MEMORY DESIGN AND EXPLORATION FOR LOW POWER, EMBEDDED SYSTEMS

Wen-Tsong Shiue
Electrical Engr. Dept.
Arizona State University
Tempe, AZ

Chaitali Chakrabarti
Electrical Engr. Dept.
Arizona State University
Tempe, AZ

Abstract - In this paper, we describe a procedure for memory design and exploration for low power embedded systems. Our procedure tries to reduce the power consumption due to memory traffic by (i) applying memory optimizing transformations such as loop transformations, (ii) storing frequently accessed variables in a register file and an on-chip cache, and (iii) reducing the conflict misses by appropriate choice of cache size and data placement in off chip memory. We then choose a cache configuration (cache size, line size) that satisfies the system requirements of area, number of cycles and energy. We include energy in the performance metrics, since for different cache configurations, the variation in energy is quite different from the variation in the number of cycles. Our memory exploration procedure considers only a selected set of candidate points, thereby reducing the search time significantly.

INTRODUCTION

In systems that involve multidimensional streams of signals such as images or video sequences, it has been shown that the majority of the area and power cost is not due to the datapath or the controllers, but due to the global communication and memory interaction [1]. In fact, 50-80% of the power cost in application-specific circuits (ASIC) for real-time signal processing is due to memory traffic caused by transfers between the ASIC and the off-chip memories. This implies that with proper design, reduction in the memory related power budget can far exceed the reduction due to voltage scaling and other power saving transformations.

In this paper, we describe a procedure for memory design and exploration for low power embedded systems. We assume that there is a register file and a cache on-hip, and a large memory off-chip. Both the memories (cache and off-chip) are single port and single module memories. Our scheme has been successfully applied to small and medium sized programs.

The first step in our procedure is application of memory optimizing transformations such as loop transformations. While there exists a large number of loop transformations in the literature [2,3], here we focus on the ones that reduce the number of memory accesses and the size of the storage. Examples of such transformations are loop fusion, loop fission, loop interchange etc. Pioneering work on applying loop transformations to reduce power in data dominated applications has been done at IMEC [1]. The next step in our procedure is determining which variables should be assigned to register files and which variables should be assigned to on-chip cache. This is done by an LP-based procedure which tries to maximize the number of accesses to the register file (which is equivalent to minimizing the number of cache accesses) subject to constraints on the size of the register file and the size of the storage required for each of the variables. The third step is data placement in the off-chip memory so that the number of conflict misses is as small as possible. The fourth step in our procedure is choosing the cache configuration

(cache size and line size) that best satisfies the system requirements of area, energy and number of cycles. Our memory exploration procedure is an extension of the work by Panda et al [4] which focuses on two performance metrics – the cache size and the number of cycles. Our study shows that it is important to include energy in the performance metrics, since variation in the energy for different cache configurations is quite different from the variation in the number of cycles. Finally, in order to reduce the search time during the memory exploration phase, we have developed efficient algorithms that operate on only a small set of candidate points.

LOOP TRANSFORMATIONS

In this example, loop transformation procedures can be used to significantly reduce the number of accesses as well as reduce the size of the on-chip/off-chip memory. While these procedures were originally developed for software compilers, not all of them were integrated into the standard compilers. We illustrate the power of loop transformations with the example in Figure 1.

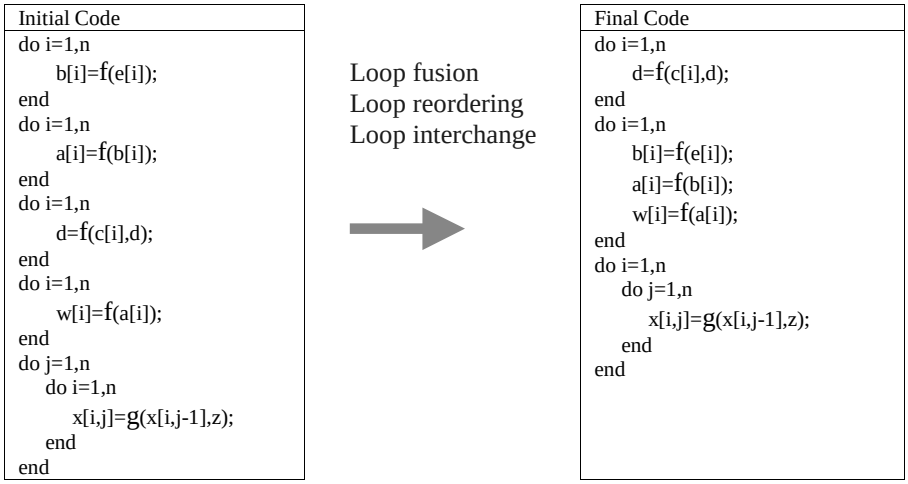


Figure 1. Example of loop transformations.

Loop fusion reduces the number of memory accesses since variables b[i] and a[i] can be stored in register files. Loop reordering allows array c[] and array w[] to share memory space, thereby reducing the size of the off chip memory. Finally, loop interchange helps to reduce the number of memory reads. We find that before doing loop transformations, we need off-chip memory of size 6n, n+2 on-chip registers and 8n accesses while after doing loop transformations, we only need off-chip memory of size 3n, 5 on-chip registers and 4n accesses.

In the examples that we have studied, we found that loop fusion, loop fission, loop interchange, loop reordering, loop tiling and loop skewing (which facilitates tiling) are the most powerful loop transformations. Our study indicates that loop fusion should not be done unless there are data dependencies between the two fusing loops that can be exploited to reduce the number of memory accesses. This is because loop fusion causes an increase in the size of the loop body which in turn causes an increase in the minimum cache size (that is required to avoid conflict misses) which

in turn causes an increase in the energy consumption. In the same note, loop fission should be done if the statements in a loop have no data dependencies.

Determining the 'best' order in which these transformations have to be applied is very difficult. A loop transformation framework that considers loop interchange, loop reversal, loop skewing and loop tiling to improve data locality have been developed in [3]. We are currently trying to develop heuristics to help determine which transformations should be applied where and in what order.

LP MODEL FOR REGISTER ASSIGNMENT

The next step in our procedure is to decide which variables will be stored in a register file. Clearly, variables which result in maximum savings in the number of memory accesses are prime candidates. Our method is based on linear programming and operates at a high level (not RT level). The application program is represented by a dependence graph (DG) where each node corresponds to a variable (in the program) and an edge exists between nodes if there are dependencies between the corresponding variables (see Figure 2(b)).

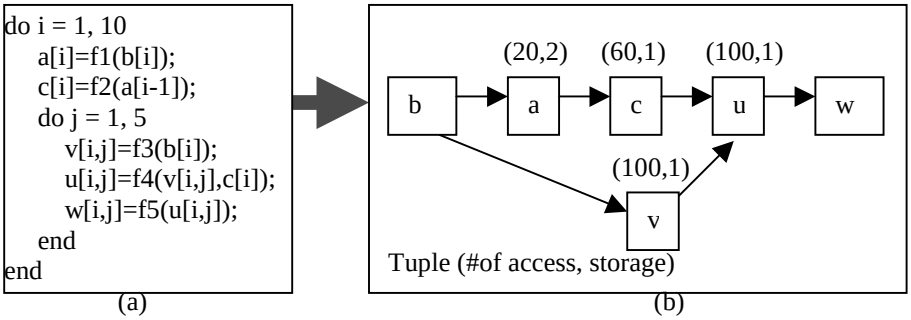


Figure 2. (a) Application program (b) Dependence graph

For instance, in the program described in Figure 2(a), a and v are functions of b, and thus there are edges between (b,a), and (b,v). For each intermediate variable (represented by the non-terminal nodes), we calculate the number of memory accesses and the storage requirement. The number of memory accesses is a function of the trip count of the variable. The storage requirement of a variable is determined by the maximum difference of the indices of that variable. For instance, the number of accesses of the intermediate variable a[i] is 2* trip_count_i = 2*floor((10-1+1)/1)= 20 [the factor 2 is due to the same number of reads and writes]. The storage requirement of a[i] is 2, since the maximum difference between its indices is 0-(-1)+1=2. Similarly, the intermediate variable u[i] has 2 * trip_count_i * trip_count_j = 2*10*5=100 accesses and a storage requirement of 1. In order to get the optimal solution for the register assignment, we develop a linear programming model. The aim is to maximize the number of register accesses (which is equivalent to minimizing the memory accesses) subject to the register size constraints and storage constraints for each of the variables. The LP model is described below. We assume b=x1, a=x2, c=x3, u=x4, v=x5 and w=x6.

Objective function:	max:20x2+60x3+100x4+100x5;
Constraints for register size:	2x2+x3+x4+x5 <= register size;

Constraints for variable size: 2x2<=2; x3<=1; x4<=1; x5<=1;

Our LP model can be solved by the LP_solver. For instance, if the register size is limited to 3, then x3=x4=x5=1. That means variables c, u and v can be assigned to register files. Such an assignment results in maximum saving in the number of memory accesses. The other array-based variables such as variables a, b and w are then assigned to the cache. If the LP_solver gives a solution that is non-integer, then we approximate the solution with the floor ($\lfloor \rfloor$) of the non-integer value. While the LP-based solution is clearly not optimal, we still choose the LP method over ILP on account of its lower complexity.

MEMORY EXPLORATION

The memory exploration procedure tries to find the cache configuration (cache size, line size) that best satisfies the system requirement of number of cycles and energy. The area bound is used to determine the largest possible cache size. For different cache size C (that is a power of two), and for different cache line sizes L, we estimate the number of processor cycles and energy consumption. We then select the cache configuration that best matches the time and/or energy requirements. For large programs, the number of processor cycles and energy consumption can be calculated very easily for each cache configuration (C,L). If tr(j) is the number of times the j^{th} kernel program is invoked, T(j) is the number of cycles of the j^{th} kernel program and E(j) is the energy of the j^{th} kernel program, then

$$CYCLES=\Sigma_j T(j)*tr(j); \qquad ENERGY=\Sigma_j E(j)*tr(j).$$

Processor Cycles: The number of processor cycles that would be required to execute the code is a function of the miss rate. We adopt the model used in [5] and assume that the number of cycles per hit is 1, 1.1, 1.12, 1.14 for 1, 2, 4, and 8-way set associative cache respectively. We also assume that the number of cycles per miss is 40, 40, 42, 44, 48, 56, and 72 for line sizes of 4, 8, 16, 32, 64, 128, and 256 respectively. The number of processor cycles is given by

T = hit_rate*trip_count*(number of cycles per hit) + miss_rate*trip_count*(number of cycles per miss)

Energy: Our energy model for on-chip cache is derived from [6] and is given below. We use Em=4.95 nJ (corresponding to SRAM CY7C1326-133).

Energy = Hit rate * Energy_hit + Miss rate * Energy_miss
Energy_hit = E_dec+E_cell
Energy_miss = E_dec+E_cell+E_io+E_main = Energy_hit + E_io+E_main
- E_dec = α*(Add_bs)
- E_cell = β*(Word_line_size) * (Bit_line_size)
- E_io = γ*(Data_bs* Cache_line_size + Add_bs)
- E_main = γ*(Data_bs* Cache_line_size) + Em * Cache_line_size

where

Add_bs = Number of bit switches on address bus per instruction.
Data_bs = Number of bit switches on data bus per instruction
Word_line_size = Number of memory cells in a word line.
Bit_line_size = Number of memory cells in a bit line.
Em = Energy consumption of a main memory access
α=0.001, β=2 and γ =20 are used for 0.8 μm CMOS technology

Finding the Minimum Cache Size

In this section we describe a procedure for computing the minimum cache size that is required to avoid cache conflicts. The cache conflicts occur when data that could possibly be reused in the near future, is mapped to the same cache line by subsequent accesses in a direct-mapped cache. For each kernel program, we partition references in a loop nest into equivalent classes of reference or equivalent cases of reference. For each class or case, we find the number of cache lines [7]. The minimum cache size (MCS) is line size times the total number of cache lines. If the cache size is smaller than MCS, the miss rate increases significantly causing a significant increase in both the number of cycles and energy. For instance, for the example Compress (see Figure 3), MCS is 4L, and thus there is a significant increase in number of cycles and energy for cache sizes < 4L. The point in the curve corresponding to MCS is referred to as the inflection point.

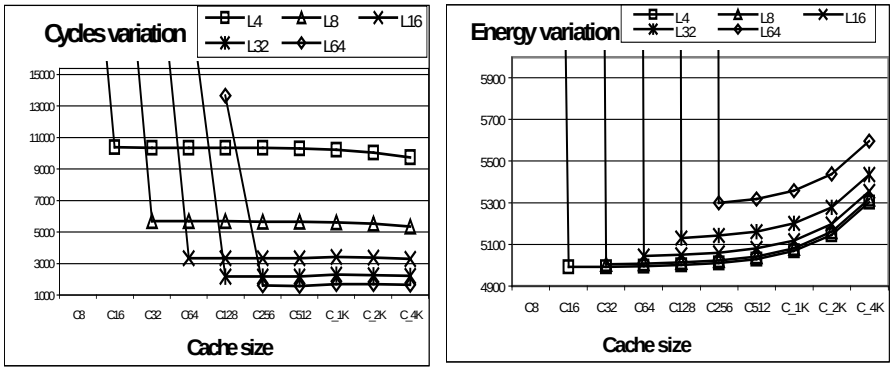
Finding the minimum cache size for large programs is a lot more involved. For starters, the minimum cache size for a large program can be larger than the MCS of its kernel programs. This is because the minimum cache size is determined by the placement of the arrays in the cache, and thus arrays which are part of multiple kernel programs can black out certain cache lines. For instance, consider an example with three arrays a, b, and c in a large program consisting of three kernel programs. Array a is part of kernels 1 and 2, array b is part of kernels 2 and 3 and array c is part of kernels 1 and 3. If the number of cache lines per array is 2, then the minimum cache size of each kernel program is 4 lines but the minimum cache size of large program is 6 lines.

A brief sketch of our procedure for finding the minimum cache size is as follows. In the first step, we create an array compatibility graph and find the maximal cost clique. The weight of each node in this graph is the number of cache lines that are assigned to that array and thus the cost of the clique is equivalent to the minimum cache size. In the next step, we identify the nodes in the clique that can share cache lines with the nodes outside the clique. The algorithm operates on one node at a time with higher priority given to high cost nodes. The details of this algorithm are beyond the scope of this paper.

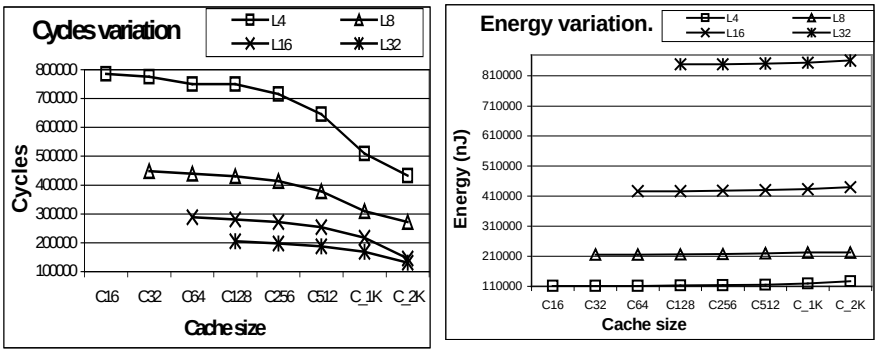
Tradeoffs between time and energy: Figure 3(a) shows the variation in the number of cycles and energy consumption for different cache and line sizes for the example Compress. Note that while the number of cycles reduces with increase in cache size and line size, the energy consumption increases. This is due to the fact that for a miss, the cycles penalty (or the number of cycles per miss) is not as significant as the energy penalty (or Energy_miss). For instance, for the same cache size, as the line size increases, the miss rate reduces and the number of cycles per miss increases mildly. As a result, the number of cycles reduces with increase in line size. On the other hand, an increase in the line size is accompanied by a large increase in the energy of a miss. This is because a large line size implies that a large amount of data have to be brought from the off-chip memory resulting in significant switching in the I/O pads of the off-chip memory and host processor. Thus the reduction in the miss rate is offset by the large increase in the energy of a miss. As a result, the energy consumption increases with the increase in line size. The study of the variation in the number of cycles and energy for different cache sizes and line sizes can be used to obtain the best cache configuration given the energy and/or

time bounds. It is interesting to note that the minimum time cache configuration corresponds to the largest possible cache and the minimum energy cache configuration corresponds to the smallest possible cache.

```
Example 1: Compress
int a[32,32]
for  i=1,31
    for j=1,31
        a[i,j]=a[i,j]-a[i-1,j]-a[i,j-1]-2*a[i-1,j-1];
```



(a)



(b)

Figure 3. Variation in the number of cycles and energy for different cache sizes and line sizes for (a) Compress kernel program (b) MPEG1 Decoder.

The trend in the cycles vs cache size and energy vs cache size is the same for large programs. Figure 3(b) shows the variation in the number of cycles and energy for different cache sizes and line sizes for the MPEG1 decoder. Note that with the increase in the cache size, while the number of cycles decreases significantly, the energy increases mildly. For instance, for L=8, the number of cycles drop from 447,500 at C_{32} to 262,000 at C_{2K}, and the energy increases from 213,000 at C_{32} to 222,000 at C_{2K}.

Data placement method: Cache performance can be improved significantly by suitable data placement in off chip memory, by tiling and by increasing the level of

set associativity [7]. Figure 4 describes how the miss rate is reduced dramatically by proper data placement for the example Compress. While proper data placement causes a significant reduction in the number of cycles, the reduction in energy is not as dramatic. This is because Energy_miss >> Energy_hit, and so a drastic reduction (5X) in the miss rate results in only a modest reduction (2X) in the energy. Once the placement of the arrays in the cache is known, placement of the arrays in the main memory is relatively straightforward. The main memory placement is done such that the amount of unused space is as small as possible. We propose a greedy algorithms that operates on one array at a time. In every step, the algorithm finds the largest array that can be placed in the main memory such that the amount of unused space is minimum. For instance, consider an example with C=20, L=4; arrays a, b, c, d, and e are of sizes, 20, 30, 10, 50 and 40 respectively; a and b are mapped to line 0, c and d are mapped to line 2 and e is mapped to line 4 of the cache. Our algorithm first assigns b and places it in memory locations 0 through 29. Next it places e in locations 36 through 75 since this assignment results in the least amount of unused space (6 locations). The final placement is as follows. b: 0-29, e: 36-75, a: 80-99, d: 100-149, c: 168-177.

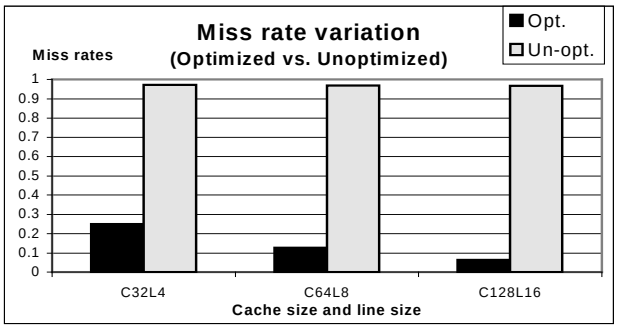


Figure 4. Example Compress. Miss rate reduction due to data placement method.

Algorithms to choose the cache configuration (cache size, line size): We present efficient algorithms to find the minimum energy cache configuration given the time bound and the minimum time cache configuration given the energy bound. These algorithms do not look at all possible cache configurations but only a small subset of configurations to find the minimum energy or the minimum time cache configurations. This reduces the time to search for the 'best' cache configuration significantly. The notations used here are shown below.

$C_s(L)$: smallest cache size (which satisfies the cycle bound) for L.
$C_L(L)$: largest cache size (which satisfies the energy bound) for L.
$C_i(L)$: inflection point for L curve.
CYC(C,L): number of cycles for cache size C and line size L.
E(C,L): energy for cache size C and line size L.
T_{bound}: cycle bound.
E_{bound}: energy bound.

Algorithm_min-energy
Step 1: for each line size, L.{
Calculate the number of cycles at C_{max}, CYC(C_{max}(L),L).

Compare with T_{bound}.}

Step 2: Denote the smallest line size that satisfies the cycle bound at C_{max} by L_1.

Step 3: Calculate the number of cycles at the inflection point of L_1, $CYC(C_i(L_1),L_1)$.
if $CYC(C_i(L_1),L_1) < T_{bound}$, {
{$C_i(L_1),L_1$} is the minimum energy cache configuration else
do step 4,5,6,7 and 8.}

Step 4: for each line size $L > L_1${
Calculate the number of cycles at its inflection point, $CYC(C_i(L),L)$.
Compare with T_{bound}.}

Step 5: Denote the smallest line size that satisfies the cycle bound at the inflection point by L_2.

Step 6: for each line size L that lies between L_1 and L_2,(i.e. $L_1<=L<L_2$){
Subroutine smallest_cache_size (L).
Calculate $E(C_s(L),L)$.}

Step 7: Calculate $E(C_i(L_2),L_2)$.

Step 8: Choose the cache configuration with the lowest energy.

Subroutine_smallest_cache_size (L):
(i.e. find the smallest cache size, C_s, that satisfies the cycle bound).
a. The left point corresponds to the inflection point, (T1,C1).
The right point corresponds to the maximum cache size, (T2,C2)
b. Calculate $\check{C}$ which is the cache size if the number of cycles curve had been a straight line. $\check{C} = C1+(T_{bound}-T1)*((C2-C1)/(T2-T1))$
c. if $\check{C}$ is not a power of 2
choose $\check{C}_2$ which are the closest powers of 2 such that $\check{C} < \check{C}_2$
d. Calculate the no. of cycles $\check{T}_2$ and $\check{C}_2$
e. if $\check{T}_2 > T_{bound}$ {$T1= \check{T}_2$; $C1= \check{C}_2$; repeat step b, c and d.}else
$\check{C}_2$ is the smallest cache size that satisfies the cycle bound.
(i.e. $\check{C}_2 = C_s(L)$)

Algorithm_min-time

Step 1: for each line size, L.{
Calculate the energy at the inflection point, $E(C_i(L),L)$.
Compare with E_{bound}.}

Step 2: Denote the largest line size that satisfies the energy bound at the inflection point by L_2.

Step 3: Calculate the energy of line L_2 at C_{max}, $E(C_{max}(L_2),L_2)$.
if $E(C_{max},L_2) < E_{bound}$, {{$C_{max},L_2$} is the minimum time cache configuration else
do step 4,5,6,7 and 8.}

Step 4: for each line size $L < L_2$, {Calculate the energy at C_{max}, $E(C_{max}(L),L)$;
Compare with E_{bound}.}

Step 5: Denote the largest line size that satisfies the energy bound at C_{max} by L_1.

Step 6: for each line size L that lies between L_1 and L_2,(i.e. $L_1<L<=L_2$){
Subroutine largest_cache_size (L);
Calculate $CYC(C_L(L),L)$.}

Step 7: Calculate $CYC(C_{max}(L_1),L_1)$.

Step 8: Choose the cache configuration with the smallest number of cycles.

Subroutine_largest_cache_size (L):
(i.e. find the largest cache size, C_L, that satisfies the energy bound).
a. The left point corresponds to the inflection point, (E1,C1).
The right point corresponds to the maximum cache size, (E2,C2)
b. Calculate $\check{C}$ which is the cache size if the energy curve had been a straight line. $\check{C} = C1+(E_{bound}-E1)*((C2-C1)/(E2-E1))$
c. if $\check{C}$ is not a power of 2
Choose $\check{C}_2$which are the closest powers of 2 such that $\check{C} < \check{C}_2$
d. Calculate the energy $\check{E}_2$ at $\check{C}_2$
e. if $\check{E}_2 < E_{bound}$ {$E1= \check{E}_2$; $C1= \check{C}_2$; repeat step b, c and d} else
$\check{C}_1$ is the largest cache size that satisfies the energy bound.
(i.e. $\check{C}_1= C_L(L)$)}

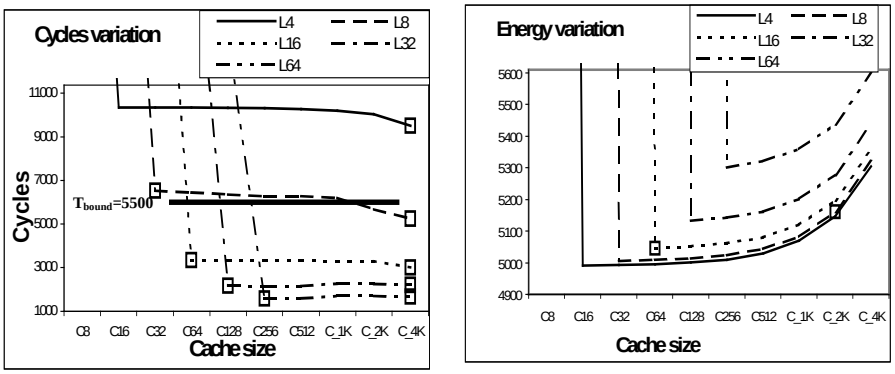


Figure 5. Illustration of the algorithm for minimum energy cache configuration.

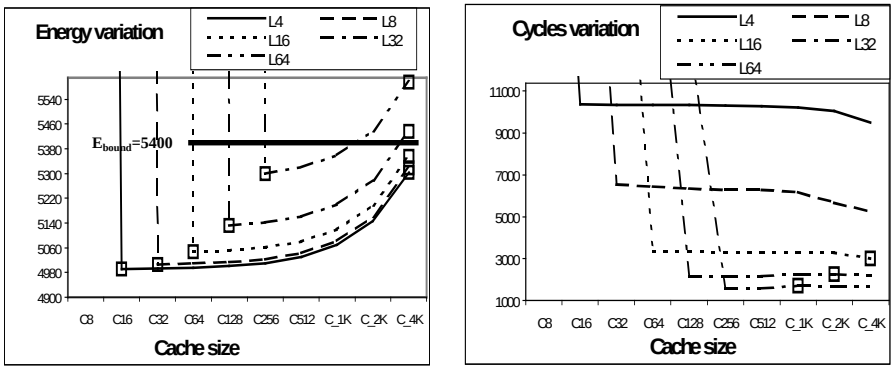


Figure 6. Illustration for algorithm of minimum time cache configuration.

We use the Compress example to first illustrate the minimum energy algorithm and then the minimum time algorithm. Let us say that our goal is to find the minimum energy cache configuration for T_{bound} =5500. In the first step, we calculate the number of cycles at C_{4K} and find that the smallest line size that satisfies the cycle bound at C_{4K} is L_8. Since the cycle bound is not satisfied at the inflection point of the L_8 curve, we have to calculate the number of cycles at the inflection points for all line sizes larger than 8. We find that L_{16} satisfies the cycle bound at the inflection point (C_{64}) and so $C_{64}L_{16}$ is one of the candidate points. For L_8, we find that the smallest cache size that satisfies the cycle bound is C_{2K}. We calculate the energy at $C_{2K}L_8$ and $C_{64}L_{16}$ and find that the minimum energy cache configuration corresponds to $C_{64}L_{16}$. Figure 5 marks the points where the number of cycles or energy had to be computed. This is a small fraction of points compared to the number of points that would have to be considered in an exhaustive search.

Next, let us say that our goal is to find the minimum time cache configuration for E_{bound} =5400. We first calculate the energy at the inflection point of each line and find that the largest line size that satisfies the energy bound is L_{64}. Since the energy at $C_{4K}L_{64}$ is larger than the energy bound, we calculate the energy at C_{4K} for all line sizes < 64. We find that $C_{4K}L_{16}$ satisfies the energy bound and mark it as a candidate point. Next, we find the largest cache size that satisfies the energy bound

for line sizes of 32 and 64, and mark $C_{2K}L_{32}$, and $C_{1K}L_{64}$ as candidate points.. We calculate the number of cycles at all three candidate points and find that the minimum time cache configuration corresponds to $C_{1K}L_{64}$. Note that the number of points at which the number of cycles or energy had to be computed is very small (see Figure 6).

CONCLUSIONS

In this paper, we have described a memory design and exploration procedure for low power embedded systems. Our procedure consists of (i) applying high level transformations such as loop transformations to reduce the number of off-chip memory accesses and the size of off-chip memory, (ii) storing the most frequently accessed variables in register files and (iii) using data placement in off chip memory to avoid conflict misses, (iv) determining the cache configuration (cache size and line size) that best satisfies the system requirements of area, time and energy.

ACKNOWLEDGMENTS

This work was carried out at the Center of Low Power Electronics which is supported by the National Science Foundation, the state of Arizona, and 13 member companies from the microelectronics industry. The authors would like to thank Andreas Wild of Motorola for many helpful discussions.

REFERENCES

[1] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, H. De Man, "Global Communication and Memory Optimizing Transformations for Low Power Signal Processing Systems", *Workshop on VLSI Signal Processing*, La Jolla CA, Oct 1994.

[2] D. F. Bacon, S. L. Graham, O. Sharp, "Compiler Transformations for High-Performance Computing", *ACM Computing Surveys*, vol. 26, no. 4, Dec. 1994.

[3] M. E. Wolf and M. Lam. "A Data Locality Optimizing Algorithm." In proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation, pages 30-44, June 1991.

[4] P. R. Panda, N. D. Dutt and A. Nicolau, "Memory Data Organization for Improved Cache Performance in Embedded Processor Applications", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 2, No. 4, Oct. 1997.

[5] J. L. Hennessy and D. A. Patterson, "Computer Architecture A Quantitative Approach", 2nd edition, *Morgan Kaufman Publishers*, 1996.

[6] C. Su and A. Despain, "Cache Design Trade-offs for Power and Performance Optimization: A Case Study", *International Symposium on Low Power Electronics and Design*, pages 63-68, 1995.

[7] Wen-Tsong Shiue and Chaitali Chakrabarti, "Memory Exploration for Low Power, Embedded Systems", *36th ACM/IEEE Design Automation Conference*, pages 140-145, 1999.