# Architectural Exploration and Optimization of Local Memory in Embedded Systems*

Preeti Ranjan Panda    Nikil D. Dutt    Alexandru Nicolau

Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA

## Abstract

*Embedded processor-based systems allow for the tailoring of the on-chip memory architecture based on application-specific requirements. We present an analytical strategy for exploring the on-chip memory architecture for a given application, based on a memory performance estimation scheme. The analytical technique has the important advantage of enabling a fast evaluation of candidate memory architectures in the early stages of system design. Our experiments demonstrate that our estimations closely follow the actual simulated performance, at significantly reduced run times.*

## 1. Introduction

Increasing design complexity and shrinking product design cycle times have fueled the need for design reuse in the IC-design industry. Reuse is enabled by modern design libraries, which frequently consist of pre-designed megacells such as microprocessor cores, memories, numeric coprocessors, and modules implementing standardized functions such as JPEG. For example, the CW33000 processor core from LSI Logic, the TMS320 series of DSP processors from Texas Instruments, the StrongARM processor from Advanced RISC Machines are available in the form of *embedded cores*, in addition to the standard packaged form. Memory-based mega-modules are widely used in the industry today in the form of cache, Scratch-pad SRAM [8] and embedded DRAM [18].

An embedded processor-based system allows for customization of on-chip memory configuration according to the requirements of a specific application. This is an important feature, because a large portion of typical (packaged, off-the-shelf) microprocessors is occupied by cache memory. However, if an analysis reveals that a smaller on-chip memory results in a satisfactory performance, the resulting die area could be reduced by using the processor core with an appropriately sized on-chip memory. The large body of work on cache organization in the traditional computer architecture domain has focussed on improving the average cache performance over a large variety of applications, relying on benchmark suites such as SPEC [16]. The impact of cache memory parameters on program behavior has been widely studied in the past [14]. Techniques for modeling cache features and utilizing them in program transformations have been reported in [1, 19]. However, individual applications have widely varying memory characteristics, and in an application-specific IC, it is essential to optimize performance by tailoring the on-chip memory organization to the requirements of a given application. We address this issue in this paper.

In high-level synthesis of application-specific systems, researchers have addressed the problem of generating efficient multiport memory organizations for storing behavioral scalar and array variables [17, 15]. A strategy for alignment of behavioral arrays in off-chip memory in order to improve cache performance was reported in [10]. The problem of memory packing – determining the memory configuration for logical behavioral arrays based on a library of physical memory components has been addressed in [6, 5]. In the PHIDEO synthesis system [7] multiport memory allocation is performed for multidimensional data streams, taking interconnect costs into account. The CATHEDRAL system [3] uses a data-flow analysis technique to determine a memory architecture consisting of one or more (possibly multiport) memories satisfying a given timing constraint. However, the above works have not addressed the memory architecture issues involved when the data size is too large to be stored on-chip.

We present an exploration strategy for determining an efficient on-chip memory architecture – characterized by Scratch-pad memory size and cache parameters – based on an analysis of a given application, so as to reduce off-chip

*This work was partially supported by grants from ARPA (MDA904-96-C-1472), NSF(CDA-9422095), and ONR(N00014-93-1-1348).

memory traffic. The advantage of such an analytical approach over a simulation-based strategy is that, it is possible to do a fast comparison of the expected performance resulting from many different memory architectures, whereas simulation of the application for all the possible memory configurations could be prohibitively expensive, and impractical for purposes of exploration.

## 2. Architecture

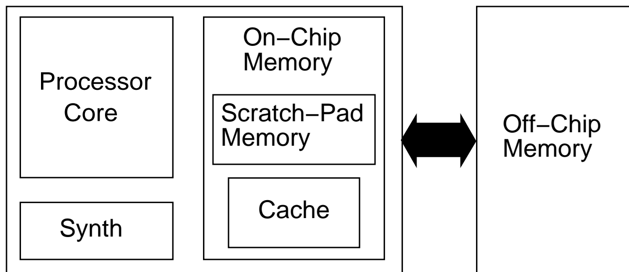



**Figure 1. Embedded Processor-based Architecture**

Figure 1 shows a simplified view of an embedded processor core-based system, consisting of a processor core, on-chip memory, and synthesized block (*Synth*), interfacing with off-chip DRAM. The synthesized hardware block *Synth*, often the result of behavioral synthesis, performs the functions specific to the application that are mapped to hardware, possibly for performance considerations. This decision of mapping different parts of a design into hardware and software is taken in a prior Hardware/Software partitioning step. In this paper, we assume that the partitioning has already been performed. The on-chip memory can be implemented as a combination of cache and *Scratch-pad SRAM*. In the current work, we limit our attention to on-chip data memory.

Data cache is fast, on-chip memory forming an interface between the processor and the off-chip DRAM, that reduces the effective memory access time by storing recently accessed data [14]. Internally, the cache is divided into blocks, or *cache lines*, which constitute the smallest unit of interaction between the cache and the off-chip memory.

Scratch-pad SRAM is on-chip memory, to which the assignment of data is compiler-controlled. The overall architecture is described in detail in [13]. In brief, a portion of the total data memory space is mapped to on-chip SRAM (typically used to store critical data), with the advantage of guaranteed fast access, unlike the cache, where hardware-controlled storage and replacement strategies could flush out data into the off-chip memory, resulting in cache misses that stall the processor. In some modern embedded systems such as the F/X256 graphics controller from Silicon Magic Corporation, the Scratch-pad memory assumes the form of *embedded DRAM* [18].

## 3. Illustrative Example

We illustrate our strategy for architectural exploration and optimization of on-chip memory on CONV, a convolution program frequently used in Image Processing tasks such as edge detection, regularization, and morphological operations [2]. The code for CONV is shown below:

```
N = 128; M = 4; NORM = 16;
int source[N][N], dest [N][N], mask [M][M];
Procedure CONV
int acc, i, j, x, y;
for (x = 0; x < N − M; x++)
    for (y = 0; y < N − M; y++) {
        acc = 0;
        for (i = 0; i < M; i++)
            for (j = 0; j < M; j++)
                acc = acc + source[x+i][y+j] * mask[i][j];
        dest[x+M/2][y+M/2] = acc/NORM;
    }
```

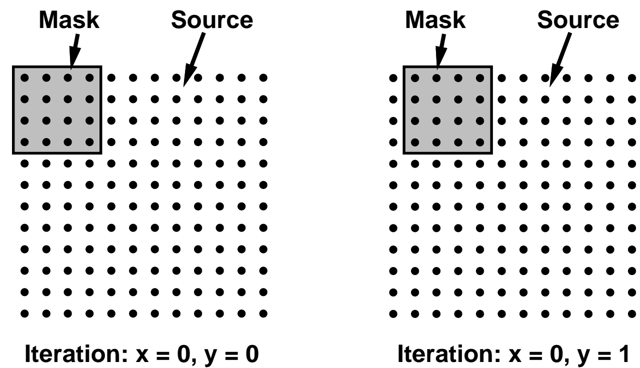



**Figure 2. Memory access pattern in CONV example**

A small $(4 \times 4)$ matrix of coefficients, *mask*, slides over the input image, *source*, covering a different $4 \times 4$ region in each iteration of $y$, as shown in Figure 2. In each iteration, the coefficients of *mask* are combined with the region of the image currently covered, to obtain a weighted average, and the result, *acc*, is assigned to the pixel of the output array, *dest*, in the center of the covered region.

We first attempt to solve the following problem: given a maximum amount, say 4 KB of on-chip memory space, find an efficient utilization of the space, in terms of data cache size, cache line size, and Scratch-pad memory size, with the objective of minimizing the number of processor cycles required to access the arrays from memory. We assume that scalar variables are assigned to registers; the data cache is

direct-mapped and write-through [14]; and the largest allowed cache line size is 128 Bytes.

We note that, since we use a write-through cache, in which memory writes do not interfere with the cache contents in case of cache misses, the access to *dest* does not cause cache conflicts. However, if the two arrays *source* and *mask* were to be accessed through the data cache, the performance would be affected by cache conflicts. Further, an associative cache, by itself, will not eliminate the problem in general, because most practical caches have a limited associativity, and the general situation might require an associativity as large as the number of conflicting arrays.

The conflict problem can be solved by storing the small *mask* array in the Scratch-pad memory. This assignment eliminates all conflicts in the data cache – the data cache is now used for memory accesses to *source*, which are very regular. Also, since $M(=4)$ rows of the *source* array are *active* at any point in time, the data cache need be only as large as 4 rows of the source array. [1] Thus, we select a data cache of size $M \times N = 4 \times 128 = 512$ words = 2 KB. Since the accesses to *source* have good spatial locality, we select the largest allowed cache line, i.e., 128 bytes. The Scratch-pad memory size is $4 \times 4$ words = 64 Bytes.

# 4. Memory Architecture Exploration

In our formulation, a local (on-chip) memory architecture for an application is defined as a combination of:

- The total size of on-chip memory used for data storage.
- The partitioning of this on-chip memory into: (1) Scratch-pad SRAM, characterized by its size; and (2) data cache, characterized by the cache size; and the cache line size.

The basic algorithm for memory architecture exploration is summarized below:

Algorithm *MemExplore*
$L_1$: **for** on-chip memory size $T$ (in powers of 2)
  $L_2$: **for** cache size $C$ (in powers of 2, $< T$)
    SRAM Size $S = T - C$
    *DataPartition*($S$)
    $L_3$: **for** line size $L$ (in powers of 2, $< C$, $<$ *MaxLine*)
      Estimate Memory Performance
  Select $(C, L)$ that maximizes performance

For each candidate on-chip memory size $T$ (loop $L_1$), we consider different divisions of $T$ (loop $L_2$) into cache (size $C$) and Scratch-pad SRAM (size $S = T - C$), selecting only powers of 2 for $C$. Procedure *DataPartition* is based on a technique for partitioning program variables into Scratch-pad memory and cache [13]. Scalar and array data identified to be the most critical, are assigned to the SRAM, based on the data size, the memory access frequency and the possibility of cache conflicts. In the rest of this section, we describe the memory performance estimation step. For each $T$, we select the $(C, L)$ pair that is estimated to maximize performance. Finally, we perform the memory address assignment of the variables using the cache and SRAM parameters selected using the algorithms in [12] and [13].

[1] As mentioned before, the *dest* array is not allocated any storage in a write-through cache. If a *write-back* cache were used instead, we would allocate an equal cache space to *dest*, along with an appropriate memory assignment so that *source* and *dest* do not conflict in the cache [12].

## 4.1. Memory Performance Estimation

There is a trade-off in sizing the cache line. If the memory accesses are very regular and consecutive, i.e, exhibit spatial locality, a longer cache line is desirable, since it minimizes the number of off-chip accesses and exploits the locality by pre-fetching elements that will be needed in the immediate future. On the other hand, if the memory accesses are irregular, or have large strides, a shorter cache line is desirable, as this reduces off-chip memory traffic by not bringing unnecessary data into the cache. The maximum size of a cache line is the DRAM page size, which is usually less than 1 KB = 256 $(= 2^8)$ words for most modern DRAMs. Thus, there are typically a maximum of 9 alternatives for the cache line size (which is usually a power of 2).

Suppose there are $N$ scalar variables stored in off-chip memory, accessed $M$ times in the program. We store all scalar variables in consecutive locations in memory. Since accesses to scalars invariably constitute a small fraction of the total accesses, we make the simplifying assumption that there is only one cache miss (a *compulsory* miss that occurs when the variable is first accessed into the cache) for every cache line containing scalars. Although this assumption looks too optimistic, it is actually reasonable when combined with our partitioning strategy for scalar and array variables – most of the scalars get mapped to the register file and Scratch-pad memory, and not to the cache. Since there are $N$ scalars, we require $\lceil \frac{N}{L} \rceil$ cache lines for them for a cache line size of $L$, i.e., there are $\lceil \frac{N}{L} \rceil$ cache misses, and consequently, $M - \lceil \frac{N}{L} \rceil$ cache hits for the $M$ accesses. A memory access resulting in a cache hit requires one cycle, while a cache miss entails a delay of $(K + L)$ processor cycles, where $K$ is a constant (usually 10–20 in modern processors [14]). Thus, the total number of processor cycles required for the $M$ accesses to scalar variables is:

$$\text{Cycles (Scalars)} = (K + L)\left\lceil \frac{N}{L} \right\rceil + M - \left\lceil \frac{N}{L} \right\rceil \qquad (1)$$

We determine an estimate of the processor cycles required to access the array elements by first dividing the application program into loop nests. Straight line code is

**for** $i = 1$ **to** $M - 1$ **step** 1
  **for** $j = 1$ **to** $M - 1$ **step** 1
    $A[i][j] = A[i][j] + A[i-1][j] + A[i+1][j] +$
      $A[i][j-1] + A[i][j+1] + B[i] + C[j][i]$

(a)

$L_1$ : **for** $i_1 = l_1$ **to** $h_1$ **step** $s_1$
  $L_2$ : **for** $i_2 = l_2$ **to** $h_2$ **step** $s_2$
    ...
    $L_m$ : **for** $i_m = l_m$ **to** $h_m$ **step** $s_m$
      Read $a[i_1][i_2] \dots [i_m]$
      ...
      $L_n$ : **for** $i_n = l_n$ **to** $h_n$ **step** $s_n$
        **Read** $b[i_1 + k_1][i_2 + k_2] \dots [i_n + k_n]$
        **Read** $b[i_1 + k'_1][i_2 + k'_2] \dots [i_n + k'_n]$

(b)

**Figure 3. (a) Example loop (b) General $n$-level loop nest**

treated as a singly-nested loop with an iteration count of one. Multi-dimensional arrays are assumed to be stored in row-major format.

Consider the example loop shown in Figure 3(a). $B[i]$ is reused in different $j$-iterations, so it is moved up into the $i$-loop. $A[i][j], A[i][j-1]$, and $A[i][j+1]$ exhibit *group-spatial* reuse [19] – the cache line accessed by one reference will usually also contain the data for the others. $A[i-1][j]$ and $A[i+1][j]$ have *self-spatial* reuse [19] – each can reuse the cache line it accessed in the previous $j$-iteration. Finally, reference $C[j][i]$ has no reuse. The memory references are grouped into *reuse equivalence classes* – each class consists of memory references that exhibit self-spatial or group-spatial reuse. This classification is used in [19] to aid in loop transformation procedures, such as skewing, reordering, etc., by enabling a comparison of the reuse properties of different candidate transformations. We propose a refinement of the above procedure in order to utilize the reuse analysis for our line size selection problem.

#### 4.1.1 Refinement to self-spatial locality

In Figure 3(a) reference $B[i]$, after being moved to the $i$-loop, was assumed to have spatial reuse. However, this reuse can occur only if the cache line corresponding to $B[i]$ is still present in the cache when the next $i$-iteration begins, and has not been flushed out by the intervening accesses.

We generalize the above reuse condition for the example $n$-level nested loop of Figure 3(b). We assume that the loop bounds, and branch probabilities for conditionals are statically known, as is the usual case in many embedded applications. We use the following locality criterion: spatial locality for the reference $a[i_1][i_2] \dots [i_m]$ in the level-$m$ loop ($L_m$) can be exploited if the total number of memory accesses in inner loops (i.e., loops $L_{m+1} \dots L_n$) is less than the cache size. Let the number of memory references at loop level $j$ be $c_j$. For example, in Figure 3(b), $c_m = 1$ (for the single memory read), and $c_n = 2$ (for the two memory reads). The number of iterations $(r_j)$ of loop level $L_j$ is given by: $r_j = \left\lceil \frac{h_j - l_j + 1}{s_j} \right\rceil$. Thus, the sufficient condition for utilizing locality for the reference at nesting level $m$ is:

$$\sum_{i=m+1}^{n} c_i \left( \prod_{j=m+1}^{i} r_j \right) \leq \mathit{CacheSize} \qquad (2)$$

Note that the condition above, which involves the *number* of elements accessed, is an approximation of the cache behavior, for it assumes a fully associative cache with a perfect replacement policy. We incorporate the effect of cache conflicts that occur in limited associativity caches in Section 4.1.3.

#### 4.1.2 Refinement to group-spatial locality

In Figure 3(a), reference $A[i-1][j]$ and $A[i][j]$ are assumed to have no spatial or temporal locality because they are in different rows. However, this is a pessimistic assumption. If one complete row of $A$ fits in the cache, then the data read by $A[i][j]$ in one iteration can be reused by the $A[i-1][j]$ reference in the next $i$-iteration.

We formalize the above observation into a general condition for predicting reuse for the two $b$-references in Figure 3(b). We first determine a feasibility condition that needs to be met if the two references are to access the same cache line in different iterations. The sharing can occur only if all the index expressions in the higher dimensions $(1 \dots n-1)$ match exactly for different values of the loop index, and the reuse-dimension (lowest dimension) resolves to expressions that differ in less than the cache line size. For example, we require that $(i_1 + k_1)$ in one iteration, say $i_1 = I$, be equal to $(i_1 + k'_1)$ in some other iteration, say $i_1 = I + s_1 \cdot t$, for an integer $t$, since the two iteration numbers are separated by a multiple of the loop stride, $s_1$. We need to have: $I + k_1 + s_1 \cdot t = I + k'_1$, i.e., $k'_1 - k_1 = s_1 \cdot t$, i.e., $(k_1 - k'_1) \bmod s_1 = 0$.

Generalizing for dimensions $[1 \dots n-1]$, we have:

$$\forall j \in [1 \dots n-1], (k_j - k'_j) \bmod s_j = 0 \qquad (3)$$

For the reuse dimension (lowest dimension, indexed by $i_n$), we do not need an exact match, but only need that the expressions differ in less than the cache line size $L$. Thus, for two iterations: $i_n = I$ and $i_n = I + s_n \cdot t$, we need to have:

$$(I + k_n + s_n \cdot t) - (I + k'_n) \leq L \qquad (4)$$

i.e.,

$$k_n - k'_n - L \leq s_n \cdot t \quad (5)$$

To be applied as a feasibility test, this can be rephrased as:

$$\exists l \in [1 \ldots L], \text{ such that } (k_n - k'_n - l) \bmod s_n = 0 \quad (6)$$

Further, we need to ensure that the number of elements brought into the cache between the two accesses to $b$ in Figure 3(b), is less than the cache size. For the two index expressions $[i_1 + k_1]$ and $[i_1 + k'_1]$, the number of iterations of loop $L_1$ that elapse between the two expressions resolving to the same value is: $\left\lfloor \frac{|k_1 - k'_1|}{s_1} \right\rfloor$. Since $c_1$ elements are accessed at the first loop level, the number of elements from loop $L_1$ brought into the cache in the $\left\lfloor \frac{|k_1 - k'_1|}{s_1} \right\rfloor$ iterations is: $c_1 \cdot \left\lfloor \frac{|k_1 - k'_1|}{s_1} \right\rfloor$. The number of elements in inner loops accessed in each iteration of loop $L_1$ is: $\sum_{i=2}^{n} c_i (\prod_{j=2}^{i} r_j)$ (LHS of Equation (2), with $m = 1$). Thus, the total number of elements ($f_1$) brought into the cache before $[i_1 + k_1]$ and $[i_1 + k'_1]$ resolve to the same value, is given by:

$$f_1 = c_1 \cdot \left\lfloor \frac{|k_1 - k'_1|}{s_1} \right\rfloor \cdot \sum_{i=2}^{n} c_i \left( \prod_{j=2}^{i} r_j \right) \quad (7)$$

Summing over all the $n$ dimensions, we have the sufficient condition to enable group-spatial reuse as:

$$\sum_{t=1}^{n} f_t \leq \mathit{CacheSize} \quad (8)$$

i.e.,

$$\sum_{t=1}^{n} \left( c_t \cdot \left\lfloor \frac{|k_t - k'_t|}{s_t} \right\rfloor \cdot \sum_{i=t+1}^{n} c_i \left( \prod_{j=t+1}^{i} r_j \right) \right) \leq \mathit{CacheSize} \quad (9)$$

We conclude that the two $b$-references in Figure 3(b) exhibit spatial locality (i.e., fall into the same reuse equivalence class) if they satisfy the conditions 3, 6 and 9. The equations can be generalized to the case where the array index expressions are, of a more general form: $[a_j i_j + k_j]$ (where $a_j$ and $k_j$ are constants), as is the case with array references in many multimedia applications. The generalization is described in [11].

#### 4.1.3 Refinement incorporating cache conflicts

Note that Equations 2 and 9 are approximations for cache reuse, for they ignore the possibility of cache conflicts. We apply the technique in [12] to determine an estimate of the number of cache conflicts between memory references in different equivalence classes. Each conflict represents one off-chip memory access. We define *EstConflict*() to be a procedure that returns the number of processor cycles wasted due to conflicts. This is given by: *EstConflict*(j) = (# Conflicts predicted for accesses in loop $L_j$) $\times (K + L)$, where $K$ and $L$ are as defined in Equation (1).

#### 4.1.4 Computation of Processor Cycles

Based on the analysis in the previous sections, we determine the number of processor cycles, $T_{ji}$, due to each reuse equivalence class $R_{ji}$ in loop level $L_j$, as in the loop nest of Figure 3(b). Let the number of memory references in $R_{ji}$ be $|R_{ji}|$. For the $r_j$ iterations of loop $L_j$, the total number of memory accesses for class $R_{ji}$ is: $r_j \cdot |R_{ji}|$, of which $\frac{r_j \cdot |R_{ji}|}{L}$ are cache misses, and the rest $(r_j \cdot |R_{ji}| - \frac{r_j \cdot |R_{ji}|}{L})$ are hits. Since a cache hit costs 1 cycle and a miss $(K + L)$ cycles, we have:

$$\begin{aligned} T_{ji} &= \frac{r_j \cdot |R_{ji}|}{L}(K + L) + r_j \cdot |R_{ji}| - \frac{r_j \cdot |R_{ji}|}{L} \\ &= r_j \cdot |R_{ji}| \cdot \left( \frac{K-1}{L} + 2 \right) \end{aligned} \quad (10)$$

For the set $E$ of all the equivalence classes at level $j$, we have, the total cycles, $T_j$ given by: $T_j = \sum_i T_{ij}$.

The number of accesses with no reuse at loop level $L_j$ is given by: $c_j - \sum_i |R_{ji}|$. The number of cycles, $Q_j$, spent in accessing elements with no reuse is given by:

$$Q_j = (K + L) \cdot \left( c_j - \sum_i |R_{ji}| \right) \quad (11)$$

Finally, the number of processor cycles, $P_j$, wasted due to cache conflicts for accesses in loop $L_j$ is given by:

$$P_j = r_j \cdot \mathit{EstConflict}(j) \quad (12)$$

Thus, the total time, spent accessing memory data at level $L_j$ is given by: $(T_j + Q_j + P_j)$. For the entire loop nest, we multiply the total cycles at each loop level by the number of times the loop $L_j$ is executed. Thus, the total cycles, $C$ for the loop nest under consideration is given by:

$$C = \sum_{j=1}^{n} \left( (T_j + Q_j + P_j) \cdot \prod_{i=1}^{j-1} r_j \right) \quad (13)$$

To determine the number of cycles required for all memory accesses in the program, we take the cumulative estimate over all loop nests. We use this estimate in the memory performance estimation step in algorithm *MemExplore*.

## 5. Experiments and Results

We present some exploration results on sample application routines from the image processing and digital signal

processing domain. The benchmark examples are: *Histogram* – a histogram evaluation routine, commonly used in image enhancement algorithms [4]; *Lowpass* – another image processing algorithm used for accentuating low frequencies in an image, so that the resulting image has lower changes between neighboring pixel values [9]; and *Beamformer* – a radar application, involving the summation of digitized signals from an antenna array [9]. We present below a comparison of the estimated memory cycles for the benchmarks with the actual memory cycles. The estimated cycles are determined by the computations in Section 4. The actual cycles are measured by a memory simulator we developed, which takes as input a stream of memory addresses generated during execution of a benchmark, and reports the number of accesses to off-chip memory by simulating the data cache and Scratch-pad memory. For our experiments, we used a penalty of 10 processor cycles for off-chip memory accesses (i.e., $K = 10$ in Equations 1 and 11).

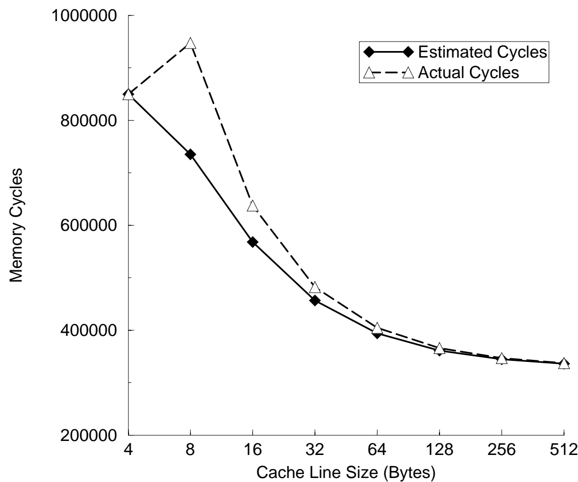



**Figure 4. Variation of memory performance with line size for fixed cache size of 1 KB (*Histogram* Benchmark)**

Our first experiment focusses on loop $L_3$ of the exploration algorithm *MemExplore*, where we study variation of the memory performance with the cache line size, for a given cache size. Figure 4 shows a comparison between the actual simulation result and the estimated number of processor cycles for the memory accesses in the *Histogram* routine, for a fixed cache size of 1 KB. We note that the estimated performance very closely follows the actual performance, except for one line size (8 bytes). The best cache line size selected by our algorithm (= 512 bytes), is also the best line size parameter determined from the simulation.

Figure 5 illustrates another slice of the exploration space on the same benchmark, where the estimated memory per-

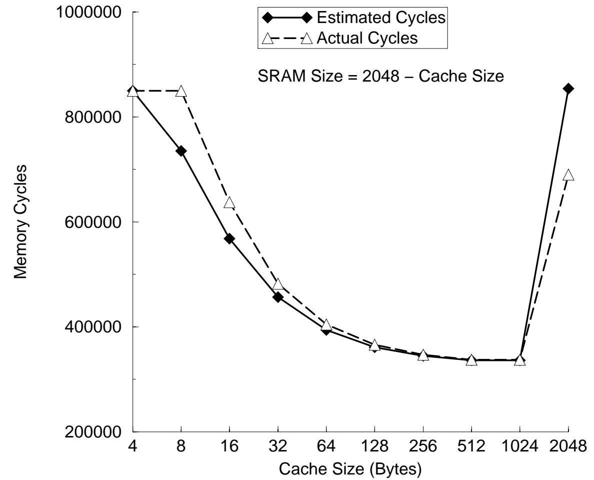



**Figure 5. Variation of memory performance with different mixes of cache and Scratch-pad memory, for total on-chip memory of 2 KB (*Histogram* Benchmark)**

formance for different divisions of a fixed total on-chip memory space of 2 KB into data cache and Scratch-pad memory, is compared against the simulated performance. The memory cycles plotted correspond to one iteration of the outer $L_2$ loop of *MemExplore*, where the best cache line size (from loop $L_3$) is selected for each candidate cache size. For each selected cache size, the corresponding Scratch-pad SRAM size is given by: 2 KB $-$ cache size. The points on the left and right extremes represent divisions incurring severe cache conflicts (for cache size = 2048 Bytes, the SRAM size is 0, causing unavoidable conflicts in the cache). The estimation process gives the best division of the 2 KB space as: 1 KB cache + 1 KB Scratch-pad memory – this selection is validated with the actual simulation results, as shown in Figure 5.

Figure 6 shows the variation of the memory performance with the total on-chip memory space for the *Histogram* example. The $y$-axis shows the best performance obtained by any architecture (i.e., division into Scratch-pad memory and cache, as well as selection of cache line size) for a given total on-chip memory space, i.e., one iteration of the outer loop $L_1$ in *MemExplore*. For a given application, the variation of the memory performance with the total on-chip memory is generated as feedback to the designer. The designer can then select an appropriate total on-chip memory size, based on the value beyond which no significant improvement is predicted. In the example of Figure 6, the total size of 2 KB is a good selection, as we observe very little improvement in cycle time beyond this cache size. Figures 7(a) and (b) show the results of the exploration for the *Lowpass* and

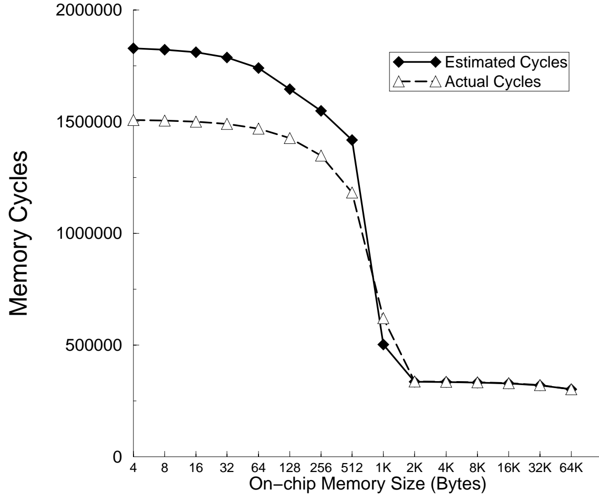
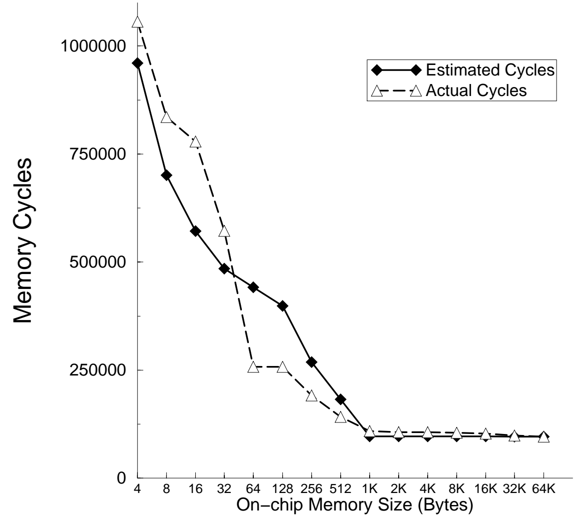
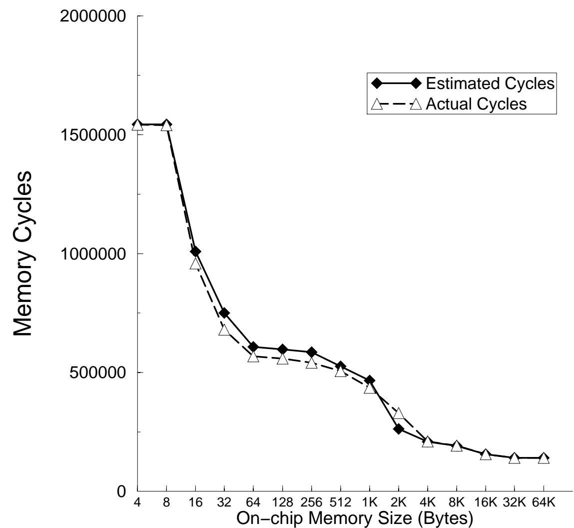



**Figure 6. Variation of memory performance with total on-chip memory space (*Histogram* Benchmark)**




(a)




(b)

**Figure 7. Memory exploration for (a) *Lowpass* and (b) *Beamformer* benchmarks**

*Beamformer* examples. We notice that the estimated performance curve follows the actual performance very closely. This curve can help the designer select the optimal on-chip memory size. Once the designer chooses an appropriate total memory size based on the estimation curve, the best memory architecture (consisting of the division of this space into Scratch-pad memory and cache, as well as cache line size) is automatically chosen. Note that the each point in the exploration spaces of Figures 6 and 7 corresponds to the best architecture found for the given total on-chip memory space.

The most important advantage of our analytical technique for exploring the memory performance of embedded applications is that candidate architectures can be rapidly evaluated for their memory performance. The estimation-based exploration for our experiments above required only a few seconds, which was about 1000 times faster than the simulation of the memory performance for the same set of architectures explored. This estimation capability is very important in the initial stages of system design, where the number of possible architectures is too many, and a simulation of each architecture is prohibitively expensive.

## 6. Conclusions

The management of local memory space is an important problem in embedded systems involving large behavioral arrays, all of which cannot be stored on-chip. The ideal local memory architecture, consisting of a judicious division of the memory into Scratch-pad SRAM (software-controlled) and data cache (hardware-controlled), as well as the cache

line size, depends on the characteristics of the specific application.

We presented a strategy for exploration of on-chip memory architecture for embedded applications, based on an estimation of the memory performance. Our experiments on benchmark routines show that the estimated performance matches the actual simulated performance very closely, and is three orders of magnitude faster. Thus, the exploration technique is very useful during the early stages of system design, when a large number of different possible memory architectures need to be evaluated for their performance.

The future work for this research includes the incorporation of other cache features, such as write policy, and the incorporation of a more accurate cost function for on-chip memory area that accounts for the memory decoder area.

## References


[1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[2] P. Baglietto, M. Maresca, M. Migliardi, and N. Zingirian. Image processing on high performance RISC systems. Technical Report TR SM-IMP/DIST/08, University of Genoa, December 1995.

[3] F. Balasa, F. Catthoor, and H. D. Man. Dataflow-driven memory allocation for multi-dimensional signal processing systems. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, November 1994.

[4] R. C. Gonzalez and P. Wintz. *Digital Image Processing*. Addison-Wesley, 1987.

[5] P. K. Jha and N. Dutt. Library mapping for memories. In *European Design and Test Conference*, pages 288–292, March 1997.

[6] D. Karchmer and J. Rose. Definition and solution of the memory packing problem for field-programmable systems. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 20–26, November 1994.

[7] P. E. R. Lippens, J. L. van Meerbergen, W. F. J. Verbaeghand, and A. van der Werf. Allocation of multiport memories for hierarchical data streams. In *Proceedings of the IEEE International Conference on Computer Aided Design*, November 1993.

[8] LSI Logic Corporation, Milpitas. *CW33000 MIPS Embedded Processor User's Manual*, 1992.

[9] P. R. Panda and N. D. Dutt. 1995 High level synthesis design repository. In *International Symposium on System Synthesis*, September 1995.

[10] P. R. Panda, N. D. Dutt, and A. Nicolau. Memory organization for improved data cache performance in embedded processors. In *International Symposium on System Synthesis*, pages 90–95, November 1996.

[11] P. R. Panda, N. D. Dutt, and A. Nicolau. Architectural exploration and optimization of local memory in embedded systems. Technical Report ICS-TR-97-31, University of California, Irvine, June 1997.

[12] P. R. Panda, N. D. Dutt, and A. Nicolau. Data cache sizing for embedded processor applications. Technical Report ICS-TR-97-30, University of California, Irvine, June 1997.

[13] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of Scratch-pad memory in embedded processor applications. In *European Design and Test Conference*, March 1997.

[14] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design – The Hardware/Software Interface*. Morgan Kaufman, 1994.

[15] L. Ramachandran, D. Gajski, and V. Chaiyakul. An algorithm for array variable clustering. In *European Design and Test Conference*, February 1994.

[16] Standard Performance Evaluation Corporation, Fairfax. *SPEC Newsletter*, December 1991.

[17] L. Stok and J. A. G. Jess. Foreground memory management in data path synthesis. *International Journal of Circuit Theory and Applications*, 20(3):235–255, 1992.

[18] R. Wilson. Graphics IC vendors take a shot at embedded DRAM. *Electronic Engineering Times*, (938):41,57, January 27 1997.

[19] 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.