Data Memory Organization and Optimizations in Application-Specific Systems

Preeti Ranjan Panda
Synopsys Inc.

Nikil D. Dutt and Alexandru Nicolau
University of California, Irvine

Francky Catthoor, Arnout Vandecappelle, Erik Brockmeyer, Chidamber Kulkarni, and Eddy De Greef
IMEC

In application-specific designs, customized memory organization expands the search space for cost-optimized solutions. Several optimization strategies can be applied to embedded systems with several different memory architectures: data cache, scratch-pad memory, custom memory architectures, and dynamic random-access memory (DRAM).

System-level designers and electronic design automation (EDA) researchers have paid considerable attention to data memory issues because of the memory subsystem's significance in determining such important design parameters as area, power, and performance. Designers have studied different approaches for the memory subsystem, ranging from the standard processor-memory hierarchy to fully customized memory architectures targeted at a given application. In the context of application-specific design, interesting new memory-organization optimization possibilities arise, no matter what the selected architecture. In addition, the choice of a suitable memory architecture itself forms an important exploration phase that can also be partially automated.

We discuss the associated optimization and exploration techniques based on the memory modules and architectures for application-specific systems surveyed in "Random Access Data Storage Components in Customized Architectures," by L. Nachtergaele, F. Catthoor, and C. Kulkarni (in this issue). These optimizations depend on the memory architecture parameters and are thus memory-platform dependent. Most of these steps heavily benefit from the up-front application of the source-to-source code transformations discussed in "Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors," by F. Catthoor and his colleaguess (also in this issue).

Memory data layout

The more familiar memory hierarchy configuration features a processor core and one or more data cache levels.[1] In this model, most optimizations performed by traditional compilers are applicable. However, the advance knowledge of the actual application to be executed on the system lets us perform more

0740-7475/01/$10.00 © 2001 IEEE

aggressive optimizations.[2,3] An example is data layout optimization, so named because the entire application is statically known, and this knowledge lets us more intelligently place data structures in memory to improve the memory performance. Compilers typically don't perform such optimizations, because, for instance, they cannot assume that the translation unit under compilation represents the entire program. Decisions on the best placement of data cannot be made, because routines in a separate translation unit (a different source file not yet compiled) might access the same data in a completely different pattern, invalidating the previous analysis. However, in application-specific design, we can reasonably assume that the entire application is available to us, so by analyzing the data access patterns, we can make intelligent data-placement decisions.

Consider a direct-mapped cache of size C words, where $C = 2^M$, with an M-word cache line size (that is, M consecutive words are fetched from memory on a cache-read miss), and a write-through cache with a fetch-on-miss policy.[1] In this article, we address the layout of array data because it typically occupies most of the data space, and array accesses constitute most of the data memory accesses. Scalar data layout is addressed elsewhere.[2]

Suppose the code fragment in Figure 1a executes on a processor with the above cache configuration, where N is an exact power of 2, and $N > C$. Assuming that a single array element occupies one memory word, let array a begin at memory location 0, b at N, and c at $2N$. In a direct-mapped cache, the cache line that would contain a word located at memory address A is given by $(A \bmod C)/M$. In the above example, array element $a[i]$ would be located at memory address i. Similarly, we have $b[i]$ at $N + i$ and $c[i]$ at $2N + i$. Because N is a multiple of C, all of $a[i]$, $b[i]$, and $c[i]$ will map into the same cache line, as Figure 1b shows. Consequently, every data access results in a cache miss.

Such memory access patterns result in extremely inefficient cache utilization, especially because many applications deal with arrays whose dimensions are an exact power of two. The cache misses lead to inferior designs, in terms of both performance and energy consumption. In such situations, increasing the cache size is not an efficient solution, because lack of capacity does not cause the cache misses. There is only one active cache line during one loop iteration. The conflict misses can be avoided if the cache size C is made greater than N, but increased cache size incurs an associated penalty in area and access time. Reorganizing the data in memory is a more elegant solution that also keeps the cache size relatively small.

One way to prevent the thrashing caused by excessive cache conflicts in the above example is to introduce M dummy memory words between two consecutive arrays that are accessed in an identical pattern in the loops. In

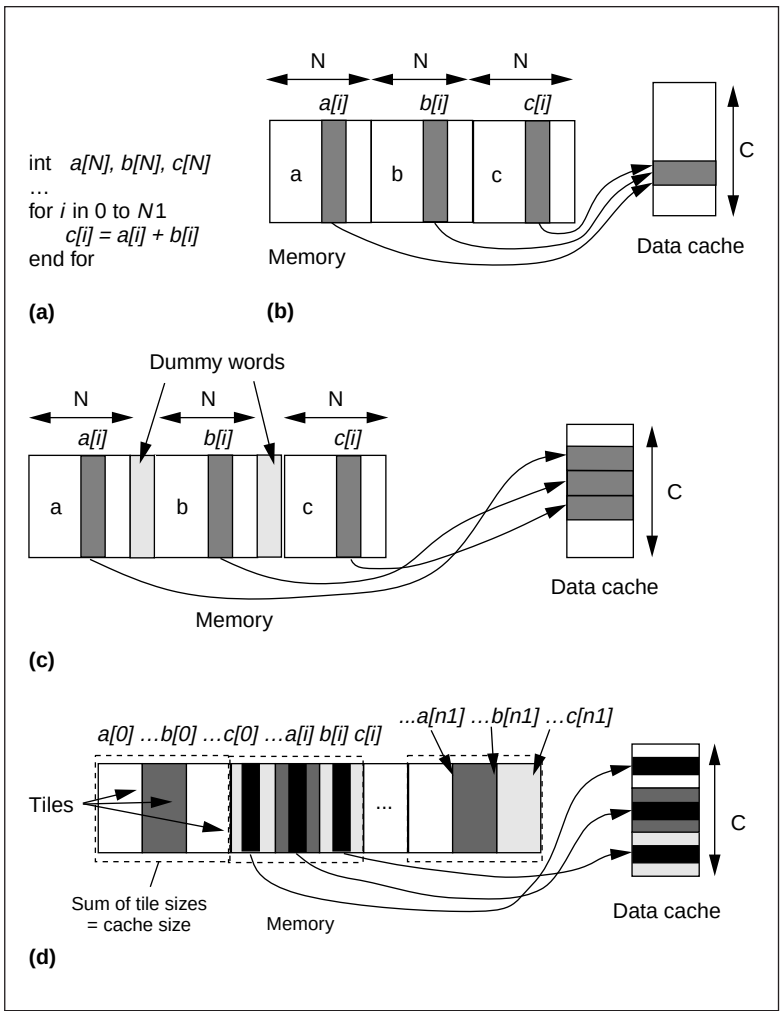


Figure 1. A sample code fragment (a); cache conflicts: *a*[*i*]; *b*[*i*], and *c*[*i*] map into the same cache line (b); data layout avoids cache conflicts—insertion of dummy words between arrays (c); and alternate data layout—tiles interleaved in memory (d).

other words, array a begins at 0, array b begins at location $N + M$ (instead of N), and array c begins at $2N + 2M$ (instead of $2N$), as Figure 1c shows. This will ensure that $a[i]$, $b[i]$, and $c[i]$ are always mapped into different cache lines, and that their accesses do not interfere with each other in the data cache. The cache size can still be small while maintaining efficient access.[2]

A different data layout approach to avoiding the cache conflict problem is to split the initial arrays into subarrays (or tiles) of equal size, as Figure 1d shows. The tile size is chosen such that the sum of the tile sizes of all arrays involved in a loop does not exceed the data cache size. The tiles are then merged in an interleaved fashion to form one large array. Because tiles of different arrays accessed in the same loop iteration are adjacent in the data layout, tile data never conflicts in the cache.[4]

In a more complex application with several loop nests, both approaches outlined above would need to be generalized to handle different access patterns in different loops. The generalizations are discussed elsewhere.[2,4] The subarrays need not be of equal size, and this extension leads to even better results.[4]

Scratch pad memory

Designers are not necessarily restricted to the single memory architecture just discussed. Because the design must execute only a single application, we can try some unconventional architectural variations that suit the specific application under consideration, such as scratch-pad memory.[5]

Problem formulation and illustration

Scratch-pad memory refers to data memory residing on chip—that is, mapped into an address space disjoint from the off-chip memory but connected to the same address and data buses. Both the cache and scratch-pad memory (usually SRAM) allow fast access to their residing data, whereas access to the off-chip memory (usually DRAM) requires relatively longer access times. The main difference between the scratch-pad SRAM and data cache is that the SRAM guarantees a single-cycle access time, whereas access to the cache is subject to cache misses.

Scratch-pad memory is an important architectural consideration in modern embedded systems, where advances in embedded DRAM technology have made it possible to combine DRAM and ordinary logic on the same chip. Data stored in embedded DRAM can be accessed far faster than in off-chip DRAM. A related optimization problem arising in this context is how to identify critical data in an application, for storage in on-chip memory.

Figure 2 shows the data address space mapping for a sample addressable memory of size N data words. Memory addresses $0 + (P - 1)$ map into the on-chip scratch-pad memory and have a single processor cycle access time. Memory addresses $P + (N - 1)$ map into the off-chip DRAM, and the CPU accesses them through the data cache. A cache hit for an address in the range $P + N - 1$ results in a single-cycle delay, whereas a cache miss, which leads to a block transfer between off-chip and cache memory, might result in a delay of, say, 10 to 20 processor cycles.

The example in Figure 3 shows the merit of such an architectural variation. Procedure CONV is the code kernel of the convolution routine commonly used in imaging applications.

A small 4×4 matrix of coefficients (*mask*) slides over the input image (*source*), covering a different 4×4 region in each iteration of y, as Figure 3 shows. In each iteration, the mask coefficients combine with the region of the image currently covered, to obtain a weighted average. The result (*acc*) is assigned to the pixel of the output array (*dest*) in the center of the covered region.

We can avoid cache conflicts from accesses to the dest array by using a write-through cache with write-around (that is, no write allocate) policy, where memory writes do not interfere with the cache in case of misses. However, if the two source and mask arrays were to be accessed through the data cache, cache conflicts would affect the performance. Furthermore, an associative cache, by itself, will not generally eliminate the problem, because most practical caches have a limited associativity. Typically, the situation might require an associativity as large as the number of conflicting arrays.

We can solve the conflict problem by stor-

ing 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 regular memory accesses to source. Because the mask array is stored on chip, we have ensured that the frequently accessed data is never ejected off chip, thereby significantly improving the memory access performance and energy dissipation.[5,6]

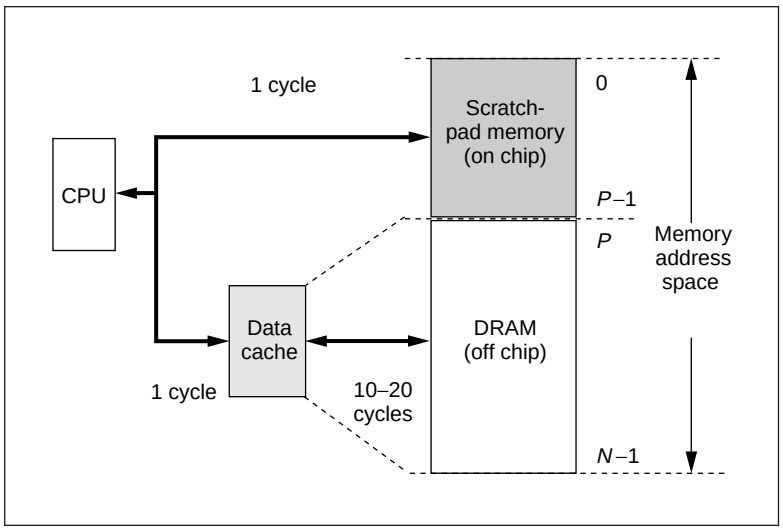


Figure 2. Division of data address space between scratch-pad memory and off-chip memory.

Scratch-pad architecture exploration

Several different memory architectures could be devised to efficiently exploit different application-specific memory access patterns. Even if we restrict the architecture's scope to those involving only on-chip memory, the exploration space of possible configurations is too large, making it infeasible to exhaustively simulate the application's performance and energy characteristics for each configuration. Thus, we need exploration tools for rapidly evaluating the effect of several candidate architectures. Such tools can greatly assist a system designer by giving fast initial feedback on a wide range of memory architectures.

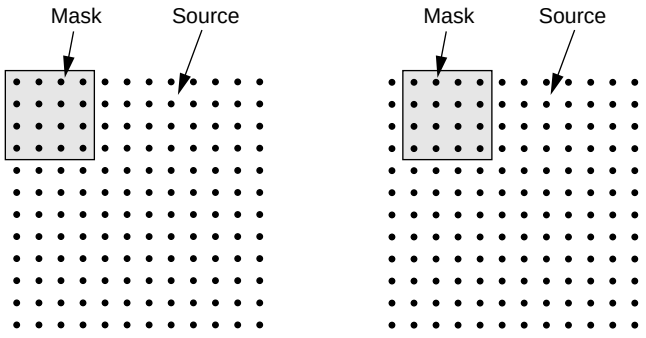
We illustrate a memory exploration strategy by restricting the design space as follows: Given a certain amount of on-chip memory space, partition it appropriately into data cache and scratch-pad memory to minimize the total access time and energy dissipation, thus minimizing the number of accesses to off-chip memory.

In our formulation, an on-chip memory architecture is defined as a combination of

- the total size of on-chip memory used for data storage, and
- the partitioning of this on-chip memory into scratch-pad SRAM, characterized by its size; and data cache, characterized by the cache size and the cache line size.

```
# define N 128
# define M 4
# define NORM 16
int source[N][N], dest [N][N];
int mask [M][M];
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;
}
```

(a)



(b)

Figure 3. Procedure CONV (a); memory access pattern in CONV (b). Procedure CONV is the code kernel of the convolution routine commonly used in imaging applications.

```
Algorithm MemExplore
L1: for on-chip memory size T (in powers of 2)
    L2: for cache size C (in powers of 2, < T )
        SRAM Size S = T – C
        DataPartition(S)
        L3: for line size L (in powers of 2, < C; < MaxLine)
            Estimate Memory Performance
    Select (C; L) that maximizes performance
```

Figure 4. Memory architecture exploration.

Figure 4 summarizes the basic algorithm for memory architecture exploration.[6]

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

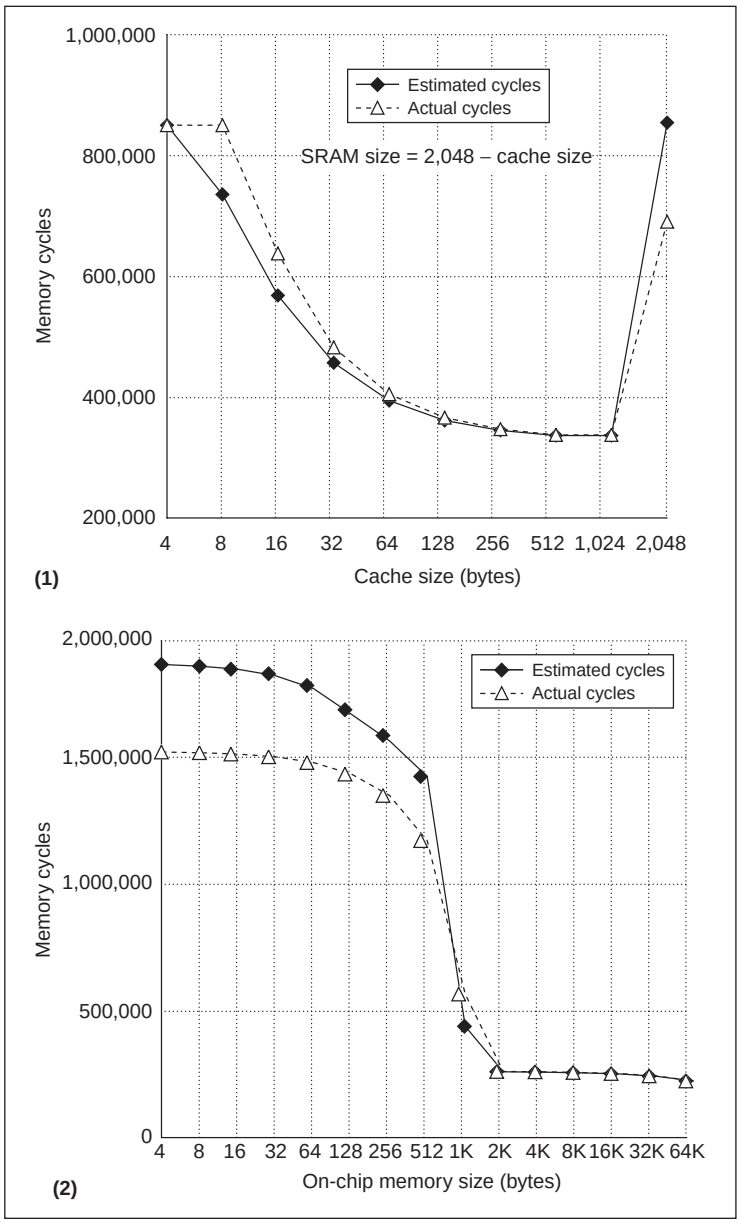


Figure A. Histogram example: variation of memory performance with different mixes of cache and scratch-pad memory, for total on-chip memory of 2 Kbytes (1); variation of memory performance with total on-chip memory space (2).

Scratch pad exploration

Figure A1 illustrates a slice of the memory exploration technique on the Histogram example, comparing the estimated memory performance for different divisions of a fixed total on-chip memory space of 2 Kbytes into data cache and scratch-pad memory against the simulated performance. The Histogram[1] routine reads every pixel in an image and constructs a histogram of the pixels' brightness levels. The memory cycles plotted correspond to one iteration of the outer L2 loop of MemExplore, where the best cache line size (from loop L3) is selected for each candidate cache size. For each selected cache size, the corresponding scratch-pad SRAM size is given by a 2-Kbyte cache size. The points on the left and right extremes represent the divisions' incurring severe cache conflicts (when the cache size is 2,048 bytes, the SRAM size is 0, causing unavoidable conflicts in the cache). The estimation process gives the best division of the 2-Kbyte space as a 1-Kbyte cache plus a 1-Kbyte scratch-pad memory. This selection is validated with the actual simulation results, as Figure A1 shows.

Figure A2 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 (divided into scratch-pad memory and cache, as well as selected cache line size) for a given total on-chip memory space (one iteration of the outer loop L1 in MemExplore.) For a given application, the variation of the memory performance with the total on-chip memory is

technique for partitioning program variables into scratch-pad memory and cache.[5] Scalar and array data, identified as most critical, are assigned to the SRAM, on the basis of data size, memory access frequency, and the possibility of cache conflicts. For the data assigned for storage in off-chip memory (and accessed through the cache), we estimate the memory access performance by combining an analysis of both the array access patterns in the application and an approximate model of the cache behavior. See the "Scratch pad exploration" sidebar for an illustration.

The estimation gives us the expected number of processor cycles required for all the memory accesses in the application. For each T, we select the (C, L) pair estimated to maximize performance. Finally, we assign the variables' memory address using the cache and SRAM parameters selected via the algorithms.[5] Shieu et al. discuss an extension of the architectural exploration to include energy consumption.[7]

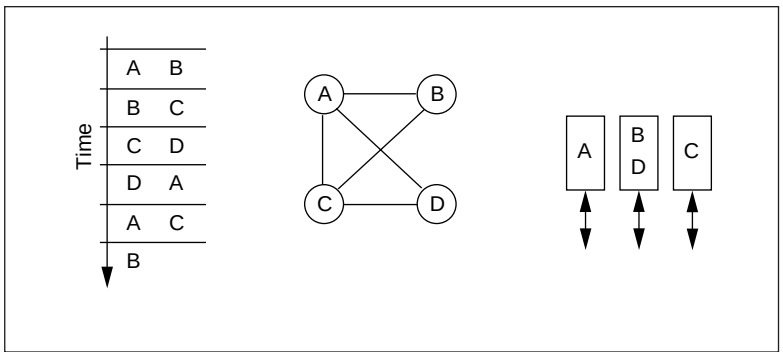


Figure 5. A conflict graph corresponding to ordered memory accesses and a possible memory organization obeying the constraints.

Custom memory architecture exploration

In many cases, a fully customized memory architecture can yield performance and power characteristics superior to a cache-based architecture—for example, when significant data parallelism is available.

A custom memory organization can potentially and significantly reduce the system cost. However, achieving these cost reductions, especially manually, is not trivial. Designing a custom memory architecture means deciding how many memories to use and of which type (single-port, dual-port, and so on). Additionally, the memory accesses must be ordered in time to meet the real-time constraints (the cycle budget). Finally, each array must be assigned to a memory, so that arrays can be accessed in parallel as required to meet the real-time constraints.[8]

Memory access ordering and conflict graphs

The memory bandwidth requirement can be modeled as a conflict graph derived from memory access ordering, as Figure 5 shows. Its nodes represent the application's arrays, and its edges connect arrays accessed in parallel. The graph models the constraints on the memory archi-

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 Figure A2, the total size of 2 Kbytes is a good selection, as we observe very little improvement in cycle time beyond this cache size.

The most important advantage of the exploration strategy is that candidate architectures can be very rapidly evaluated for their memory performance. The estimation-based exploration requires only a few seconds, which was about three orders of magnitude times faster than the simulation of the memory performance for the same set of architectures explored. This estimation capability is vital in the initial stages of system design, in which the number of possible architectures is too many and a simulation of each architecture prohibitively expensive.

Reference

1. P.R. Panda, N.D. Dutt, and A. Nicolau, "On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems," *ACM Trans. Design Automation of Electronic Systems*, vol. 5, no. 3, July 2000, pp. 682-704.

tecture. In the example in Figure 5, array A cannot be stored in the same single-port memory as any other array; it conflicts with them. However, there is no edge between B and D, so these two arrays can be stored in the same memory. To support multiport memories, this model must be extended with hyperedges.[8,9]

From the conflict graph, we can estimate the cost of the memory bandwidth required by the specified ordering of the memory accesses. Three main components contribute to cost:

- Two accesses to the same array occurring in parallel indicates a self-conflict, a loop in the conflict graph. Such a self-conflict forces a two-ported memory in the memory architecture. Two- and multiported memories are far more costly; therefore, self-conflicts carry a heavy weight in the cost function.
- In the absence of self-conflicts, the conflict graph's chromatic number indicates the minimal number of single-port memories required to provide the necessary bandwidth. For example, in Figure 5, the graph's chromatic number is three: for example, A, B, and C need three different colors to color the graph. That also means a minimum of three single-port memories is required, even though three accesses never occur in parallel. More memories make the design potentially more costly; therefore, the conflict graph's chromatic number is the second cost component.
- Finally, the more conflicts in the conflict graph, the more costly the graph is. Indeed, every conflict subtracts some freedom for assigning arrays to memories. In addition, not every conflict carries the same weight. For instance, a conflict between a very small, heavily accessed array and a very large, lightly accessed array is not so costly, because it is energy efficient to divide these two over different memories. A conflict between two arrays of similar size, however, is fairly costly because the two cannot be stored in the same memory, and potentially must be combined with other arrays, which do not match very well.

This abstract cost model lets us evaluate and explore the ordering of memory accesses without yet detailing a memory architecture. Indeed, the latter is in itself a complex task, so combining the two is not feasible. The conflict graph is heavily influenced by the access ordering, and tool support is crucial to perform this tedious, error-prone task. Wuytack et al. have described efficient techniques based on this approach.[9] Another approach focuses on periodic streams.[10]

Memory allocation and assignment

In custom memory architecture, the designer can choose memory parameters such as the number, size, and number of ports. This decision, which considers the constraints derived above, is the focus of the memory allocation and assignment step. We can subdivide the problem into two subproblems. First, memories must be allocated. Several memories are chosen from the available memory types and the different port configurations; possibly, different types are intermixed, and some memories might be multiported. The dimensions of the memories are, however, determined only in the second stage. When arrays are assigned to memories, their sizes can be added up and the maximal bit width taken to determine the memory's required size and bit width. With this decision, the memory organization is fully determined. Our assumption for this discussion is an optimal assignment.

Allocating more or fewer memories affects the chip area and the memory architecture's energy consumption. Large memories consume more energy per access than small memories, because of the longer word and bit lines. Therefore, the energy consumed by a single large memory containing all the data is far larger than when the data is distributed over several smaller memories. Also, the area of the one-memory solution is often higher when different arrays have different bit widths. For example, when a 6-bit and an 8-bit array are stored in the same memory, two bits are unused for every 6-bit word. Storing the arrays in different memories, one 6 bits wide and the other 8 bits wide, can avoid this overhead.

At the other end of the spectrum is storing all the arrays in different memories. This also leads to relatively high energy consumption, because the external global interconnection lines con-

necting all these (small) memories with each other and with the data paths become a significant energy drain. Likewise, the area occupied by the memory system expands because of the interconnections, fixed address decoding, and other overhead per memory.

The interesting memory allocations lie somewhere between these two extremes. The area and the energy function reach a minimum somewhere, but at different points. The useful exploration region to trade off area with energy consumption lies between the two minima, as Figure 6 shows. The chromatic number of the ECG and the number of signals bound the useful range. The area curve contains a minimum between these extremes, due to bit waste (relating to the memory space "lost" by a mismatch between the required variable size and the allocated memory width) and the peripheral overhead. The energy also exhibits a similar minimum but at larger memory counts because of the overhead either of too-large memories (storing several data and hence requiring longer internal lines) or of the inter-memory interconnect.

The cost of memory organization depends on memory allocation and the assignment of arrays to the memories. When several memories are available, we have many ways to assign arrays to them. In addition to the conflict cost mentioned earlier, the optimal assignment of arrays to memories depends on the memories used. For example, the energy consumption of some memories is very sensitive to their size; for others, it is not. In the former case, it might be advantageous to accept some wasted bits to keep the heavily accessed memories very small; in the latter, the reverse might be true. To find a near-optimal signal-to-memory assignment, we must consider many possibilities, and an indispensable tool for this step is one based on a good memory library. Although early work in this area focused on stream-based applications,[11] general-purpose CAD techniques have more recently been proposed to implement this assignment based on the ECG.[8]

Both the memory allocation and the signal-to-memory assignment must consider the constraints generated to meet the real-time requirements. Memory allocation requires a certain minimum number of memories. This minimum is derived from the conflict graph's chromatic number. For the signal-to-memory assignment, conflicting arrays cannot be assigned to the same memory. When there is a conflict between two arrays, which in the optimal assignment are stored in the same memory, we must make a less-efficient assignment. Thus, more conflicts make the memory organization potentially more expensive.

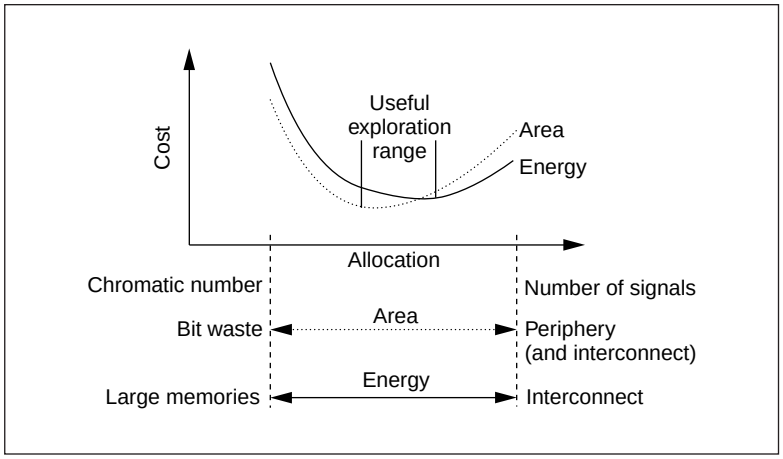


Figure 6. Memory allocation exploration.

Reducing memory size requirements

One important memory-related optimization in system design is the reduction in the data-memory size requirements for an application. We can sometimes affect this optimization by reducing the actual allocated space for temporary arrays. We accomplish this reduction by mapping, in place, different sections of the logical array into the same physical memory space when the lifetimes of these sections are nonoverlapping.[8]

To illustrate in-place mapping, an example routine performs autocorrelation in a linear prediction coding vocoder. Figure 7 (next page) shows the initial algorithm. Two signals are responsible for most of the memory accesses and are dominant in size, namely ac_inter[] and Hamwind[], consisting of 26,400 and 2,400 integer elements. The loop nest has nonrectangular access patterns dominated by accesses to the temporary variable ac_inter[]. So, reducing power requires reducing the number

```
for(i6=0;i6<11;i6++) {
    ac_inter[i6][i6] = Hamwind[O] * Hamwind[i6];
    ac_inter[i6][i6+1] = Hamwind[l] * Hamwind[i6+1];
    for(i7=(i6+2);i7<2400;i7++)
        ac_inter[i6][i7] = ac_inter[i6][i7–1] +
            ac_inter[i6][i7–2] + (Hamwind[i7–i6] * Hamwind[i7]);
    AutoCorr[i6] = ac_inter[i6][23991];
}

for(i8=0;i8<10;i8++) {
    v[0][i8] = AutoCorr[i8+1];
    u[O][i8] = AutoCorr[i8];
}
```

Figure 7. Initial algorithm for autocorrelation in a linear prediction coding vocoder.

```
for(i6=0;i6<11;i6++) {
    ac_inter[i6][i6%3] = Hamwind[O] * Hamwind[i6];
    ac_inter[i6][(i6+1)%3] = Hamwind[1] * Hamwind[i6];
    for(i7=(i6+2);i7*<2400;i7++)
        ac_inter[i6][i7%3] = ac_inter[i6][(i7–1)%3] +
            ac_inter[i6][(i7–2)%31 + (Hamwind[i7–i6] * Hamwind[i7]);
}

for(i8=0;i8<10;i8++) {
    v[O][i8] = ac_inter[i8+][2]; /* 2399 % 3 = 2 */
    u[O][i8] = ac_inter[i8][2];
}
```

Figure 8. Intrasignal and intersignal in-place data mapping to reduce required storage size.

of memory accesses to ac_inter[]. This is possible only by first reducing the size of ac_inter[] and placing this signal in a local memory.

Because ac_inter[] depends only on two of its earlier values, only three (earlier) integer values need be stored for computing each autocorrelated value. Thus, performing intrasignal in-place data mapping, as shown in Figure 8 can dramatically reduce the signal's size from 26,400 to 33 integer elements.

AutoCorr[] is a temporary signal. Reusing the memory space of ac_inter[] for storing AutoCorr[] can further reduce the total required memory space. We achieve this by intersignal in-place mapping of array AutoCorr[] on ac_inter[]. Thus, initially, because of the signal's large size, ac_inter[] could not be accommodated in the on-chip local memory, but we've eliminated this problem, achieving reductions in both memory size and associated power consumption.

To explore the many in-place mapping opportunities here required CAD techniques (see the summary of early IMEC work[8] and also more recent work[12]). The development of high-level in-place estimates for steering the (platform-independent) system-level code transformations has also been addressed.[13]

Synthesis with DRAMs

Applications involving large amounts of data traditionally needed to store the data in off-chip DRAM because DRAMs required a different fabrication process. However, embedded DRAM technology lets you integrate DRAM blocks on the same chip as the rest of the application, freeing you from the constraints of a fixed DRAM architecture and protocol. If the DRAM is embedded on chip, then the pin count constraints that limit the number of address and data buses on the DRAM disappear. Thus, parallel access to the core storage is now possible, thereby significantly improving memory access performance. Parallel access also significantly reduces power consumption because memory accesses now involve on-chip data, instead of off-chip bus, transfers. The DRAM's internal architecture is now visible to the application, which can use the knowledge in ways not previously possible.

The DRAM memory address is internally split into a row address consisting of the most significant bits and a column address consisting of the least significant bits. The row address selects a page from the core storage; the column address selects an offset within the page to arrive at the desired word. When an address is presented to the memory during a read operation, the entire page addressed by the row address is read into the page buffer, in anticipation of spatial locality. If future accesses are made to the same page, there's no need to access the main storage area, because it can just be read off the page buffer, which acts like a cache. Thus, subsequent accesses to the same page are very fast. Incorporating this knowledge into an automatic synthesis tool requires modifying traditional synthesis models.[2,14] (See the "Modeling of DRAM access for synthesis" sidebar for a discussion of related modeling issues.)

The presence of embedded DRAMs adds several dimensions to traditional architecture

Modeling of DRAM access for synthesis

Figure B1 shows a simplified timing diagram of the read cycle of the 1M 64-bit IBM11T1645LP extended data-out (EDO) DRAM. The memory read cycle is initiated by the falling edge of the row address strobe (RAS) signal, at which time the row address is latched from the address bus. The column address is latched at the falling edge of the column address strobe (CAS) signal, which should occur at least T_{ras} = 45 ns later. Following this, the data is available on the data bus after T_{cas} = 15 ns. Finally, the RAS signal is held high for at least T_p = 45 ns to allow for bit-line precharge, which is necessary before the next memory cycle can be initiated.

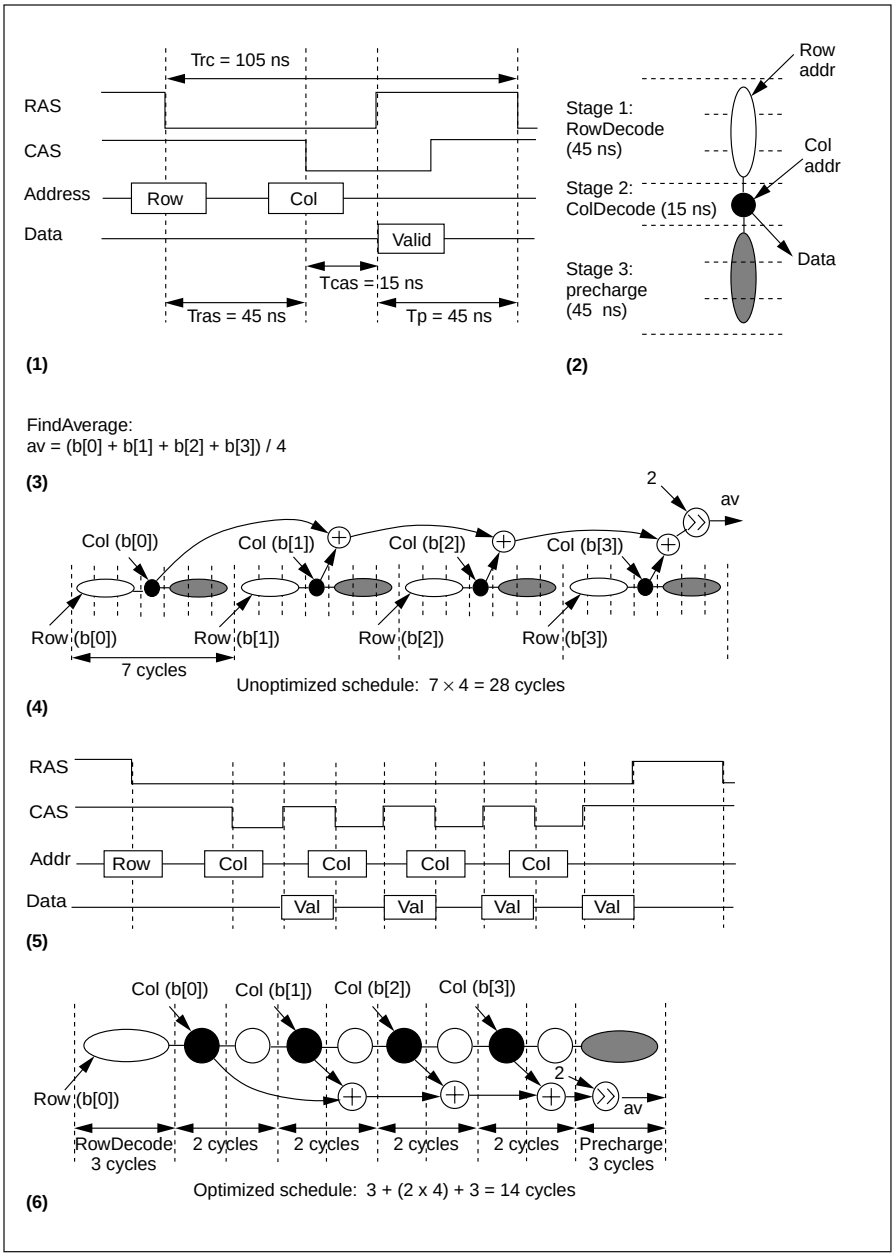


Figure B. Timing diagram for memory read cycle (1); model for memory read operation (2); code for the FindAverage routine (3); treating the memory accesses as independent reads (4); timing diagram of page mode read cycle (5); and treating the memory accesses as one page mode read cycle (6).

To use the above information in an automated scheduling tool, we need to abstract a set of control dataflow graph (CDFG) nodes from the timing diagram.[1] The CDFG node cluster for the memory read operation consists of three stages (Figure B2): row decode; column decode; and precharge. The row and column addresses are available at the first and second stages, respectively, and the output data is available at the beginning of the third stage.

Assuming a clock cycle of 15 ns, and a 1-cycle delay for the addition and shift operations, we can derive the schedule shown in Figure B4 for the code in Figure B3, using the memory read model in Figure B2. Because the four accesses to array b are treated as four independent memory reads, each of these incurs the entire read cycle delay of T_{rc} = 105 ns (7 cycles), requiring a total of 7 4 = 28 cycles.

However, DRAM features such as page mode read can be efficiently exploited to generate a far tighter schedule for behaviors such as the FindAverage example, which successively access data in the same page. Figure B5 shows the timing diagram for the page mode read cycle,

continued on p. 66

continued from p. 65

and Figure B6 shows the schedule for the FindAverage routine using the page mode read feature. The page mode doesn't incur the long row decode and precharge times between successive accesses, thereby eliminating significant delay from the schedule. In this case, the column decode time is followed by a minimum pulse width duration for the CAS signal, which is also 15 ns in our example. Thus, the effective cycle time between successive memory accesses has been greatly reduced, resulting in an overall reduction of 50% in the total schedule length.

The key feature in this dramatically reduced schedule length is the recognition that the input behavior is characterized by memory access patterns amenable to the page mode feature, and by incorporation of this observation in the scheduling phase.[1] The same idea has been applied in the context of memory-aware compilation, where the protocols of specific memory library parts are exploited to hide memory latency during scheduling.[2]

References

1. P.R. Panda, N.D. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration,* Kluwer Academic Publishers, Norwell, Mass., 1999.
2. P. Grun, N.D. Dutt, and A. Nicolau, "Memory-Aware Compilation through Accurate Timing Extraction," *Proc. Design Automation Conf.*, ACM Press, New York, 2000, pp. 316-321.

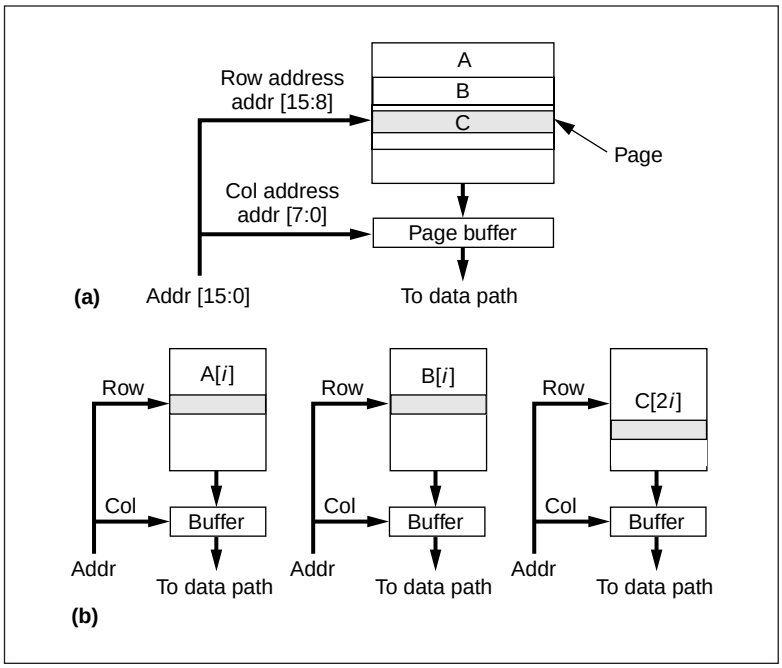


Figure 9. Arrays mapped to single-bank memory (a); three-bank memory architecture (b).

exploration. One interesting aspect of DRAM architecture that can be customized for an application is the banking structure. Figure 9a shows a common problem with the single-bank DRAM architecture. If we have a loop that successively accesses data from three large arrays A, B, and C, each of which is much larger than a page, then each memory access leads to a fresh page being read from storage, effectively canceling the page buffer's benefits.

The page buffer interference problem described here cannot be escaped with a fixed-architecture DRAM. However, an elegant solution to the problem results if we customize the DRAM's banking configuration for our application. Thus, in our example, the arrays can be assigned to separate banks, as Figure 9b shows. Because each bank has its own private page buffer, there is no interference between the arrays, and the memory accesses do not present a bottleneck.

Customizing the banking structure for an application requires solving the memory bank assignment problem: determining an optimal banking structure (number of banks) and determining the assignment of each array variable into the banks to maximize performance.[14,15] The bank assignment has a clear link with the memory assignment problem we described earlier.

We solve the memory bank customization problem by modeling the assignment as a partitioning problem: We partition a given set of nodes into a given number of groups such that a given criterion is optimized (in this case, page-miss count is minimized). The partitioning proceeds by associating a cost of assigning two arrays into the same bank. The cost is determined by the number of accesses to the arrays and the loop count. If the arrays are accessed in the same loop, then the cost is high, thereby discouraging the partitioning algorithm from assigning them to the same bank. On the other hand, if two arrays are never accessed in the same loop, then they

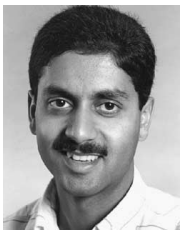
are candidates for assignment into the same bank. This pairing is associated with a low cost, guiding the partitioner to cluster them together. Experiments with several examples show that the bank exploration algorithm does indeed generate good assignments, although the problem is generally NP-complete.[14]

THE POSSIBILITY of customizing the target memory architecture and its associated data organization reveals interesting theoretical and practical problems during embedded-system design. An automatic synthesis tool must use existing compiler technology and extend it to determine an efficient mapping of the application data. But, at the same time, it should intelligently adapt the memory organization parameters that optimize the implementation. In this article, we've described several relevant point techniques for this purpose. Memory allocation, assignment, access ordering, size reduction, and other steps have been combined in a consistent script.[8] Such techniques will evolve with the introduction of more sophisticated memory technology and architecture. Important areas of future investigation include the role of memory in networked and multiprocessor-based embedded systems. Also crucial are new ways to handle very dynamic concurrent applications where complex data types and processes are created and deleted at runtime. ■

■ References

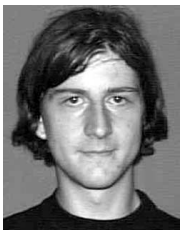
1. J.L. Hennessy and D.A. Patterson, *Computer Architecture—A Quantitative Approach,* Morgan Kaufmann, San Francisco, 1994.
2. P.R. Panda, N.D. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration,* Kluwer Academic Publishers, Norwell, Mass., 1999.
3. C. Kulkarni, F. Catthoor, and H. De Man, "Cache Optimization for Multimedia Compilation on Embedded Processors for Low Power," *ACM/IEEE Proc. Parallel Processing Symp. (IPPS)*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 292-297.
4. C. Kulkarni et al., "Cache-Conscious Data Layout Organization for Embedded Multimedia Applications," *Proc. 4th ACM/IEEE Design and Test in Europe Conf.,* ACM Press, New York, Mar. 2001.
5. P.R. Panda, N.D. Dutt, and A. Nicolau, "On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems," *ACM Trans. Design Automation of Electronic Systems*, vol. 5, no. 3, July 2000, pp. 682-704.
6. P.R. Panda, N.D. Dutt, and A. Nicolau, "Local Memory Exploration and Optimization in Embedded Systems," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 1, Jan. 1999, pp. 3-13.
7. W.-T. Shiue and C. Chakrabarti, "Memory Exploration for Low Power Embedded Systems," *Proc. Design Automation Conf.*, ACM Press, New York, 1999, pp. 140-145.
8. F. Catthoor et al., *Custom Memory Management Methodology,* Kluwer Academic Publishers, Dordrecht, the Netherlands**,** 1998 (also contains references to all the ATOMIUM-related papers published before 1998).
9. S. Wuytack et al., "Minimizing the Required Memory Bandwidth in VLSI System Realizations," *IEEE Trans. VLSI Systems*, vol. 7, no. 4, Dec. 1999, pp. 433-441.
10. W. Verhaegh et al., "Improved Force-Directed Scheduling in High-Throughput Digital Signal Processing," *IEEE Trans. Computer-Aided Design*, vol. 14, no. 8, Aug. 1995, pp. 945-960.
11. P.E.R. Lippens et al., "Allocation of Multiport Memories for Hierarchical Data Streams," *Proc. IEEE Int'l Conf. Computer-Aided Design*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 728-735.
12. V. Lefebvre and P. Feautrier, "Optimizing Storage Size for Static Control Programs in Automatic Parallelizers," *Proc. Euro-Par 97 Conf., Lecture notes in Computer Science*, vol. 1300, Springer-Verlag, Heidelberg, Germany, 1997.
13. P-G. Kjeldsberg, F. Catthoor, and E.J. Aas, "Automated Data Dependency Size Estimation with a Partially Fixed Execution Ordering," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design*, IEEE CS Press, Los Alamitos, Calif, 2000, pp. 44-50.
14. P.R. Panda, "Memory Bank Customization and Assignment in Behavioral Synthesis," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design*, IEEE CS Press, Los Alamitos, Calif, 1999.
15. A. Khare et al., "High-Level Synthesis with SDRAMs and Rambus DRAMs," *IEICE Trans. Fundamentals of Electronics, Comm., and Computer Sciences*, vol. E82-A, no. 11, Nov. 1999, pp. 2347-2355.

Preeti Ranjan Panda is a senior R&D engineer at Synopsys. His research interests include memory issues in high-level and system-level synthesis, compilation for embedded processors, cache-oriented compiler optimization, and hardware-software codesign. He has a B.Tech in computer science and engineering from the Indian Institute of Technology, Madras, and an MS and PhD in information and computer science from the University of California, Irvine.

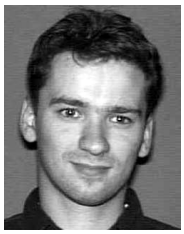


Nikil D. Dutt is a professor in the Center for Embedded Computer Systems at the University of California, Irvine, with academic appointments in the ICS and ECE departments. His research interests include embedded computer systems design automation, computer architecture, and compiler optimization. Dutt has a PhD in computer science from the University of Illinois at Urbana-Champaign.

Alexandru Nicolau is a professor of computer science at the University of California, Irvine, where he leads the ESP/PS parallelization project. His research interests include fine-grain parallelizing compilers and environments, program transformations, and parallel architectures. Nicolau has a PhD in computer science from Yale University.



Arnout Vandecappelle is a researcher and software architect in the DESICS division at IMEC. His research interests include optimization of memory architecture and memory management units for power and performance. Vandecappelle has an M.Sc. in computer sciences from the Katholieke Universiteit, Leuven.



Erik Brockmeyer is an application engineer and researcher in the Design Technology for Integrated Information and Communication Systems (DESICS) division at the Inter-University Micro-Electronics Center (IMEC) in Leuven, Belgium. His research interests include memory-related optimizations in data-dominated multimedia applications, and the related system design technology aspects. Brockmeyer has an Eng. degree in electrical engineering from the University of Eindhoven, the Netherlands.

Eddy De Greef is a senior CAD software engineer in the Multimedia Image Compression Systems group of the DESICS division at IMEC, Heverlee, Belgium. De Greef has an Eng. degree in electrical engineering and a PhD in applied sciences, both from the Katholieke Universiteit, Leuven.

The biography of **Francky Catthoor** appears on page 4 of this issue. The biography of **Chidamber Kulkarni** appears on page 54.

■ Direct comments and questions about this article to Preeti Ranjan Panda, Synopsys Inc., 700 E. Middlefield Rd., Mountain View, CA 94043; panda@synopsys.com.