Abstract—Several application domains, including multimedia and network processing, are highly memory intensive, making memory a bottleneck to designing higher performance and lower power application-specific integrated circuits (ASICs). Design methodologies based on innovative architectures, namely distributed logic-memory architectures and computation-unit integrated memories, have been shown to improve circuit performance significantly.

In this paper, these design methodologies are discussed and evaluated through the implementation of an ASIC for the JPEG still image compression standard. The implemented system is capable of stand-alone image compression and, has been synthesized using the TSMC 0.13um 1.20V eight-layer metal CMOS process. A four-way distributed implementation can achieve an execution time of 2.23ms (a speed-up of 2.87X) for a 128 x 128 input image at the cost of chip area overhead of 51.4% while the energy-delay product is reduced by 2.35X. Design metrics of various other implementations are also compared.

I. INTRODUCTION

A wide variety of multimedia processing, scientific computing and database management applications are dominated by array references that correspond to data accesses from a physical memory system, which present severe performance bottlenecks to circuit and system designers. The increasing processor-memory gap and large disparity between on-chip and off-chip memory bandwidths have become the driving force for several prominent research works to integrate general-purpose processors and memory at various levels. Examples of such architectures include IRAM [1], Smart Memories [2], FlexRAM [3], Terasys [4], etc. These architectures share a common template: an interconnected set of tiles, with each tile consisting of general-purpose processors and localized memory organization. The success of these specialized architectures depends largely on how well applications can take advantage of the underlying hardware resources through compilation support [5], [6], [7], [8].

On the other hand, ASIC designers target circuit design for specific applications, carefully examining the trade-offs among different aspects under various design constraints, e.g., system performance, chip area, energy consumption, etc. Our design methodologies based on distributed logic-memory architecture (DLMA) [9] and computation-unit integrated memory (CIM) [10] exploit “distributed computation” concepts in the ASIC design flow. These techniques have the potential to help designers to systematically improve the performance of customized architectures for memory-intensive applications. However, their effectiveness and applicability for complex designs has not been demonstrated.

In this paper, we present our design of a JPEG image compression IC based on simultaneous use of the DLMA and CIM concepts. We evaluate our hybrid methodology in the context of a commercial design flow, and demonstrate that distributed ASIC architectures can achieve significant performance improvements and reduce the energy-delay product over a conventional design that does not employ a distributed architecture. For example, a two-way distributed implementation can achieve an execution time of 3.74ms (a speed-up of 1.71X) for a 128 x 128 input image at the cost of chip area overhead of 24.7% while the energy-delay product is reduced by 1.57X.

The rest of this paper is organized as follows. Section II describes our design methodologies based on DLMA and CIM architectures. Section III illustrates the system architecture that we implemented for the JPEG grayscale compression application, while Section IV presents experimental results. Section V concludes the paper.

II. BACKGROUND

The basic model and high-level design methodologies for DLMA and CIM are briefly illustrated in Sections II-A and II-B, respectively.

A. Distributed Logic-memory Architecture

Fig. 1 shows a generic DLMA template, consisting of a top-level control logic and K partitions {Sub1, Sub2, ..., and SubK}. The top-level logic responds to the system inputs and co-ordinates the functioning of all partitions. Each partition consists of a controller, datapath and SRAM (as shown for one of the partitions in the figure). In general, the partitions may not be identical, with each partition having a different customized logic implementation or operating on different memory sizes, which are specifically synthesized for the given application. The various partitions execute their assigned computations concurrently, while reading/writing from a local memory (memory in the same sub-system) or remote memory (accesses to the memory in any other subsystem). Communication among sub-systems, and between sub-systems and the top-level logic, can be accomplished in a variety of pre-specified ways, such as hierarchical buses, mixed crossbars with arbitration, etc. Once computations in all the partitions terminate, the top-level logic communicates the results to the global system outputs.

Given a behavioral description, our design methodology can automatically determine a partitioning of operations (computations and memory accesses), along with the requisite partitioning of physical memory, targeting overall system performance under certain design constraints, such as resource and area constraints, etc. The key steps include:

- Use of behavioral simulation to determine the array access footprints, the array locations accessed by the references, and the frequency of the accesses, on which the subsequent operation clustering step is essentially based. Such a scheme allows us to handle array references with index functions of arbitrary complexity.
- Clustering of array access operations into partitions according to their locality and data access workloads, by using the concept of center-of-gravities of data array access footprints and a dendrogram.
- Derivation of operation partitions in “min-cut” style to minimize overall execution time, taking synchronization overheads into account by inserting virtual delay nodes into the datapath.
- Iterative determination of physical memory partitions, which starts with seeking seed locations (memory access “hotspots”) and expands them under the cost function guideline.

The techniques are designed to be applicable to behaviors with complex array access patterns. Therefore, the final output of synthesis can be a homogeneous or heterogeneous set of partitions, depending on the behavior being analyzed.

B. Computation-unit Integrated Memory

Fig. 2 demonstrates a novel architectural template, termed a CIM-based architecture. The template consists of a controller-datapath (hence-
forth, referred to as the processing unit) that works off a memory system enhanced with application-specific hardware (CIM logic). The CIM logic has its own controllerdatapath implementation, which resides within the on-chip memory. Besides the memory read and write operations used in conventional memory systems, the processing unit can now issue instructions to the CIM logic. These instructions are implemented as references to pre-determined memory addresses. The CIM decoder monitors the address bus at all times. When a reserved memory address appears on the bus, it signals the CIM logic to perform the corresponding computations and allows this logic full access to the SRAM. At other times, the CIM logic is bypassed by the decoder as if the CIM is a simple memory providing storage for the processing unit.

The performance can be improved some of the memory-intensive operations are performed by computation units integrated within the memory system, instead of being performed by the main processing unit. The effectiveness of a CIM-based architecture in terms of both efficiency and practicality stems from the following observations:

- **Integrating computation units tightly with the memory system** effectively reduces the quantity of data transferred across the controller-datapath and memory interface, thereby lowering the delays associated with communications over the system bus.
- **Computation units in a CIM can leverage off the wider internal data channels** that may be present within the memory system. For example, in banked memory systems, the computation units are directly connected to the memory banks so that the operations can take advantage of the increased memory bandwidth available.
- **While a CIM functionally requires more “instructions” than the standard read and write commands associated with conventional memories, it requires no physical overhead in exposing the additional functionality, i.e., it preserves the conventional memory interface.**

Our ASIC design methodologies based on DLMAs and CIMs [9], [10] have the potential to benefit many memory-intensive applications. However, their utility has not been demonstrated for a complex design. Moreover, while DLMAs and CIMs have been evaluated separately, their combined benefit has not been evaluated. In the rest of this paper, we describe our design of an encoder for the JPEG image compression standard. The design combines DLMAs and CIMs to achieve high performance without the need for aggressive circuit optimization. We also study various alternative architectures and evaluate tradeoffs in performance, area, and energy consumption.

### III. SYSTEM ARCHITECTURE

The JPEG (Joint Photographic Expert Group) [11] standard specifies a “baseline” method for encoding continuous-tone still pictures, together with a set of extensions. This paper uses an encoding/decoding method for encoding continuous-tone still pictures, together with a baseline method for encoding. We have the potential to benefit many memory-intensive applications. Moreover, while DLMAs and CIMs have been evaluated separately, their combined benefit has not been evaluated. In the rest of this paper, we discuss the two-way partitioned system throughout the rest of the paper. However, the implementation can be easily extended to a larger number of partitions, such as four-way or eight-way, etc. We will compare the results for various partitioning solutions in Section IV.

Fig. 3 shows a choice of dividing matrices $P_{8 	imes 8}$, $TMP_{8 	imes 8}$, and $DCT_{8 	imes 8}$ among two sub-systems A and B. P1 in the figure refers to data from the left half of matrix $P$ ($P[0 \leq i < 8, 0 \leq j < 4]$), while $TMP1$ refers to data from the upper-left quadrant of matrix $TMP$ ($TMP[0 \leq i < 4, 0 \leq j < 4]$), and so on.

Data are organized among the two partitions so that partition A computes $TMP1$ and $TMP3$, while partition B computes $TMP2$ and $TMP4$, during the first matrix multiplication $TMP = C \times P$. In other words, the computation on matrix $P$ is distributed to two sub-systems in the form of a column-major access pattern. Observe that constant array $C$ is needed for both partitions A and B. We make array $C$ available to both partitions at the expense of data duplication (and the associated increase in memory size). Communication-free partitions are now feasible so that the computation on source data $P1$ and $P2$ can be performed concurrently.

The second matrix multiplication $DCT = TMP \times C^T$ accesses the intermediate result matrix $TMP$ in a row-major way, which requires data from both partitions in order to compute any value in matrix $DCT$. For example, data in $TMP1$ and $TMP2$ are needed to compute $DCT1$. However, data from $TMP1$ and $TMP2$ do not need to be accessed at the same time. We reorganized the computing sequence.
so that $DCT_1$ and $DCT_4$ are computed concurrently. Meanwhile, when the computation for $DCT_1$ accesses $TMP_1$, the computation for $DCT_4$ accesses $TMP_4$. Now we can ensure conflict-free memory access between these two partitions. However, we still need to take into account the longer latency of remote memory accesses.

Fig. 4 gives us a fine-grained view of workload partitioning while processing each image block. The input image consists of many blocks, which can be interleaved as shown in Fig. 5. The left half of all the blocks is assigned to partition A, and the right half to partition B. Data interleaving is conducted by the top-level logic when the source image is input to the system through global I/O.

**B. Quantization**

The resulting 64 DCT coefficients are regarded as the relative amplitudes of two-dimensional “spatial frequencies” that comprise the input signal’s “spectrum”. They consist of the DC coefficient, the coefficient with zero frequency in both dimensions (the origin, which is the upper-left corner in the DCT coefficient matrix), and the remaining 63 AC coefficients.

Quantization is performed by dividing each value in the DCT coefficient matrix by the step size of the quantizer, which varies depending upon its distance from the origin and the quality factor. The quality factor is externally specified and determines the trade-off between image quality and size of the compressed image.

The quantization of a DCT matrix is calculated using the equation given below.

$$Quant(i, j) = \frac{DCT(i, j)}{1 + quality \times (1 + i + j)}$$

where $Quant_{8x8}$ is the quantized $DCT$ value matrix, and $quality$, given as an input parameter, is the compression quality factor.

Since $quality$ is part of the system input, the step size of the quantizer, i.e., $1 + quality \times (1 + i + j)$ cannot be pre-computed at compile-time. However, it can be computed before quantization starts. The computation is implemented using a CIM. In each sub-system, another smaller SRAM, besides the main SRAM, is employed to store all the quantizer step sizes as a look-up table (shown in Fig. 6). Using a separate SRAM prevents interference with the main SRAM bandwidth, allowing us to perform the look-up table operation in the background at the same time the DCT computation is performed. Besides the quantizer look-up table, matrices $TMP$ and $C$ are also allocated to the smaller SRAM, which provides opportunities to take advantage of the wider internal memory bandwidth.

Furthermore, the entire quantization step can be implemented inside the memory system by using the CIM logic. After DCT computation finishes, the processing unit issues a CIM instruction using a reserved memory address, causing the CIM logic to start quantization. The CIM logic fetches data from matrix $DCT$, as well as the quantization step sizes from the look-up table, and performs division operations. The quantized values, $Quant$, are written back into the main SRAM directly. Hence, no data transactions are needed to/from the main processing unit through the memory interface, thus reducing the memory interface traffic and saving the communication latency.

The range shifting computation mentioned in Section III-A is also implemented as part of the CIM logic. It basically reduces each value of source image pixels by a shifting factor.

**C. Zig-zag Sequence and Run-length/Entropy Encoding**

After quantization, a data re-organization in a “zig-zag” sequence helps to increase the number of consecutive zeros, which helps facilitate entropy coding in the next step of the compression sequence.

Three different encoding techniques are used in this implementation: run-length encoding, entropy encoding, and “end of block” tagging. Run-length encoding converts a number of recurring characters into one character, followed by the number count. Entropy encoding is a compression technique used to represent more common patterns with fewer bits than less common patterns. “End of block” tagging employs the bit pattern “0000” to indicate the “end of block” (EOB), i.e., all remaining matrix values are zeros.

Zig-zag sequence re-organization and run-length/entropy encoding are carried out by the top-level logic. Fig. 6 illustrates the entire system architecture of a two-way partitioned DLMA with CIM support for the JPEG encoder.

**IV. CHIP IMPLEMENTATION AND RESULTS**

We synthesized the JPEG encoder architecture discussed in this paper within the framework of an existing ASIC design flow. High-level synthesis (HLS) of the input C behavior (without our architectural
augmentations) was performed using an HLS tool, called Cyber [12], from NEC. The resultant RTL implementation was synthesized and technology-mapped with Synopsys Design Compiler using the TSMC 0.13 micron standard cell-based library (1.20V; eight-layer metal CMOS process technology), to obtain a gate-level netlist. Gate-level functional simulation was performed using the Synopsys VSS simulator. Sequence Design’s integrated circuit power estimation tools, Watt Watcher and Peak Watcher, were used to estimate circuit power and energy consumption in the full simulation mode. Finally, layouts of the original and enhanced circuits were obtained using a suite of layout tools from Cadence (Silicon Ensemble and DFII toolset).

The TSMC 0.13µm process high-speed single-port synchronous SRAM generator was used to produce high-performance memory blocks. SRAM access is synchronous and is triggered by the rising edge of the clock, CLK. Input address, input data, write enable, and chip enable are latched by the rising edge of the clock, respecting individual setup and hold times. Power dissipation is minimized using static circuit implementations. A standby mode is provided to further reduce power dissipation during periods of non-operation. While in standby mode, address and data inputs are disabled; data stored in the memory is retained, but the memory cannot be accessed for reads or writes. Static power consumption of the SRAM is limited to leakage provided all of the input signals except CLK are held steady.

TABLE I

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Execution time (µs)</th>
<th>Total area (mm²)</th>
<th>Energy x delay (nJ·s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic design</td>
<td>6.40</td>
<td>1.46</td>
<td>343.8</td>
</tr>
<tr>
<td>Two-way memory banking</td>
<td>5.56 (1.15X)</td>
<td>1.66 (13.7%)</td>
<td>286.5 (1.20X)</td>
</tr>
<tr>
<td>Two-way partition without CIM</td>
<td>4.66 (1.37X)</td>
<td>1.73 (18.5%)</td>
<td>251.3 (1.31)</td>
</tr>
<tr>
<td>Two-way partition with CIM</td>
<td>3.74 (1.71X)</td>
<td>1.82 (24.7%)</td>
<td>219.0 (1.57X)</td>
</tr>
<tr>
<td>Four-way partition with CIM</td>
<td>2.23 (2.87X)</td>
<td>2.21 (31.4%)</td>
<td>146.3 (2.35X)</td>
</tr>
<tr>
<td>Eight-way partition with CIM</td>
<td>2.20 (2.91X)</td>
<td>2.91 (99.3%)</td>
<td>186.2 (1.85X)</td>
</tr>
</tbody>
</table>

Fig. 7. Layout of the JPEG implementation methodolgy helps distribute hardware sources (computing logic, memory, interconnect, etc.) throughout the chip, while restricting computations and communications to geographic proximities. Experiments with various implementations in the context of a state-of-the-art design flow demonstrated that distributed ASIC architectures achieve significant performance improvements at reasonable cost in chip area. Furthermore, our results showed that distributed architectures are also good from energy x delay considerations.

TABLE II

<table>
<thead>
<tr>
<th>Functionality</th>
<th>JPEG grayscale compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>TSMC 0.13 µm CMOS process</td>
</tr>
<tr>
<td>Total area</td>
<td>1.43mm x 1.27mm</td>
</tr>
<tr>
<td>On-chip SRAM</td>
<td>two 16K x 8, two 2K x 8</td>
</tr>
<tr>
<td>Power voltage supply</td>
<td>1.2V</td>
</tr>
<tr>
<td>Input image</td>
<td>128 x 128 grayscale image</td>
</tr>
<tr>
<td>Execution time</td>
<td>3.74 ms</td>
</tr>
<tr>
<td>Operation speed</td>
<td>350MHz at 1.2V</td>
</tr>
<tr>
<td>Energy x delay</td>
<td>219.0 nJ·s at 1.2V, 350MHz</td>
</tr>
</tbody>
</table>

We compared five different implementations for the JPEG encoder: 1) Monolithic design: a conventional monolithic design without any distributed architecture employed, 2) Two-way memory banking: memory partitioning using two-way memory banking, 3) Two-way partition without CIM: a two-way partition without CIM support, 4) Four-way partition with CIM: a four-way partition with CIM support, and 5) Eight-way partition with CIM: an eight-way partition with CIM support. The resulting circuit implementations and layouts were compared with respect to the following metrics: execution time, area, and energy-delay product. These metrics were extracted from the technology-mapped circuits and designer-provided testbenches. The results are summarized in Table I. The specification of the two-way distributed implementation shown in Fig. 6 is given in Table II, while Fig. 7 demonstrates the circuit layout. The four-way partitioned implementation has better performance relative to total area. Performance improvements begin to diminish as the number of partitions reaches eight, in which case remote memory accesses among sub-systems start dominating. Eventually, the benefits of partitioning get outweighed by communication overheads.

V. CONCLUSIONS

The JPEG grayscale image compression standard was implemented based on the DLMA and CIM architectures. Our systematic design