Exhaustive* slows down considerably as the number of arrays become large, and for design phoneme which contains 28 arrays, it was not able to produce a solution within 6 hours.

design	$\|L\|$	MEM-exp	*Greedy*	red.
		cost (nJ)	cost (nJ)	(%)
kalman	180	6.95	8.09	14.0
sor	200	1158.89	3827.38	69.7
heat	180	14.29	20.91	31.7
phoneme	180	362.66	1293.56	72.0
ddpoly	200	16.75	18.64	10.1
tridag	200	25.38	28.38	10.6
avg.				34.6

Table 3: Comparisons of energy consumption for the memory configurations produced by MEM-exp **and a greedy heuristic.**

• **Effectiveness of minimizing total energy consumption**: We compare our results, in terms of energy consumption, with those produced by *Exhaustive* and a greedy heuristic (called *Greedy*). The results are summarized in Table 3. The idea in *Greedy* was taken from [1], in which it tries to assign arrays that are more frequently accessed to memories with less energy consumption. Since the heuristic in [1] does not take into account the latency constraint, we modified the heuristic so that it allocates memories in a greedy manner while satisfying the latency constraint. The comparisons in Table 3 indicate that MEM-exp allocates memories more selectively according to the access patterns and frequencies in the behavioral specification, with 34.6% less energy consumption than that produced by *Greedy*.

5. CONCLUSIONS

In this paper, we proposed a new (SRAM/DRAM) memory exploration algorithm to solve two important problems: *(1) a tight link to the scheduling effect*, which was totally or partially ignored by the existing memory synthesis systems, and supporting *(2) non-uniform access speeds* among the ports of memories, which greatly diversify the possible practical memory configurations. We solved the problems in *an integrated fashion* to explore memory configurations more fully and effectively by devising efficient cost evaluation and timing-violation checking techniques. Experimental results showed that the run time of our algorithm is significantly faster than that of the exhaustive search based method while the quality, in terms of memory area, of the memory configurations was within 1.3% difference (on average) with the optimal solutions. In term of energy consumption, our algorithm was able to find schedules and memory configurations with 34.6% less energy consumption (on average) compared to the existing method [1].

Acknowledgment : This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center(AITrc).

6. REFERENCES

[1] W.-T. Shiue, *et. al*, "Low Power Multi-Module, Multi-Port Memory Design for Embedded Systems", *Proc. of Signal Processing Systems*, 2000.

[2] N. Holmes and D. Gajski, "Architectural Exploration for Datapaths with Memory Hierarchy", *Proc. of EDAC*, 1994.

[3] L. Ramachandran, *et. al*, "An Algorithm for Array Variable Clustering", *Proc. of the EDAC*, 1994.

[4] P. R. Panda, "Memory Bank Customization and Assignment in Behavioral Synthesis", *Proc. of ICCAD*, 1999.

[5] P. R. Panda, *et. al*, "Incorporating DRAM Access Modes into High-level Synthesis", *IEEE Trans. on CAD*, 1998.

[6] S. Wuytack, *et. al*, "Minimizing the required memory bandwidth in VLSI system realizations.", *IEEE Trans. on VLSI Systems*, 1999.

[7] P. R. Panda, *et. al*, "Data and Memory Optimization Techniques for Embedded Systems", *ACM Trans. on Design Automation of Electronic Systems*, 2001.

[8] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Trans. on CAD*, 1989.

[9] N. D. Dutt, "High-Level Synthesis Design Repositories", *http://www.ics.uci.edu/ dutt*.

[10] Schmit, H., Thomas, D.E. "Synthesis of application-specific memory designs", *IEEE Trans. on VLSI Systems*,, 1997.

[11] W. H. Press, *et. al*, *Numerical Recipes in C*, Cambridge University Press, 1988.