Design Automation for Embedded Systems, 9, 235–261, 2004.
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

Multi-Module Multi-Port Memory Design for Low Power Embedded Systems

WEN-TSONG SHIUE ra9164@freescale.com
Division of Multimedia Application Processor, WMSG, Freescale Inc., Austin, TX 78735

CHAITALI CHAKRABARTI chaitali@asu.edu
Department of Electrical Engineering, Arizona State University, Tempe, AZ 85287-7206

Abstract. In this paper we describe a multi-module, multi-port memory design procedure that satisfies area and/or energy constraints for embedded applications. Our procedure consists of application of loop transformations and reordering of array accesses to reduce the memory bandwidth followed by memory allocation and assignment procedures based on ILP models and heuristic-based algorithms. The specific problems include determination of (a) the memory configuration with minimum area, given the energy bound, (b) the memory configuration with minimum energy, given the area bound, (c) array allocation such that the energy consumption is minimum for a given memory configuration (number of modules, size and number of ports per module). The results obtained by the heuristics match well with those obtained by the ILP methods.

Keywords: multi-module memory, multi-port memory, memory allocation and assignment, ILP models

1. Introduction

Minimization of energy is one of the most important performance metrics in the design of portable systems. Traditionally, the main focus has been to reduce the energy consumption in the datapath components. However, in systems that involve multidimensional streams of signals such as images or video sequences, the majority of the area and energy cost is not due to the data-path or controllers but due to the global communication and memory interactions [4]. This is due to the fact that energy consumed in memory transfers is significantly larger than that due to data-path operations. This implies that with proper design, reduction in the memory related energy budget can far exceed the reduction due to voltage scaling and other energy saving transformations.

In some applications, assuming that the on-chip memory is a single module, single port memory, is restrictive. In fact, some applications demand data parallel access capabilities, which are necessary to complete the computations within a cycle budget. Such capabilities can only be supported by multi-port, multi-module memories.

In this paper we describe a multi-module, multi-port memory design and exploration procedure for embedded systems. First, we apply loop transformations and reorder the array accesses to reduce the memory bandwidth. Next, we determine the memory configuration and the allocation of arrays to the different memory modules such that the energy and/or area constraints are satisfied. Given high-level code, area and energy models, and the system constraints of area and energy, our aim is to find the best memory configuration (and array

© 2005 Springer Science + Business Media Inc.

assignment). We develop ILP-based models and heuristics to address the following three problems:

- Determine the memory configuration (and array assignment) with minimum area, given the energy bound.
- Determine the memory configuration (and array assignment) with minimum energy, given the area bound.
- Determine the allocation of arrays onto a fixed memory configuration (number of modules, size and number of ports per module is given) such that the energy consumption is minimum.

The heuristic-based algorithms are very efficient since they look at a small subset of configurations. The results obtained by our heuristics match very well with those obtained by the ILP models. In addition, we exploit the fact that arrays with overlappings can share memory modules.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 describes the storage bandwidth optimization procedure followed by the area and energy models and the area-energy tradeoffs. Section 4 describes (i) how to find the best memory configuration and array assignment given area/energy bound and (ii) energy-efficient array assignment given the memory configuration. Section 5 demonstrates application of our methodology on a large benchmark, namely, the Binary Tree Predictive Coder. Section 6 concludes the paper.

2. Related Work

There has been extensive work done in the area of memory organization and optimization. For a comprehensive treatment, please refer to [14]. In this section we briefly describe the work in allocation of variables (scalar and array) and memory bandwidth reduction since they are closely related to the contents of this paper. We start with the work that has been done in the area of binding variables to registers, register files and memories. Register allocation of scalar variables is typically done by graph coloring the register compatibility graph. When the individual registers are replaced by a memory module, the number of allowed simultaneous accesses is limited to the number of available ports in memory. The method in [2] is a 0-1 LP formulation to allocate multiport memories by first assigning variables to registers and then merging registers to form a multiport memory module. Ahmad and Chen [1] extends the approach in [2] and determines the minimum number of multiport memory modules and also minimizes the interconnection.

In [11], memory allocation is done during high-level synthesis resulting in more efficient utilization of multiport memory. The scheduling algorithm minimizes the number of memory ports by evenly distributing data transfers over all the control steps. Most existing allocation methods group the variables (or registers) to form memory modules and then

© 2005 Springer Science + Business Media Inc.

determines the interconnection between the modules and the functional units. Clearly, the cost of interconnection depends on the variable grouping. The method in [9] assumes that interconnect cost is dominant and tries to minimize the cost of interconnection first and then group the variables to form memory modules.

A memory binding procedure for control flow intensive applications has been proposed in [8]. The algorithm starts with the initial assignment where each array is housed in a separate module. It then groups two arrays based on minimizing the cost of the grouping (in terms of performance) using a greedy algorithm. The method in [13] chooses a memory configuration by varying the number of modules and studying the tradeoffs between area and delay for different array assignments in each configuration. The approach in [3] is different from others in that it starts with the dynamic execution profile and finds the optimal multi-banked SRAM architecture to fit the profile. The automated search technique based on recursive partitioning uses a comprehensive cost function, which considers both the energy due to access, and the overhead associated with increasing the number of banks.

Issues regarding packing the array variables in memory have been addressed in [7, 15]. The procedure in [15] packs the array variables vertically (arrays in the same module) and horizontally (arrays with different bit ranges are packed to the same physical word) using simulated annealing. The procedure in [7] decomposes the problem into three subproblems: port mapping, bitwidth mapping and word mapping, and solves the subproblems using exhaustive search and even linear approximation algorithms.

More recently, there have been several research efforts at reducing the memory bandwidth. This problem is very important since it translates into reducing the area and power of the memory components. Most of the existing methods try to minimize the number of simultaneous data accesses without taking into account which data is being accessed. However, real life applications consist of data with distinctive access patterns and locality. By customizing the local memory architecture to match the diverse access patterns and locality, the main memory bandwidth can be reduced significantly. A double cache architecture consisting of a temporal cache and a spatial cache that uses dynamic prediction to route the data to one of the two caches has been proposed in [5]. A similar architecture that allocates the variable statically thereby avoiding the overhead of dynamic prediction has been proposed in [6].

The approach presented in this paper is most closely related to that in [4, 17]. There the first step is that of storage cycle budget distribution where the minimal memory bandwidth required to meet the real-time constraints is derived. This is followed by the memory allocation and assignment step where a memory architecture is designed based on the constraints of the first step. The method in [4] constructively generates a partial memory access ordering where the cost function takes into account size, access frequencies, etc. The result is an extended conflict graph (ECG): the nodes represent arrays, the edges represent compatibility between arrays, and the edges are annotated with the number of read/write ports needed by a memory which stores the arrays representing the nodes they connect. Since many possible orderings are possible for a given ECG, the memory allocation and assignment step systematically chooses the number and type of memories and assigns every basic group to one of the allocated memories based on a cost function that involves power consumption and chip area.

© 2005 Springer Science + Business Media Inc.

3. Background

3.1. Storage Bandwidth Optimization

The goal of Storage Bandwidth Optimization is to minimize the required memory bandwidth within the given cycle budget, by adding ordering constraints to the flow graph [4, 17]. The resulting memory architecture has a smaller number of memories and fewer memory.

3.1.1. Definitions

We explain the terms conflict graph, extended conflict graph, self-conflict, and hyper-edges and their implications with the help of the example in Figure 1. Here the arrow indicates the design flow.

Conflict Graph: A conflict graph consists of a set of nodes that correspond to the arrays. There is an edge between two nodes whenever the corresponding arrays are in conflict.

Extended Conflict Graph (ECG): This is a conflict graph that contains additional information regarding read, write and read/write ports, and also self-conflict and hyper edges (as will be described next).

Self-Conflict Edges: These edges are due to the same array being accessed more than once in the same cycle. In Figure 1(a), array E has a self-conflict in the initial ADOC in cycle 4. Since there are two simultaneous accesses to E, the memory should have at least 2 ports. It is important to reduce the number of self-conflict edges since fewer the number of self-conflict edges, fewer the number of modules or ports.

Hyper Edges: If multiple arrays are accessed in the same cycle, then the corresponding nodes are linked by a hyper-edge. In Figure 1(a), there is a hyper-edge between arrays A, B, C, and D in cycle 8 implying that if all four arrays are stored in the same memory module, it should have at least four ports (3 read ports and 1 write port).

If there are self-conflict and hyper-edges in an ECG, the conflicting arrays then have to be assigned to either different memories or to a multiport memory. In such cases, if only single port modules are available, then the number of modules is equal to the chromatic number of the ECG. If multiport modules are available, then the number of ports should be larger than the chromatic number of the ECG.

Figure 1 clearly illustrates the effect of SBO on the memory bandwidth. After application of SBO, the number of self-conflict edges is reduced from 1 to 0, the number of hyper-edges is reduced from 1 to 0, and the chromatic number is reduced from 4 to 2. Thus if we assign all arrays to the same memory module, the number of ports is reduced from four ports (1 write port, 2 read ports, and 1 read/write port) to two ports (2 read/write ports).

3.1.2. Proposed Technique

Loop transformation procedures have been shown to significantly reduce the number of accesses and the size of on-chip/off-chip memory in single port, single module memories. Loop transformations can also be used to design area/power-efficient multi-port, multi-module memories as is described below. The proposed method uses a combination of loop

© 2005 Springer Science + Business Media Inc.

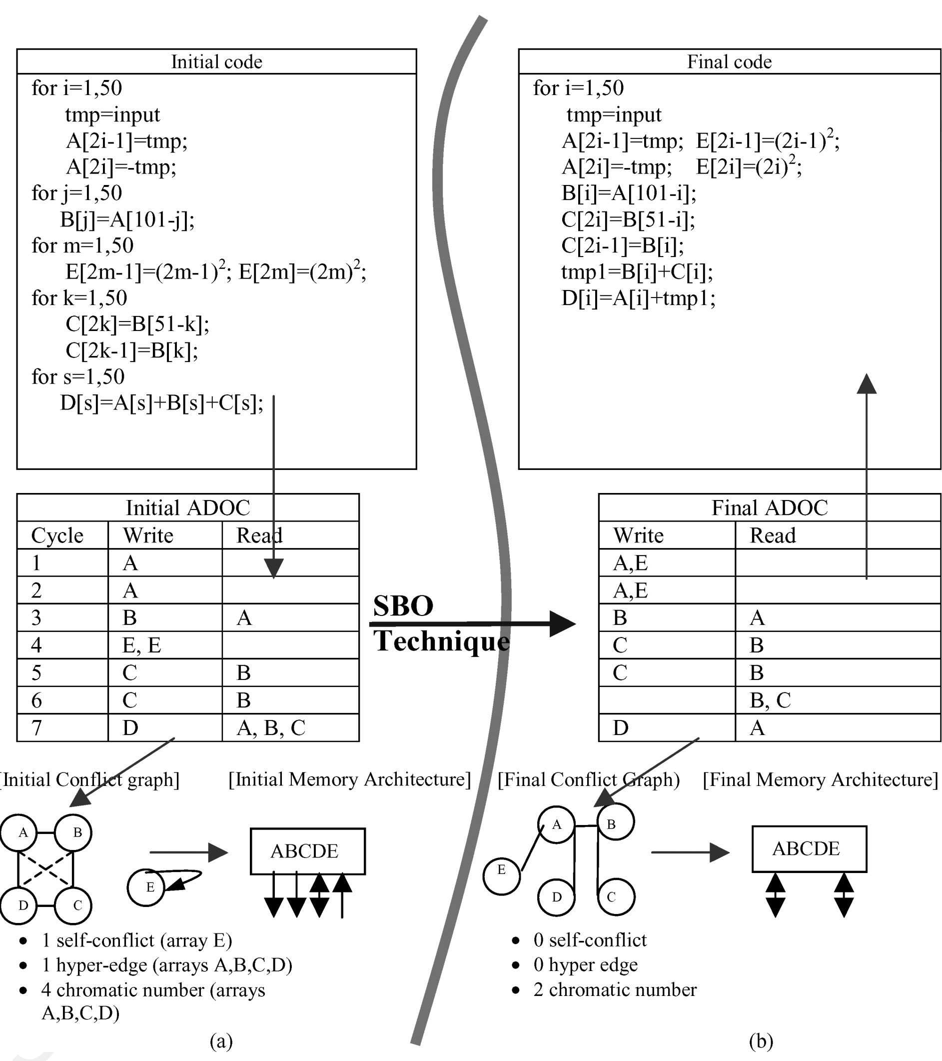


Figure 1. Example illustrating effect of application of the SBO technique, (a) Initial and (b) Final code, ADOC, Conflict Graph, and Memory Architecture.

transformations and reordering of the array accesses to reduce the number of self-conflict edges, hyper-edges and chromatic number.

[Proposed SBO Procedure]

1. *Loop Unrolling.*
   [*helps to reduce the cycle budget*]
2. *Loop Peeling/Loop Normalization/Loop Fusion.*
   [*facilitates array reordering*]

© 2005 Springer Science + Business Media Inc.

3. *Array Reordering Scheme & Scalar Replacement*
   [*helps to reduce the number of self-conflict edges, hyper-edges and the chromatic number*]

We explain the procedure with a set of examples.

Step 1: Loop unrolling can be used to reduce the cycle budget when the datapath consists of multiple computation units. In Example 1, if the initial cycle count is 100 and there are two computation units, then loop unrolling by a factor of 2 will drop the cycle count to 50. However, this results in introduction of self-conflicting edges, which can be removed by the application of additional loop transformations in Step 2.

[Example 1]

Before	After
For j=1,100 S[j]=j^2;	For j=1,50 S[2j-1]=$(2j-1)^2$; S[2j]=$(2j)^2$;
Conflict graph S	*Conflict graph* S (self-edge)

[Example 2]

Initial codes (Cycle count=50+50+75=175)	After loop transformation (Cycle count=50*4=200)	After loop transformation + array-reordering scheme (Cycle count=50*3=150)
for j=1,50 A[2j-1]=t1; A[2j]=t2; for k=1,75 B[2k-1]=$(2k-1)^2$; B[2k]=$(2k)^2$;	for j=1,50 A[2j-1]=t1; A[2j]=t2; B[2j-1]=$(2j-1)^2$; B[2j]=$(2j)^2$; B[j+100]=$(j+100)^2$;	for j=1,50 A[2j-1]=t1; B[2j-1]=$(2j-1)^2$; A[2j]=t2; B[2j]=$(2j)^2$; B[j+100]=$(j+100)^2$;
Conflict graph A B (self-edge on B)	*Conflict graph* A B (self-edge on B)	*Conflict graph* A — B

Step 2: Loop fusion (in conjunction with loop peeling and loop normalization) is used to facilitate the array reordering scheme in Step 3. Loop peeling and loop normalization are used to normalize the iteration space before invoking loop fusion. While loop fusion may increase the cycle count, it facilitates reordering of array accesses. For instance, in Example 2, after application of loop fusion, the cycle count increases from 175(=

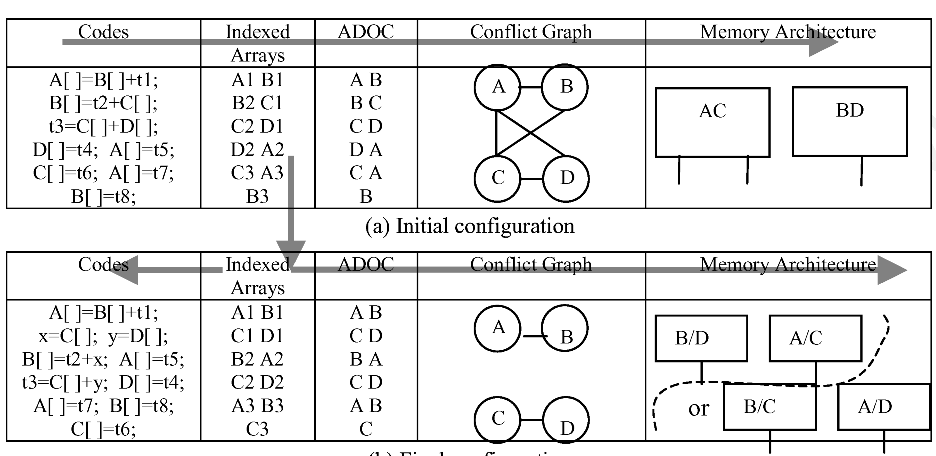


Figure 2. Example illustrating array reordering scheme for arrays executed in the same loop.

$50 + 50 + 75$) to 200 ($= 50 + 50 + 50 + 50$). However after array reordering, the cycle count drops to $150 (= 50 + 50 + 50)$.

Step 3: Array reordering scheme is used to reduce the number of self-conflict edges, the number of hyper-edges and the chromatic number. Scalar replacement is an integral part of this scheme. The proposed algorithm is greedy and consists of two main steps. The first step is to identify a set of groupings referred to as *patterns* that would cover all the arrays. The next step is to reorder the array accesses such that the arrays that are accessed in the same cycle match one of the identified patterns. We describe this with the help of the example in Figure 2.

In order to identify the patterns, we first count the number of accesses to each array. For instance, in the example in Figure 2(a), the access count for array A is 3, for array B is 3, for array C is 3 and for array D is 2. Since two arrays can be accessed in the same cycle (we assume that there are two computation units), each pattern consists of two arrays. The first pattern that we choose consists of arrays A and B since their access counts are large. Once the pattern $\{A,B\}$ has been identified, we choose the next pattern consisting of arrays C and D. Note that $\{A,C\}$ and $\{B,D\}$ or $\{B,C\}$ and $\{A,D\}$ also form valid pattern sets.

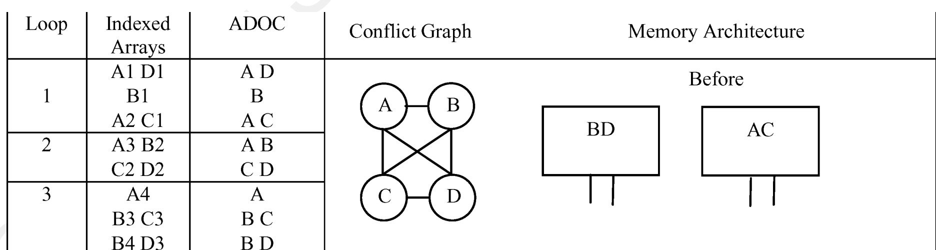
Once the patterns $\{A, B\}$ and $\{C, D\}$ are identified, the procedure to reorder the arrays accesses is invoked. The procedure is a one-pass greedy algorithm which reorders the accesses such that there is minimal increase in the number of cycles and the data dependencies are not violated. The basic idea is to distribute the arrays (that were assigned in the same cycle) over multiple cycles if they do not match an identified pattern, and then assign arrays to the empty slots such that the arrays in the same cycle match one of the patterns. For instance, is there is an empty slot next to array P, and Q, R are the candidate arrays, then we choose array Q if $\{P, Q\}$ is an identified pattern. Note that scalar replacement is used to distribute the array accesses over multiple cycles. In order to ensure that data dependencies

© 2005 Springer Science + Business Media Inc.

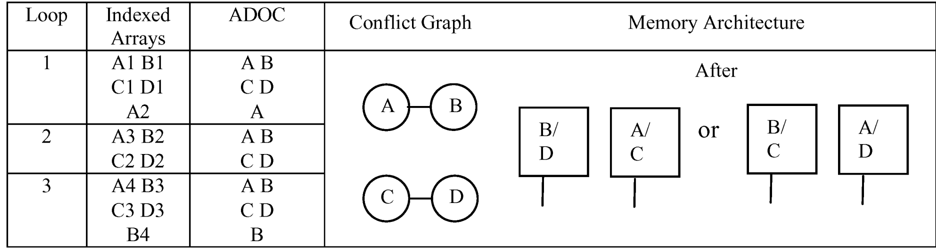
are not violated, we index the arrays in the order in which they are accessed. For instance, the first time array A is accessed, we refer to it by Al, the next time by $A2$ and so on, and while reordering, we make sure that $A2$ is not accessed before $A1$.

We next describe how the arrays are reordered in the example in Figure 2(a). Cycle 1 consists of accesses to arrays $A1$ and $B1$. Since $\{A, B\}$ is a pattern, we do not make any changes in the access order. Cycle 2 consists of accesses to arrays $B2$ and $C1$. Since $\{B, C\}$ is not a pattern, we assign $C1$ in cycle 2 and $B2$ in cycle 3. The next cycle consists of accesses to arrays $C2$ and $D1$. Since $\{C, D\}$ is a pattern, we can either assign $C2$ and $D1$ in cycle 4, or split the two arrays and assign $D1$ to cycle 2 and $C2$ to cycle 4. If we assign $C2$ and $D1$ in cycle 4, then while reordering the accesses in the next cycle, array $D2$ cannot be packed with $C1$ in cycle 2(because it may violate dependence constraints) resulting in larger number of cycles. So we choose to assign $C2$ in cycle 4 and $D1$ in cycle 2. The next set of accesses are to arrays $D2$ and $A2$. These are split and $A2$ is assigned to cycle 3 (since $\{A, B\}$ is a pattern) and $D2$ is assigned in cycle 4 (since $\{C, D\}$ is a pattern). The accesses in cycle 5, that is, accesses to $C3$ and $A3$ are split; $A3$ is assigned in cycle 5 and $C3$ is assigned in cycle 6. Finally $B3$ is assigned in cycle 5 since {A,B} is a pattern. The arrays are de-indexed and the final conflict graphs and source codes are generated. The simpler conflict graph translates to a better memory configuration (see Figure 2(b)). The notation A/C means that arrays A and C can share the same memory module.

Figure 3(a) gives an example where the array reordering scheme is applied across loops. In order to generate the patterns, we now count the number of accesses per array across all loops. For instance, the array access count for array A, B, C and D are 4, 4, 3 and 3 respectively and the corresponding patterns are $\{A, B\}$ and $\{C, D\}$. The array access



(a) Initial configuration



(b) Final configuration

Figure 3. Example illustrating array reordering scheme for arrays executed in different loops.

© 2005 Springer Science + Business Media Inc.

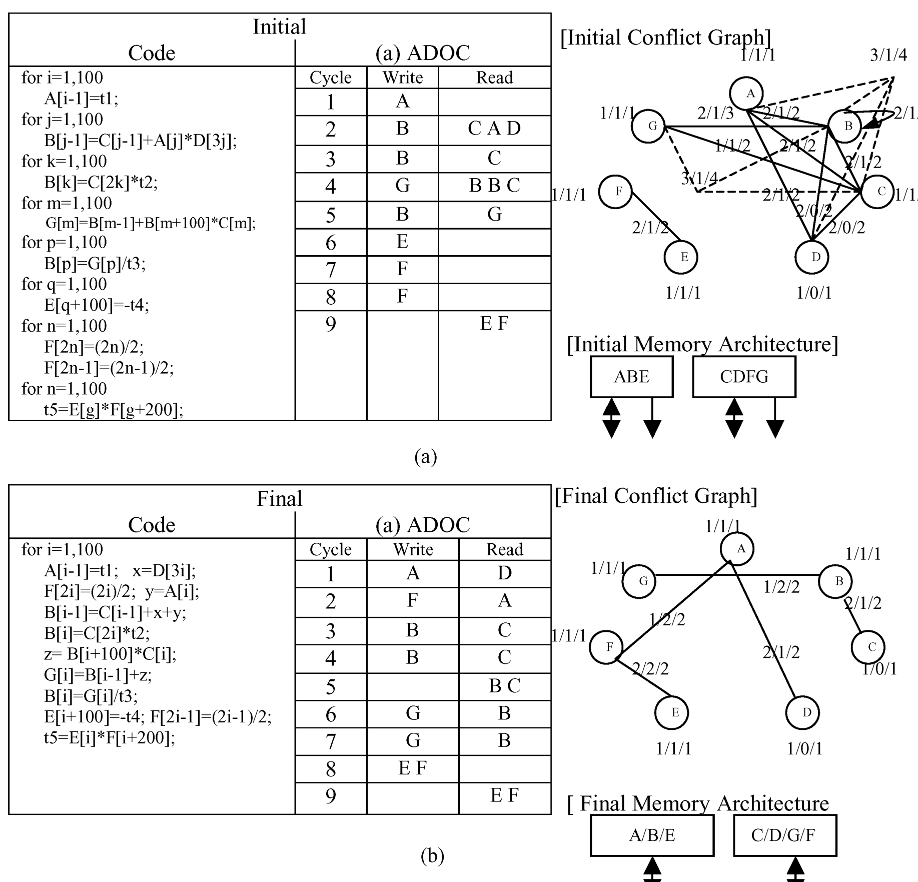


Figure 4. (a) Initial and (b) Final configuration for ADOC, conflict graph, and memory architecture.

ordering is done for each loop separately. Figure 3(b) shows the reordered accesses and the corresponding conflict graph.

In the rest of this paper, we use the following example to illustrate the memory assignment and allocation procedure. Figures 4 describe the initial and final (after application of SBO) configurations. In Figure 4(a), there are hyper-edges between nodes (A, B, C, D) and nodes (B, C, G), there is a self-conflict edge on node B, and the chromatic number is 4. After the application of loop fusion and array reordering scheme, the hyper-edges and self-conflict edges have been eliminated and the chromatic number reduced to 2 (see Figure 4(b)). The final memory architecture consists of two single port memory modules.

3.2. Area and Energy Models

In order to compare the different memory configurations, it is necessary to develop good area and energy models. The area model that we developed is based on Mulder's model

© 2005 Springer Science+Business Media Inc.

[12] and modified according to the data points in [17] for 1.2 μm CMOS technology. Given the bit width, word length and R/W/RW port-information, we can calculate the area for a multi-port module as follows.

$A = TF^{*}bits^{*}(1 + \alpha^{*} p1)^{*}sqrt(words)^{*}PF^{*}K$; [unit:mm^2]
where $PF = [1 + 0.25^{*}(p1 + p2 - 2)]$; $p1 = R + W$; $p2 = RW$;
A: *Area.*
TF: *technology scaling factor (min-geometry[μ]/2)^2.*
PF: *empirical factor for the number of ports in SRAM.*
words: *the number of words.*
bits: *the width of a word in bits.*
$p1$ *the total number of single-end ports.*
$p2$: *the total number of ports.*
α: *0.1*
K: *3.9174e-2*

We also developed an energy model that is a modified version of the model in [10] for multi-port on-chip memory architectures. Given the bit/word length of arrays, read/write accesses, the clock frequency, and the number of ports, we can calculate the energy consumption using this model.

$E = (Eread + Ewrite)^{*}ports^{*}1e6$; [$unit : uJ$]
where
*Cread=(9707+108*words+1126*bits+6*words*bits)*1e-15;*
*Cwrite = (7994 + 117*words + 759*bits + 9*words*bits)*1e-15;*
$Eread = 0.5^{*}Vdd^{2*}Cread^{*}Raccess$; *(Here Vdd = 5V).*
$Ewrite = 0.5^{*}Vdd^{2*}Cwrite^{*}Waccess$;
E: *energy.*
Eread: *energy due to read.*
Ewrite: *energy due to write.*
Raccess: *number of read accesses.*
Waccess: *number of write accesses.*
words: *the number of words.*
bits: *the width of a word in bits.*
ports: *the total number of ports.*

Both the area and the energy models do not take into account the effect of routing, control, etc. Clearly, availability of such information will make the model more accurate and improve the quality of the solution.

3.3. Area-Energy Tradeoffs

Since small memory modules consume less energy, assigning individual arrays to separate memory modules results in the smallest energy consumption. However this configuration

© 2005 Springer Science + Business Media Inc.

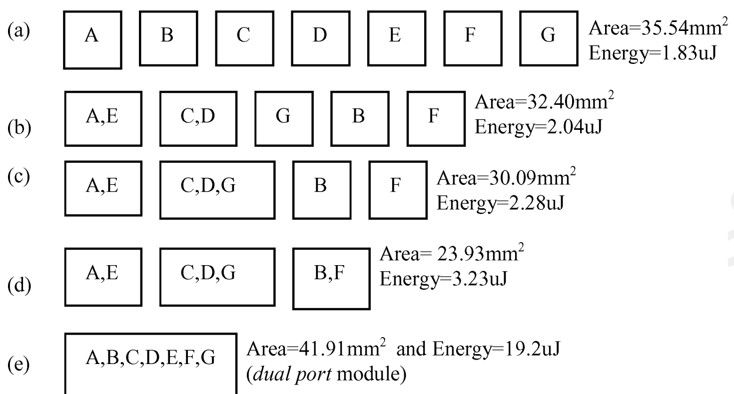


Figure 5. Area and energy values for different memory configurations for the example in Figure 4(b).

results in a large area. Figure 5(a) describes this configuration for the example in Figure 4(b). Assigning arrays with the same bit width to the same module is very area-efficient as shown in Figure 5(d). The energy consumption of this configuration is however larger than Figure 5(a), as is expected. Configurations that illustrate the area-energy tradeoffs are shown in Figure 5(b) and (c).

Grouping all the arrays in to a single memory module results in both large energy consumption and area as shown in Figure 5(e). This is because the resulting module has two ports and grouping arrays with different bit widths results in loss of memory.

4. Choosing Best Memory Configuration and Array Assignment

In this section, we describe the ILP-based models and heuristics to solve the following three problems:

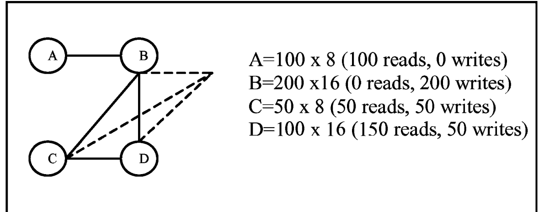
Problem 1: Given the energy bound, find the memory configuration with minimum area.
Problem 2: Given the area bound, find the memory configuration with minimum energy.
Problem 3: Given the memory configuration (number of modules, size and the number of ports per module), map the arrays to given single-/multi-port memory modules such that the energy consumption is minimum.

4.1. Problem 1: Find the Memory Configuration with Minimum Area, Given the Energy Bound

4.1.1. ILP Model

The ILP model is used to find the number of modules and the size of each module such that the area is minimum and the energy constraint is satisfied. This translates to finding

© 2005 Springer Science + Business Media Inc.



(a)

Variables	Allowed groupings	Area (mm^2)	Energy (uJ)
X1	A	1.69	0.04
X2	B	4.79	0.18
X3	C	1.2	0.03
X4	AC	2.07	0.11
X5	D	3.38	0.12
X6	AD	4.79	0.26

(b)

```
min:
      1.69x1+4.79x2+1.2x3+2.07x4+3.38x5
      +4.79x6;
N1: x1+x4+x6=1;
N2: x2=1;
N3: x3+x4=1;
N4: x5+x6=1;
P1: x1<=1;
P2: x2<=1;
P3: x3<=1;
P4: x4<=1;
P5: x5<=1;
P6: x6<=1;
E1:
0.04x1+0.18x2+0.03x3+0.11x4+0.12x5+0.26
x6<=0.45; /* energy bound=0.45uJ */
int x1,x2,x3,x4,x5,x6;
```

(c)

Figure 6. Example illustrating the ILP model. (a) conflict graph, (b) area and energy costs of the allowed groupings, (c) ILP formulation.

which arrays are housed in each module. We refer to the set of arrays that are assigned in each module as an array grouping. The first step in the procedure is to identify the permissible array groupings. The arrays in an array grouping are assigned to the same module. For instance, for the example in Figure 6, groupings corresponding to the edges and hyper edges, i.e., $\{AB\}$,$\{BC\}$,$\{CD\}$, $\{BD\}$,$\{BCD\}$, and those that include the *ECG*-based groupings, i.e., $\{ABC\}$, $\{ABD\}$,$\{ACD\}$,$\{ABCD\}$, are not allowed. The permissible array groupings are $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$, $\{AC\}$, and $\{AD\}$.

The next step is to derive the ILP model. Let x_i be a $\{0,1\}$ integer which assumes a value 1 if array grouping i is chosen. Let S_i be the area cost of array grouping i. Then the objective is to minimize $\Sigma_i S_i x_i$ (where $i = 1, N$), given the energy bound, where N is the number of permissible groupings. The constraint equations consist of (i) array constraint that ensures that an array can be assigned to only one of the modules, (ii) 0-1 constraint that ensures that x_i is either 0 or 1, and (iii) energy constraint that ensures that the energy of the assignment satisfies the energy bound and (iv) the integer constraint.

For $E_{\text{bound}} = 0.45$ uJ, the ILP solver finds three array groupings, $\times 2$, $\times 4$ and $\times 5$ to be equal to 1. Thus the final assignments has three memory modules with $\{\mathrm{D}\}$ in one module, $\{B\}$ in one module and $\{A\}$ and $\{C\}$ in another module. The area of this assignment is 10.24 mm^2 and the energy is 0.41 uJ, which is less than the energy bound. To incorporate into this ILP model, we need to only change the area and energy costs of the allowed groupings. If two arrays P: $(n_p \times b_p)$ and Q: $(n_q \times b_q)$ are in the same grouping, then the area cost is a function of $N = \max\{n_p, n_q\}$ and $B = \max\{b_p, b_q\}$, where n_i is the number of words in array i and b_i is the bitwidth in array i. The energy cost is calculated for a memory of size $N \times B$. Note that if is not considered, the area cost would be a function of $N' = n_p + n_q$ and $B = \max\{b_p, b_q\}$.

© 2005 Springer Science+Business Media Inc.

<ILP Model>

S: cost of area, T: cost of energy, Y_{bound}: energy bound, N: number of possibilities, N_a: number of arrays.

(i) Objective function:

$$\min : \sum_{i=1}^{N} S_i x_i;$$

(ii) Array constraint:

$$\forall 1 \leq i \leq N, x_i \leq 1;$$

(iii) 0-1 constraint:

$$\forall 1 \leq j \leq N_a, \sum_{i=1}^{N} x_i = 1;$$

(iv) Energy constraint

$$\sum_{i=1}^{N} T_i x_i \leq Y_{\text{bound}};$$

(v) Integer Constraint:

$$\forall 1 \leq i \leq N, \text{int} x_i;$$

4.1.2. Heuristic Algorithm

The proposed algorithm iteratively finds the memory configuration with minimum area given the energy bound by either grouping arrays or splitting already grouped arrays based on whether the energy bound is closer to the minimum area configuration or the minimum energy configuration. Let U correspond to the configuration with maximum energy (obtained by assigning arrays with same bit width to the same memory module) and minimum area, and L correspond to the configuration with minimum energy (obtained by assigning each array to a separate module) and maximum area. Let X be the desired configuration and $E(X)$ the given energy bound.

The algorithm in Figure 7 decides whether to group arrays or split already grouped arrays based on the value of $E(X)$ with respect to $E(L)$ and $E(U)$. If $E(X)$ is closer to $E(L)$ (than

© 2005 Springer Science + Business Media Inc.

```
Main (E(X),E(U),E(L),R1,R2)
E(U) – E(X) =d1; E(X) – E(L) =d2; D=d1+d2;
•  if (d1>=d2),   /*Fuse array*/
   -if (d2/D) >= ¼, {Fuse arrays with higher priority according to R=R1}.
            else {Fuse arrays with lower priority according to R=R2}.
   -Calculate the energy for the configuration Y, i.e. E(Y).
   -if E(Y) > E(X) (OVERSHOT), {Go back to the previous configuration)
      else {Update the energy value of L, (E(L)=E(Y) )}.
   -while (set R ≠{}) OR (E(Y) < E(X)) {Main (E(X),E(U),E(L),R1,R2)}
   -if (set R={}) or (E(Y) = E(X)), {Current configuration is the best configuration; Break;}
         else { - Configuration A obtained by splitting the 2nd most recently merged group in
                  the current configuration.
                  if E(Y)<=E(X),    Solution3=1.
                  - Previous configuration B. Solution1=1.
                  - Configuration C obtained by fusing arrays in B with the lowest priority in set
                  R.
                  if E(Y)<=E(X), Solution2=1}
•  else (i.e. d1 < d2 ) /*Split array*/
   -if (d1/D) >= ¼, {Split arrays with higher priority according to R=R2}.
            else {Split arrays with lower priority according to R=R1}.
   -Calculate the energy for the configuration Y, i.e. E(Y).
   -if E(Y) < E(X) (OVERSHOT), {Go back to the previous configuration}.
            else {Update the energy value of U, (E(U)=E(Y))}.
   -while (set R ≠{}) OR (E(Y) > E(X))
                  Main(E(X),E(U),E(L),R1,R2)}
   -if (E(Y) = E(X)), {Current configuration is the best configuration; Break}.
            else { - Current configuration A. Solution1=1.
                  - Configuration B obtained by fusing the 2nd most recently splitted group in A.
                    if E(Y)<=E(X), Solution2=1.
                  -    Configuration C obtained by splitting arrays in previous configuration
                       with the lowest priority in  set R.
                  if E(Y)<=E(X), Solution3=1}.
   -if (Solution2=1) AND (Solution3=1), {Compare the configurations with Solution2 and
   configuration with Solution3 and pick the one with larger energy value}.
   elseif (Solution2=1), {Configuration with Solution2 is the best configuration}
   elseif (Solution3=1), {Compare the configurations with Solution1 and configuration with
   Solution3 and pick the one with larger energy value }
   else { Configuration with Solution1 is the best configuration }
```

Figure 7. Problem 1. Heuristic-based algorithm.

$E(U)$), we group arrays. On the other hand, if $E(X)$ is closer to $E(U)$, we split already grouped arrays. To help in deciding which arrays should be grouped or split, the arrays in the candidate set R are prioritized. If $E(X)$ is very close to $E(L)$, then arrays with lower energy value (lower area value if their energy values are equal) are given higher priority for grouping. If $E(X)$ is close to $E(L)$ but not very close, then arrays with higher energy value (lower area value if their energy values are equal) are given higher priority for grouping. If $E(X)$ is very close to $E(U)$, then arrays with lower energy value (higher area value if their energy values are equal) are given higher priority for splitting. If $E(X)$ is close to $E(L)$ but not very close, then arrays with higher energy value (higher area value if their energy values are equal) are given higher priority for splitting.

After a successful grouping (energy $< E(X)$), candidate set R and configuration L are updated. Similarly after a successful split (energy $> E(X)$), set R and configuration U

© 2005 Springer Science + Business Media Inc.

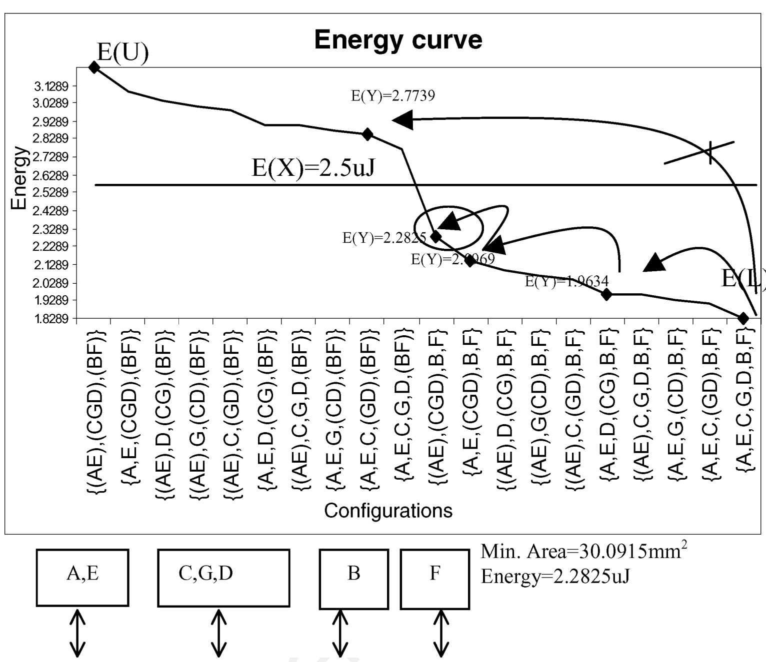


Figure 8. Min-area memory configuration for the given energy bound ($E(X) = 2.5$ uJ).

are updated. If the grouping is unsuccessful (energy > $E(X)$), we evaluate the following possible configurations and pick the one which is the closest to the energy bound: (i) configuration A obtained by splitting the second most recently merged group in the current configuration, (ii) previous configuration B, and (iii) configuration C obtained by fusing arrays in B with the lowest priority in set R. Similarly, if the splitting is unsuccessful (energy is a lot less than $E(X)$), we evaluate the following configurations and pick the one which is the closest to the energy bound: (i) current configuration A, (ii) configuration B obtained by fusing the second most recently split group in A, and (iii) configuration C obtained by splitting arrays in previous configuration with the lowest priority in set R. The procedure is iterative and continues until the set R is empty. The proposed algorithm for finding the minimum area memory configuration, given energy bound, is shown in Figure 8.

The inputs are $E(X)$, $E(U)$, $E(L)$, and the priority sets $R1$ and $R2$. While the arrays in $R1$ and $R2$ are both ordered primarily with respect to energy, smaller arrays are given higher priority in $R1$ and larger arrays are given higher priority in $R2$.

We illustrate this algorithm with the example in Figure 4 for an energy bound, $E(X) =$ 2.5 uJ. U corresponds to the following grouping $\{(A, E), (C, D, G), (B, F)\}$ and has an energy of $E(U) = 3.2275$ uJ. L corresponds to the grouping $\{A, B, C, D, E, F, G\}$ and has an energy of $E(L) = 1.8289$ uJ. The value of d1 is $E(U) - E(X) = 0.7275$ and the value of $d2$ is $E(X) - E(L) = 0.6711$. Since $d1 >= d2$ and $d2/(d1 + d2) >= 1/4$, we first fuse arrays B, F (with high priority) in set R. The energy for this configuration $\{(BF), C, E, G, A, D\}$ is 2.7739 uJ, which is larger than $E(X)$.

© 2005 Springer Science+Business Media Inc.

So we go back to the previous solution and remove array B. Now $R = \{F, C, E, G, A, D\}$. Array F can not get fused with any arrays in set R because of bitwidth mismatch and it is removed from set R. Next, we fuse arrays C and G since they are of the same bit width. The energy for the configuration $\{B, F, (CG), E, A, D\}$ is 1.9634 uJ, which is smaller than $E(X)$. We update the value of $E(L)$ to 1.9634 uJ and replace the arrays C and G in set R with a new array CG. The priority lists are also updated. Now $R = \{CG, E, A, D\}$. The process is repeated. The final configuration is $\{B, F, (CGD), (EA)\}$ with an energy of 2.2825 uJ $<$ 2.5 uJ. Figure 7 describes the algorithm trace for this example.

4.1.3. *Results*

We simulate the example in Figure 4 for different energy bounds. The results are given in Figure 9. Note that the results obtained by our heuristic match exactly with those obtained by the ILP method. For large examples (see Section 4), there are differences in the results obtained by the two methods.

Figure 10 shows the reduction in area in the memory configuration if arrays are taken into account. The average reduction is ∼32.71% which is significant.

In [16], we showed the use of loop transformations to systematically reduce array conflicts so that the resulting memory architecture contains fewer memory modules and fewer memory ports. Figure 11 demonstrates how area is reduced as a result of loop transformations and considerations. For instance, for energy bound of 2.5 uJ, the area if loop transformations and are considered, is 16.51 mm^2 compared to 32.4 mm^2, otherwise. The reduction in memory size is on the average 34.4% for the examples considered in Figure 11.

Energy Bound (uJ)	Area (mm^2)		% error
	Heuristic algorithm	ILP	
2.8	29.375	29.375	0
2.5	30.091	30.091	0
2.1	32.400	32.400	0

Figure 9. Problem 1. Minimum area memory configuration, given different energy bounds for the examples in Figure 4(b).

Energy Bound(uJ)	*Without*		*With*		% red.
	Config	A (mm^2)	Config.	A (mm^2)	
2.5	(AE),(CGD), (B),(F)	30.09	(B/D/F), (A/C/E/G)	16.5	45.1
2.1	(AE),(C),(GD), (B),(F)	32.40	(C/D/G), (B/F),(A/E)	17.5	46.0
1.9	(A),(E),(C),(G), (D),(B),(F)	35.54	(B/D),(E) (A/C/G),(F)	27.0	23.2

Figure 10. Problem 1. Results illustrating the % reduction in memory size if is taken into consideration.

© 2005 Springer Science + Business Media Inc.

Energy bound (uJ)	Before loop transformation		After loop transformation		% red.
	Without	*With*	*Without*	*With*	
2.5	32.40	20.89	30.09	16.51	49.0
2.1	32.40	20.89	32.40	17.50	46.0
1.9	35.54	30.46	35.54	27.08	23.8

Figure 11. Problem 1. Results illustrating the % reduction in memory size if loop transformation as well as array are taken into consideration.

4.2. Problem 2: Find the Memory Configuration with Minimum Energy, Given the Area Bound

The formulation of this problem (both ILP and heuristic) is very similar to that of Problem 1 and we will not discuss it here. For the example in Figure 4(b), we quote the results obtained by the heuristic and the ILP formulations for different area bounds in Figure 12. The results obtained by the heuristic match very well with those obtained by the ILP method.

Figure 13 shows the reduction in energy in the memory configuration if is taken into account. The average reduction is ∼24.31% which is significant. Figure 14 demonstrates how energy is reduced as a result of loop transformations and considerations. For instance, for area bound of 31.5 mm^2, the energy if loop transformations and considered, is 1.8288 uJ compared to 2.6388 uJ, otherwise. The reduction in energy consumption due to the reduction of memory size is on the average 25.6% for the examples considered in Figure 14.

Area Bound (mm^2)	Energy (uJ)		% error
	Heuristic algorithm	ILP	
32.5	2.04	2.04	0
30.5	2.28	2.28	0
27.5	2.85	2.85	0

Figure 12. Problem 2. Minimum energy memory configuration, given different area bounds for the example in Figure 4(b).

Area Bound (mm^2)	*Without*		*With*		% Red.
	Config.	*E(uJ)*	*Config.*	*E (uJ)*	
32.5	(AE),(C), (GD),(B),(F)	2.04	(C/D/G),(B), (F),(A),(E)	1.82	10.6
30.5	(AE),(CGD) (B),(F)	2.28	(C/D/G),(B), (F),(A),(E)	1.82	19.9
27.5	(A),(E),(C), (GD),(BF)	2.85	(C/D/G),(B), (F),(A/E)	1.87	34.5

Figure 13. Problem 2. Results illustrating the % reduction in energy consumption if is taken into consideration.

© 2005 Springer Science + Business Media Inc.

Area bound (mm^2)	Before loop transformation		After loop transformation		% red.
	Without	*With*	*Without*	*With*	
32.5	2.04	1.82	2.04	1.82	10.6
30.5	2.77	1.87	2.28	1.82	34.1
27.5	2.85	2.04	2.85	1.87	34.5

Figure 14. Problem 2. Results illustrating the % reduction in memory size if loop transformation as well as array are taken into consideration.

4.3. Problem 3: Map the Arrays to Given Single-/Multi-Port Memory Modules Such That the Energy Consumption is Minimum, Given the Memory Configuration (Number of Modules, Size and the Number of Ports per Module)

The proposed ILP and heuristic based algorithms try to assign frequently accessed arrays to modules that are small in size and/or have few ports. This is because the energy consumption is a function of the size of the memory, number of ports and the number of accesses. Specifically,

$$\begin{aligned} Energy &= (Eread + Ewrite)^{*}ports,\\ &=(Cread * Raccess + Cwrite * Waccess) * ports\\ &= ports * Cread * (Raccess + Ratio * Waccess)\\ &= Cr * y\\ &\text{where, } Cr = ports^{*}Cread,\ y = Raccess{+}Ratio^{*}Waccess,\\ &Ratio = Cwrite/Cread \cong 1.2. \end{aligned}$$

Thus the parameter Cr is a function of the size and the number of ports, and the parameter y is a function of the number of accesses. Note that while the ratio Cwrite/Cread does not vary with memory size, Cread is a function of the memory size (number of words and bit width).

4.3.1. ILP Model

We develop a simple ILP model based on the ball-hole assignment problem. Let x_{ij} be an $\{0,1\}$ integer value, which assumes a value of 1 if array i is assigned to module j and is 0 otherwise. Our objective is to (i) minimize the energy consumption which is a function of Cr and y. Our constraint equations consist of (ii) array constraint that ensures an array (ball) can be assigned to only one of the modules (holes), (iii) edge constraint that ensures that the ECG edge constraints and the number of ports match, and (iv) size constraint that ensures that the sum of the sizes of the assigned arrays is less than or equal to the module size, and (v) the integer constraint. The ILP model can be formulated as follows.

<ILP Model>

C: cost of energy.

© 2005 Springer Science + Business Media Inc.

N_a: number of arrays; N_m: number of modules.
V: set of arrays; E: set of edges; H: set of hyper edges.
Np: number of ports.
W: word size of array; Z: word size of module.

(i) Objective function:

$$\min : \sum_{i=1}^{N_a} \sum_{j=1}^{N_m} (y_i * Cr_j) x_{ij};$$

where y_i is a function of the number of accesses of array i and Cr_j is a function of the size and the number of ports of module j.

(ii) Array constraint:

$$\forall 1 \leq i \leq N_a, \sum_{j=1}^{N_m} x_{ij} = 1;$$

(iii) Edge constraint:

$$ECG = (V; E; H)$$
$$\forall (v_s, v_t) \in E, \forall 1 \leq j \leq N_m, x_{sj} + x_{tj} \leq Np_j;$$
$$\forall (\vee_1, \vee_2, \ldots \vee_m) \in H, \forall 1 \leq j \leq N_m, \sum_{p=1}^{m} x_{\vee_p j} \leq Np_j;$$

(iv) Size constraint:

$$\forall 1 \leq j \leq N_m, \sum_{i=1}^{N_a} W_i x_{ij} \leq Z_j;$$

(v) Integer constraint:

$$\forall 1 \leq i \leq N_a, \forall 1 \leq j \leq N_m, x_{ij} \in \{0, 1\};$$

4.3.2. *Heuristic Algorithm*

The heuristic algorithm operates on *R*, a set of arrays that are prioritized based on the value of y, and *S*, a set of modules that are prioritized based on the value of *Cr*. The algorithm is greedy and tries to assign an array with a high value of y to a module with a low value of Cr. An assignment is valid only if (i) there is enough space in the module to house that array and (ii) the limitations imposed by the conflict graph matches the number of available ports. For instance, if array A has conflict with an array that has already been housed in the high priority module, M, and if M has a single port, then A cannot be assigned to M. If, on the other hand, M has two ports, then A can be assigned to M. Finally, if the bitwidth of the array, ba, is larger than that of the module, bm, then multiple memory words (i.e.

© 2005 Springer Science + Business Media Inc.

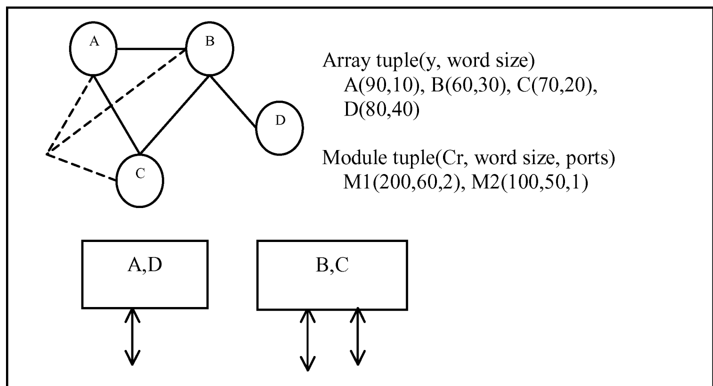


Figure 15. Example illustrating the algorithm of array assignment.

$\lceil (ba/bm) \rceil$ words) have to be dedicated for each array entry. If ba $<$ bm, then the memory is under-utilized. A solution is not feasible if the number of arrays connected by the largest hyper edge is larger than the total number of ports.

We explain the algorithm with the help of the example in Figure 15. Arrays A, B, C, D have to be housed in two memory modules $M1$(word size 60, 2ports) and $M2$(word size 50, 1port). Since three arrays (A, B, and C) are connected by the hyper edge and there are 3 ports (2 in $M1$, 1 in $M2$), a memory assignment is possible with the given memory configuration. The arrays in set R are prioritized based on the value of y. Thus priority of $A >$ priority of D and $R = \{A, D, C, B\}$.

The modules in set S are prioritized based on the value of Cr and $S = \{M2, M1\}$. First array A is assigned to $M2$. The next higher priority array, array D, can be assigned to $M2$ since there is no edge in ECG between A and D, and the space of $M2$ is enough to house array D. The next higher priority array is array C. Array C cannot be assigned to $M2$ because $M2$ does not have enough space and so it is assigned to $M1$ instead. Finally, array B is assigned to $M1$. Even though there is an edge between B and C, there is no conflict, since $M1$ is a dual port memory. The final assignment is also shown in Figure 15.

Algorithm: Array Assignment (given memory configuration)

(i) $R =$ *the set of arrays to be assigned. List the priority of arrays in set R based on the value of y. Higher value of y has higher priority, where y is related to the number of memory accesses.*

(ii) $S =$ *the set of modules. List the priority of modules in set S based on the value of Cr. Smaller value of Cr has higher priority, where Cr is related to the size and the number of ports.*

© 2005 Springer Science + Business Media Inc.

Given modules	Size (mm^2)	Ports	Final assignment	Min. Energy (uJ)
M1 M2	16 17	1 1	{A,E,B} {C,G,D,F}	5.48
M1 M2	17 18	1 1	{B,F,D} {A,E,C,G}	6.03
M1 M2 M3 M4	12 12 14 14	1 1 1 1	{B,A} {F} (C,E,G) {D}	3.60

Figure 16. Results of array assignment for different memory configurations for the example in Figure 4(b).

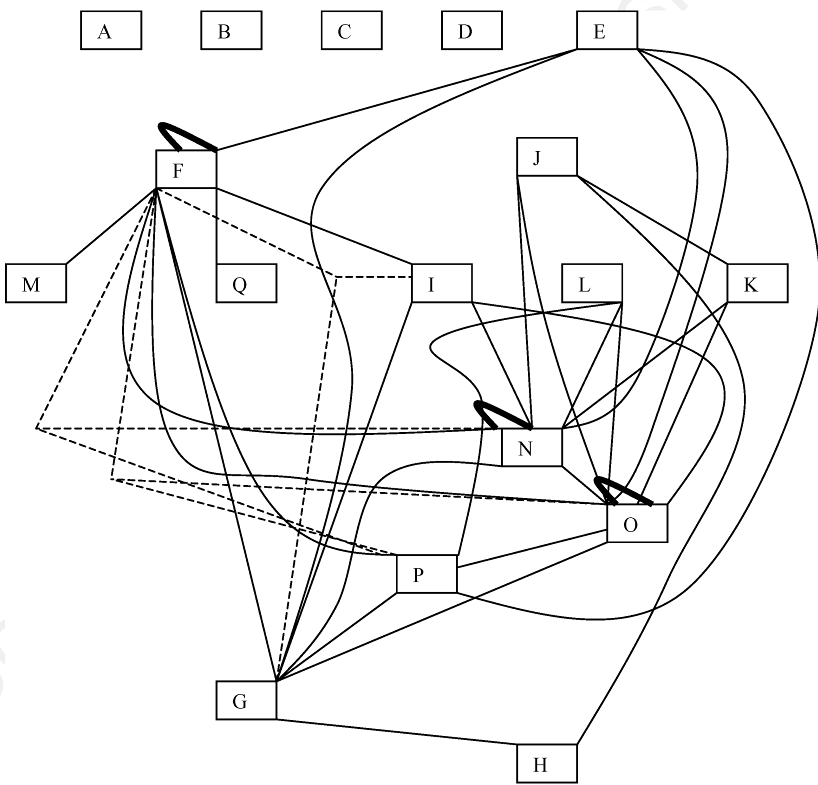


Figure 17. Extended conflict graph (*ECG*) for BTPC. All conflict, including the self-conflicts (boldface lines) and hyper edges (dashed lines), are propagated to arrays.

(iii) *Repeat till set R is empty.*

- *Assign the array with higher priority in set R to a module with higher priority in set S only (a) there is enough space to house the corresponding and (b) it satisfies the extended conflict graph (ECG) constraints.*

© 2005 Springer Science+Business Media Inc.

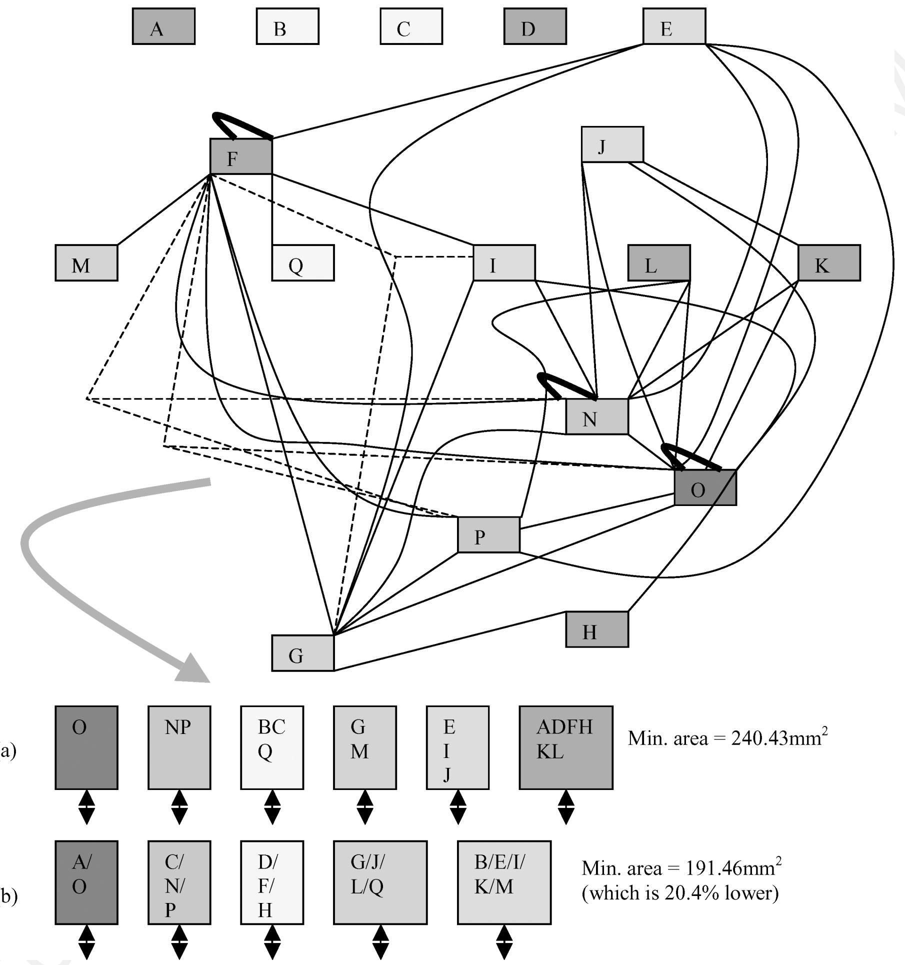


Figure 18. Minimize area, given energy bound (= 29,000 uJ). Example illustrating the best configuration (a) without lifetime consideration and (b) with lifetime consideration.

Energy bound (uJ)	*Area (mm²)*		% error
	ILP	*Heuristic*	
24,000	247.8	260	4.92
21,000	260.7	286.5	9.89
16,000	308.3	325	5.41

Figure 19. Minimum area memory configuration given energy bound for the BTPC example.

© 2005 Springer Science + Business Media Inc.

Area bound (mm^2)	*Energy (uJ)*		% error
	ILP	*Heuristic*	
315	18251	17421.8	4.76
275	19116	19270	.80
250	23695.4	24154	1.91

Figure 20. Minimum energy memory configuration given area bound for the BTPC example.

Given	Size (mm^2)	P	Final assignment	Energy (mJ)
M1	192	4	{G,K,F,N,D,A}	29.15
M2	32	1	{B,E,I,J}	8.7
M3	128	4	{C,O,P,M,Q,L }	17.54
M1	128	4	{E,I,J,D,O,P}	8.91
M2	128	4	{F,G,K,L,M,A,B,C,Q}	60.4
M3	64	2	{H,N}	5.0
M1	64	1	{A,B,C,D,H,Q,P.K}	16.60
M2	192	4	{E,G,I,M,N,Q,J}	33.60
M3	128	4	{F,O}	8.73

Figure 21. Results of array assignment for different memory configurations for the BTPC example.

- *Remove the visited array from set R.*
- *Update the size of the visited module.*

4.3.3. Results

In Figure 16, we show the array assignment for different memory configurations for the example in Figure 4. For these examples, the results of the final assignment are the same as that obtained by the heuristics.

5. Case Study: Binary Tree Predictive Coder (BTPC)

Binary Tree Predictive Coding (BTPC) is a general-purpose image coding method for lossless or lossy compression of photographs/graphics. In this section, we illustrate our procedures with the help of this example. To generate the ILP results, we have used LP_Solve version 3.0 in Sun Ultra 10 with Solaris 2.5 to run the BTPC examples. Each configuration took 0.5 to 7 minutes (depending on how close the bound is to upper bound or lower bound). The ECG given in [4] is used and shown in Figure 17.

In the ILP formulation, the number of the allowed permissible array groupings is 4,127. The lower bound of energy is 15494.22 uJ and the upper bound of area is 345.07 mm^2. For

© 2005 Springer Science+Business Media Inc.

Energy bound	Results *Without* life time consideration		Results *With* life time consideration		% red. in area
	Configurations	Area (mm^2)	Configurations	Area (mm^2)	
32,000	(O),(NP),(GJMQ),(EI),(ABCDFHKL)	238.5	(B/E/I/K/M),(G/J/L/Q),(D/F/H),(C/N/P),(A/O)	191.46	19.72
31,000	(O),(NP),(BCQ),(GJM),(EI),(ADFHKL)	239.1	(B/E/I/K/M),(G/J/L/Q),(D/F/H),(C/N/P),(A/O)	191.46	19.92
30,000	(O),(NP),(BCQ),(GJM),(EI),(ADFHKL)	239.1	(B/E/I/K/M),(G/J/L/Q),(D/F/H),(C/N/P),(A/O)	191.46	19.92
29,000	(O),(NP),(BCQ),(GM),(EIJ),(ADFHKL)	240.4	(B/E/I/K/M),(G/J/L/Q),(D/F/H),(C/N/P),(A/O)	191.46	20.37
26,000	(O),(NP),(ABCDHQ),(GJLM),(FJ),(EI)	243.8	(B/E/I/K/M),(G/J/L/Q),(D/F/H),(C/N/P),(A/O)	191.46	21.47
25,000	(O),(NP),(ABCDHQ),(GJM),(FKL),(EI)	244.0	(B/E/I/K/M),(G/J/L/Q),(D/F/H),(C/N/P),(A/O)	191.46	21.53
24,000	(O),(NP),(BCQ),(GKM),(FL),(EIJ),(ADH)	247.8	(B/E/I/K/M),(G/J/L/Q),(D/F/H),(C/N/P),(A/O)	191.46	22.74
21,000	(P),(N),(ABCDHQ),(EIJ),(MO),(G),(FKL)	260.7	(B/E/I/K/M),(G/J/L/Q),(D/F/H),(C/N/P),(A/O)	191.46	26.56
16,000	(KLM),(IJ),(H),(P),(O),(N),(BCQ),(G),(F),(E),(AD)	308.3	(M/O),(J),(I/K/L),(H),(N/P),(B/C/Q),(G),(F),(E),(A/D)	221.67	28.10
15,495	(A),(B),(C),(D),(E),(F),(G),(H),(I),(J),(K),(L),(M),(N),(O),(P),(Q)	347.8	(M/O),(J),(I/K/L),(H),(Q),(N/P),(B/C),(G),(F),(E),(A/D)	255.9	26.42

Figure 22. Minimize area, given energy bound. Results illustrating the % reduction in area for the BTPC application if life-time is considered.

energy bound of 29,000 uJ, the ILP solver finds the array groupings, $\times 4$, $\times 8$, $\times 1549$, $\times 146$, $\times 218$, $\times 2499$ to be equal to 1. Thus the final assignment has 6 memory modules with the configuration: $\{(O),(NP),(BCQ),(GM),(EIJ),(ADFHKL)\}$.

The minimum area of this assignment without lifetime consideration is 240.43 mm^2 and the minimum area of this assignment with lifetime consideration is 191.46 mm^2 (see Figure 18). Thus there is a 20.37% memory size reduction if we consider the of arrays. The final configuration with lifetime consideration is $\{(B/E/I/K/M),(G/J/L/Q),(D/F/H),(C/N/P),(A/O)\}$.

Figure 19 shows the minimum area obtained from ILP and heuristics for the BTPC example for different energy bounds. Similarly, Figure 20 shows the minimum energy

© 2005 Springer Science + Business Media Inc.

obtained from ILP and heuristics for the same example for different area bounds. Figure 21 shows the array assignment for different given memory configurations.

Figure 22 shows that the area is reduced significantly (average ∼22.68% reduction in area) if life-time of arrays is taken into account. Figure 23 shows that the energy is reduced significantly (average ∼13.36% reduction in energy due to a decrease in memory size if life-time of arrays is taken into account.

Area bound (mm^2)	Results *Without* life time consideration		Results *With* life time consideration		% red. in energy
	Configurations	Energy (uJ)	Configurations	Energy (uJ)	
350	(A),(B),(C),(D),(E),(F),(G),(H),(I),(J),(K),(L),(M),(N),(O),(P),(Q)	15493.6	(A/D),(B),(C),(E),(F),(G),(H),(I/L),(J),(K),(M/P),(N),(O),(Q)	15493.6	0
340	(A),(B),(C),(D),(E),(F),(G),(H),(I),(J),(KL),(M),(N),(O),(P),(Q)	15557.8	(A/D),(B),(C),(E),(F),(G),(H),(I/L),(J),(K),(M/P),(N),(O),(Q)	15493.6	0.41
330	(M),(KL),(J),(I),(P),(O),(N),(BCQ),(G),(F),(E),(D),(AH)	15655.1	(A/D),(B),(C),(E),(F),(G),(H),(I/L),(J),(K),(M/P),(N),(O),(Q)	15493.6	1.03
300	(P),(O),(N),(ABCJL),(HIKM),(DQ),(G),(F),(E)	15850.9	(A/D),(B),(C),(E),(F),(G),(H),(I/L),(J),(K),(M/P),(N),(O),(Q)	15493.6	2.25
290	(IKLM),(P),(O),(N),(BQ),(G),(F),(EJ),(C),(ADH)	17095.6	(A/D),(B),(C),(E),(F),(G),(H),(I/L),(J),(K),(M/P),(N),(O),(Q)	15493.6	9.37
280	(IJ),(P),(O),(N),(BHQ),(GK),(E),(DLM),(C),(AF)	18039.2	(A/D),(B),(C),(E),(F),(G),(H),(I/L),(J),(K),(M/P),(N),(O),(Q)	15493.6	14.11
270	(MN),(P),(O),(BCQ),(G),(FKL),(EIJ),(ADH)	19614.8	(A/D),(B),(C),(E),(F),(G),(H),(I/L),(J),(K),(M/P),(N),(O),(Q)	15493.6	21.01
250	(O),(NP),(BCQ),(GM),(FKL),(EIJ),(ADH)	23695.4	(M/N/P/Q),(K),(I/J/L),(H),(D),(B/C),(G),(F),(E),(A/D)	15498.7	34.59
245	(O),(NP),(ABCDHQ),(GKM),(FJL),(EI)	24794.6	(M/N/P/Q),(H),(D),(B/C),(G),(F),(E),(D/K/L),(A/I/J)	15511.4	37.44

Figure 23. Minimize energy, given area bound. Results illustrating the % reduction in energy for the BTPC application if life-time is considered.

© 2005 Springer Science + Business Media Inc.

6. Conclusion

In this paper, we have developed a memory design and exploration technique for multi-module, multi-port memories. We have established that loop transformations and reordering of array accesses simplifies the conflict graph and results in a more efficient memory. Next, we have developed ILP and heuristic-based procedures for memory allocation and assignment that satisfy area and/or energy constraints.

The heuristic algorithms are based on a thorough understanding of the area-energy trade-offs. While grouping with the same bit width causes a reduction in the area, it results in an increase in the energy. Similarly, splitting already grouped arrays causes a reduction in the energy but results in an increase in the area. A study of these tradeoffs helped reduce the search space during the memory exploration phase. The ILP models generate optimal solutions and have been used to evaluate the goodness of the heuristics. In all the examples, the results obtained by the heuristics were quite close with those obtained using the ILP models.

Our memory exploration procedures for multiple module memories rely heavily on the energy and area models. The better the model, the better the quality of the final solution. The current energy model considers only the switching component—that is, does not consider the leakage component. However, in deep submicron technology, the leakage component can be as significant, and thus its effect cannot be ignored. The effect of the leakage power on the procedures developed here remain to be investigated.

Acknowledgment

This work was supported by the NSF/S/IUCRC Center for Low Power Electronics (EEC-9523338), that is jointly funded by the National Science Foundation, the State of Arizona and the member companies.

References

1. Ahmad, I. and C.Y.R. Chen. Post Processor for Datapath Synthesis Using Multiport Memories. In *Proceedings of IEEE International Conference on Computer-Aided Design*, November 1991, pp. 276–279.
2. Balakrishnan, M., A. Majmudar, D. Banerji, J. Linders, and J. Majithia. Allocation of Multiport Memories in Data Path Synthesis. *IEEE Transactions on CAD/CAS*, vol. 7, no. 4, pp. 536–540, 1988.
3. Benini, L., A. Macii, and M. Poncino. A Recursive Algorithm for Low Power Memory Partitioning. In *Proceedings of ACM/IEEE International Symposium on Low Power Electronics and Design,* Rapallo, Italy, July 2000, pp. 78–83.
4. Catthoor, F., S. Wuytack, E. De. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology—Exploration of Memory Organization for Embedded Multimedia System Design*. Norwell, MA, Kluwer, 1998.
5. Gonzales, A., A. Aliagas, and M. Valero. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In *Proceedings of ACM International Conference on Supercomputing*, July 1995, pp. 338–347.

© 2005 Springer Science + Business Media Inc.

6. Grun, P., N. Dutt, and A. Nicolau. Access Pattern Based Local Memory Customization for Low Power Embedded Systems. In *Proceedings of the Design Automation and Test in Europe Conference*, Munich, Germany, March 2001.
7. Jha, P.K. and N. Dutt. Library Mapping for Memories. *European Design and Test Conference*, March 1997, pp. 288–292.
8. Khouri, K.S., G. Lakshminarayana, and N.K. Jha. Memory Binding for Performance Optimization of Control-Flow Intensive Behaviors. In *Proceedings of IEEE International Conference on Computer-Aided Design*, San Jose, CA, November 1999, pp. 482–488.
9. Kim, T. and C. Liu. Utlization of Multiport Memories in Data Path Synthesis. In *IEEE/ACM Design Automation Conference*, June 1993, pp. 298–302.
10. Lanneer, D., M. Cornero, G. Goosesens, and H. de Man. Data Routing: A Paradigm for Efficient Data-Path Synthesis and Code Generation. In *Proc. 7th ACM/IEEE International Symposium on High-Level Synthesis* May 1994, pp. 17–22.
11. Lee H.D. and S.Y. Hwang. A Scheduling Algorithm for Multiport Memory Minimization in Datapath Synthesis. *IEEE/ACM Design Automation Conference*, June 1995, pp. 93–100.
12. J.M. Mulder, N.T. Quach, and M.J. Flynn. An Area Model for On-Chip Memories and Its Application. *IEEE Journal on Solid-State Circuits*, vol. 26, pp. 98–105, 1991.
13. Panda, P.R. Memory Bank Customization and Assignment in Behavioral Synthesis. In *Proceedings of ACM/IEEE International Conference on Computer-Aided Design*. San Jose, CA, November, 1999, pp. 477–481.
14. Panda, P.R., F. Catthoor, N.D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P.G. Kjeldsberg. Data and Memory Optimization Techniques for Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 2, 2001.
15. Schmit, H. and D.E. Thomas. Synthesis of Application-Specific Memory Designs. *IEEE Transactions on VLSI Systems*, vol. 5, no. 1, 1997.
16. Shiue, Wen-Tsong. Memory Design and Exploration for Low Power Embedded Systems. Ph.D. Dissertation in Arizona State University, and Technical Report, CLPE-TR-4-2000-32 in Center for Low Power Electronics. May, 2000.
17. S. Wuytack, F. Catthoor, G. De. Jong, and H. De Man. Minimizing the Required Memory Bandwidth in VLSI System Realizations. *IEEE Transactions on VLSI Systems*, vol. 7, no. 4, Dec, 1999.

© 2005 Springer Science + Business Media Inc.