A Reconfigurable Multifunction Computing Cache Architecture

Huesung Kim, *Student Member, IEEE*, Arun K. Somani, *Fellow, IEEE*, and Akhilesh Tyagi, *Member, IEEE*

***Abstract*—A considerable portion of a microprocessor chip is dedicated to cache memory. However, not all applications need all the cache storage all the time, especially the computing bandwidth-limited applications. In addition, some applications have large embedded computations with a regular structure. Such applications may be able to use additional computing resources. If the unused portion of the cache could serve these computation needs, the on-chip resources would be utilized more efficiently. This presents an opportunity to explore the reconfiguration of a part of the cache memory for computing. Thus, we propose adaptive balanced computing (ABC)—dynamic resource configuration on demand from application—between memory and computing resources. In this paper, we present a cache architecture to convert a cache into a computing unit for either of the following two structured computations: finite impulse response and discrete/inverse discrete cosine transform. In order to convert a cache memory to a function unit, we include additional logic to embed multibit output lookup tables into the cache structure. The experimental results show that the reconfigurable module improves the execution time of applications with a large number of data elements by a factor as high as 50 and 60.**

***Index Terms*—Cache memory, reconfigurable computing.**

I. INTRODUCTION

THE number of transistors on a chip has increased dramatically in the last decade. Within the next five to ten years, we will have a billion transistors on a chip. In a modern microprocessor, more than half of the transistors are used for cache memories. This trend is likely to continue. However, many applications do not use the entire cache all the time. Such applications result in low utilization of the cache memory. Many times, these applications are bandwidth limited. This suggests using the unutilized cache resources for computing. Therefore, we propose adaptive balanced computing (ABC)—dynamic resource configuration on demand from application—between memory and computing resources.

Several researchers have studied the use of reconfigurable logic for on-chip coprocessors [1]–[5]. Such logic can speed up many applications. An on-chip coprocessor improves the performance of the applications and reduces the bottleneck of off-chip communications. In Garp architecture [1], programmable logic resides on a processor chip to accelerate some computations. The frequently used computations are mapped to the programmable logic. If an application does not need the logic, these functions remain idle. PipeRench [6] reconfigures the hardware every cycle to overcome the limitation of hardware resources.

A field-programmable gate array (FPGA) can be viewed as a two-dimensional (2-D) array of configurable logic blocks (CLBs)—CLB is a primitive programmable element (PE)—with interspersed routing channels. Each CLB consists of configurable gates realized through lookup tables (LUTs). The LUT is an essential component to construct FPGAs. LUTs usually have four inputs and one output out of an SRAM-based memory to keep the overall operation and routing efficient. However, the 1-bit output granularity of each LUT results in a large interconnect area—even larger than the area of LUTs—and delay due to a number of switches for the programmability [8].

Xilinx Virtex FPGA family [7] allows concurrent and partial reconfiguration. However, the dynamic partial reconfiguration can be only done at the granularity of a configurable logic block consisting of four-input LUTs. An advantage of this architecture is that a number of smaller configuration memory blocks can be combined to obtain a larger memory. However, a fine-grained memory cannot be synthesized efficiently in terms of area and time. In particular, providing a large number of decoders for small chunks of memory is expensive.

These observations motivate the design of a reconfigurable module that works as a function unit as well as a cache memory. Our goal is to develop such a reconfigurable cache/function unit module to improve the overall performance with low area and time overhead using multibit output LUTs. The expectation is that significant logic sharing between the cache and function unit would lead to relatively low logic overhead for a reconfigurable cache (RC). If the area overhead of an RC exceeds the area of the dedicated logic for that function, or if the time overhead of cache is significant (if the time increases more than 5–10%—commonly treated as a significant increase), this is too big a compromise.

Single-instruction multiple-data (SIMD) multimedia applications with large streamed working data sets, in which data are used once and then discarded [9], can be accelerated by a special function unit. A larger on-chip cache hardly helps these applications due to the lack of temporal locality [10], [11]. Since SIMD applications need less reconfiguration at run-time by the nature of SIMD, the run-time reconfiguration does not affect the overall execution time significantly once we configure the RC as a function unit. The multiply-and-accumulation [(MAC)—core of finite impulse response (FIR)] and ROM-based distributed arithmetic [(DA)—core of discrete/indiscrete cosine transform

Manuscript received January 15, 2000; revised October 17, 2000. This work was supported by Carver Trust Grants, Iowa State University, and by the National Science Foundation under Grant CCR-9900601.

The authors are with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011 USA.

Publisher Item Identifier S 1063-8210(01)03515-6.

1063–8210/01$10.00 © 2001 IEEE

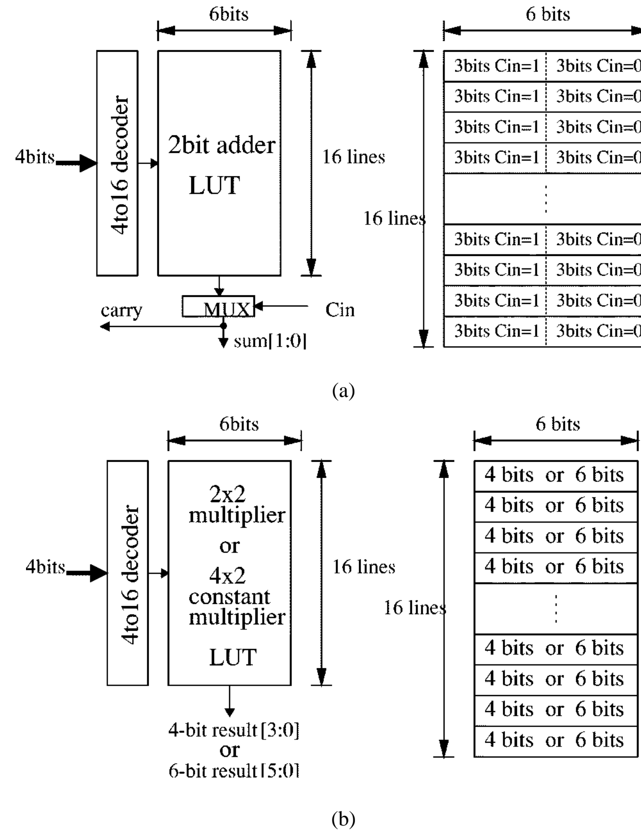

Fig. 1. Multioutput LUTs: (a) 2-bit adder; (b) 2×2 or 4×2 constant coefficient multiplier.

(DCT/IDCT)] functions are good examples of such SIMD applications. Such structured computations are more easily targeted for a reconfigurable cache, especially within the low area and time overhead constraints. Hence, in the first phase of this research, we have implemented two computing primitives needed in structured video/audio processing: FIR and DCT/IDCT. We partition the cache into several smaller caches. Each cache is then designed to carry out a set of specialized dedicated compute-intensive functions.

The experimental results show that the reconfigurable module improves the execution time of applications with a large number of data elements by a large factor (as high as 50 and 60). With respect to two cache organization models, a memory cell array cache and a base array cache with segmented/partitioned bit/word lines, the area overhead of the reconfigurable cache module for FIR and DCT/IDCT is less than the core area of those functions. The reconfigurable cache (RC) based on a cache with a large memory cell array may have faster access time and larger area overhead, while the RC built in the base array cache structure may increase the access time slightly with lower area overhead. The concept of reconfigurable cache modules can be applied at Level-2 caches instead of Level-1 caches to provide an active-Level-2 cache similar to "Active pages" in [12].

Section II describes the architecture of a reconfigurable module with the function unit and cache operations with multibit output LUTs. The configuration and scheduling of the module are described in Section III. Section IV presents experimental results on the reconfigurable module. We conclude this paper in Section V.

II. Reconfigurable Cache Module Architecture

In this section, we describe how the proposed reconfigurable cache module architecture (RCMA) is organized and works. First, we introduce multibit output LUTs to be used in the reconfigurable cache (RC) in Section II-A. Second, we show the overall architecture of RC and a conventional microprocessor architecture in Section II-B. Third, we describe the core design of RC architecture, such as how it operates as a cache memory and a special function unit in Section II-C. In Section II-D, we compare and estimate the cache access time of RC.

A. Multibit Output LUTs

In most FPGA architectures, an LUT usually has four inputs and one output to keep the overall operation and routing efficient. However, an SRAM-based single-bit output LUT does not fit well with a cache memory architecture because of a large area overhead for the decoders in a cache with a large memory block size. Instead of using a single-bit output LUT, we propose to use a structure with multibit output LUTs. Such LUTs produce multiple output bits for a single combination of inputs and are better suited for a cache than the single-bit output LUTs. Since a multibit output LUT has the same inputs for all output bits, it is less flexible in implementing functions. However, it is not a major bottleneck in our problem domain. A 2-bit carry select adder and a 2-bit multiplier or a 4×2 constant coefficient multiplier (all need the same size, up to 6-bit output, of LUT) are depicted in Fig. 1(a) and (b), respectively.

If a multibit output LUT is large enough for a computation, no interconnection (for example, to propagate a carry for an

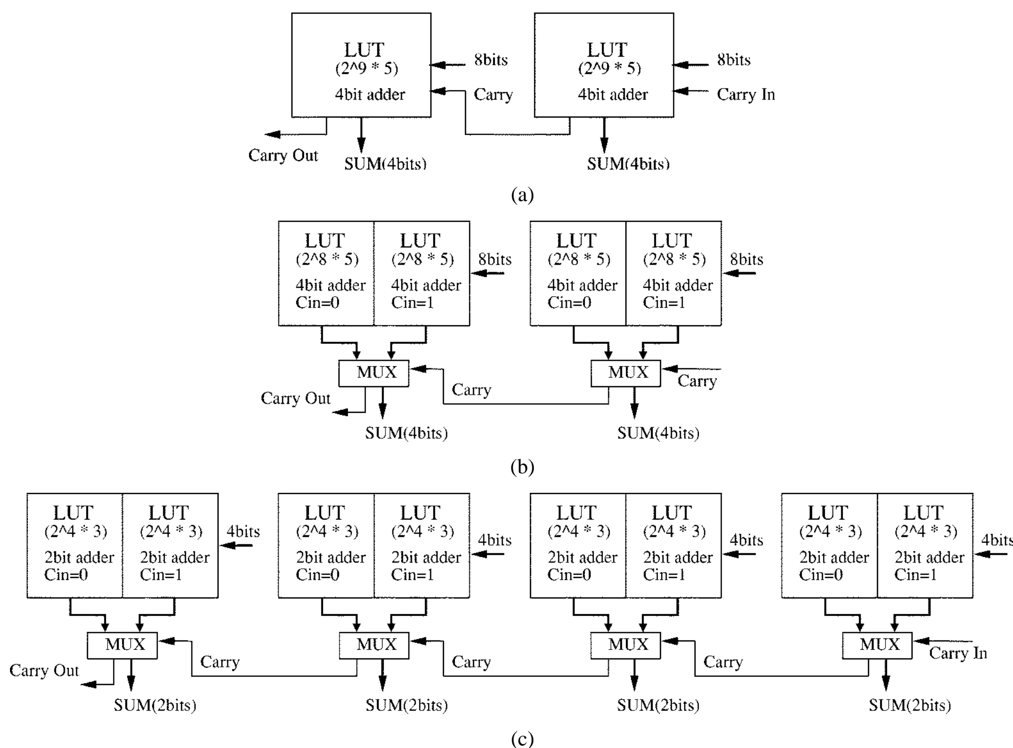

Fig. 2. Eight-bit adder using (a) two 9-LUTs, (b) two 8-LUTs, and (c) four 4-LUTs.

adder) may be required since all possible outputs can be stored in the large memory. In addition, unlike a single-bit output LUT, a multibit output LUT requires only one decoder or a multiplexer with multiple inputs. Thus, the area for decoders reduces. However, the overall memory requirement to realize a function increases. The required memory size increases exponentially with the number of inputs. Therefore, multibit LUTs may not be area-efficient in all situations. The computing time in this case may also not reduce much due to the complex memory block and the increased capacitance on long bit lines for reading.

Instead of using one large LUT, we show implementations of an 8-bit adder with a number of smaller multibit output LUTs, as shown in Fig. 1. Fig. 2(a) depicts an 8-bit adder consisting of two nine-input LUTs. Each 9-LUT has two 4-bit inputs, one 1-bit carry-in, and a 5-bit output for a 4-bit addition. Thus, the total memory requirement is $2 \times 2^9 \times 5 = 5120$ bits. The carry is propagated to the next 9-LUT after the previous 4-bit addition in one LUT is completed (i.e., a ripple carry). Since each LUT must be read sequentially, this adder takes longer to finish an addition. By employing the concept of carry select adder as depicted in Fig. 2(b), a faster adder using two 8-LUTs can be realized as the reading of the LUTs does not depend on the previous carry. In this case, the actual result of each 4-bit addition is selected using a carry propagation scheme. However, all LUTs are read in parallel. The total time for the modified adder is the sum of the read time for one 8-LUT and the propagation time for two multiplexers. Thus, it is faster. This adder also requires the same amount of memory (i.e., $4 \times 2^8 \times 5 = 5120$ bits).

To make an area efficient adder, a 4-LUT with 6-bit outputs can be employed [Fig. 2(c)]. The same carry propagation scheme as in Fig. 2(b) is applied to the 4-LUTs to implement an 8-bit adder, but four 4-LUTs are used. The total time of the adder using the 4-LUTs might be higher than that using the 8-LUTs because it has twice the number of multiplexers to be propagated. However, the read time for a 4-LUT is faster than for an 8-LUT since it has a smaller decoder and shorter data lines for memory reading. We therefore recommend the design in Fig. 2(c).

B. Overview of the Processor with Reconfigurable Caches

In an RCMA, we assume that the data cache is physically partitioned into n cache modules. Some of these cache modules are dedicated caches. The rest are reconfigurable modules. A processor is likely to have 256 KB to 1 MB Level-1 data cache within the next five to ten years. Each cache module in our design is 8 KB, giving us 32–128 cache modules. A reconfigurable cache module can behave as a regular cache module or as a special-purpose function unit.

Fig. 3 shows the overview of the processor with reconfigurable caches (RCs). In an extreme case, these n cache modules can provide an n-way set associative cache. m modules out of n cache modules are reconfigurable. Whenever one of these cache modules is converted into a computing unit, the associativity of the cache drops or vice versa. Alternatively, the address space can be partitioned dynamically between the active cache modules with the use of address bound registers to specify the cached address range. The details of this architecture are being developed in [13]. The RCMA simulation on real multimedia programs [13] expects to settle a mix of the following issues. How large should the mix of RC modules m/n be? How many and what functions ought to be supported in each RC? What kind of connectivity is needed between these RCs? RC1, RC2, RC3, . . ., RCm in Fig. 3 can be converted to function units, for example,

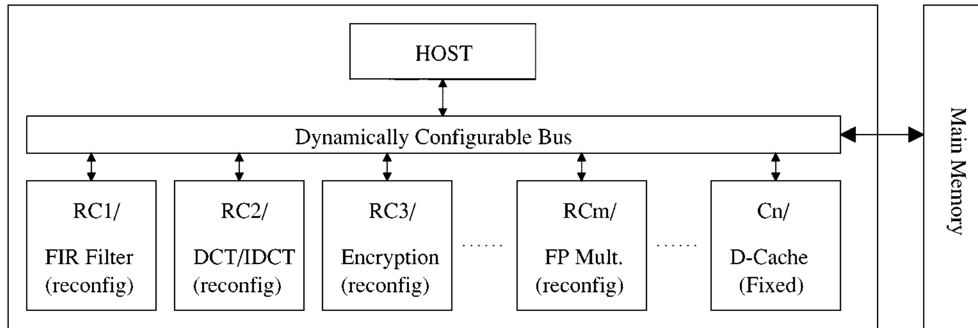

Fig. 3. Overview of a processor with multiple reconfigurable cache modules.

to carry out functions such as FIR filter, DCT/IDCT, encryption, and a general computation unit like a floating-point multiplier, respectively. When some subset of the m RCs is used as function units, the other caches continue to operate as memory cache units as usual. It is also possible to configure some cache modules to become data input and output buffers for a function unit. The RCs are configured by the processor in response to special instructions. We envision a dynamically configurable bus (as shown in Fig. 3) to support dynamic communication needs between RCs.

In this paper, we propose that each cache module be designed to be reconfigurable into one of several specific function units. Since each reconfigurable module can be converted into a small set of functions with similar communication needs, interconnections for each RC are fixed to be a super set of the communication needs of the supported functions. The advantages of fixed interconnection are as follows. The fixed interconnection is less complex, takes less area, and allows faster communication than a programmable interconnection. Moreover, our experience demonstrates the feasibility of merging several functions into one RC with fixed interconnections.

C. Organization and Operation of a Reconfigurable Cache Module

Since we target computation-intensive applications with a regular structure, such as digital signal processing (DSP) and image applications (FIR, DCT/IDCT, Cjpeg, Mpeg, etc.), as mentioned in Section I, we first partition them at coarse level into repeated basic computations. A function in each stage can be implemented using the multibit output LUTs, as described in Section II-A. We only add pipeline registers to each coarse-level stage, which contains a number of LUTs, to make the entire function unit efficient. All these registers are enabled by the same global clock. Therefore, a number of coarse-level computations can be performed in a pipelined fashion.

Fig. 4 shows a coarse template for a module. The cache can be viewed as a 2-D matrix of LUTs. Each LUT has 16 rows to support a 4-LUT function and as many multibits in each row as required to implement a particular function. In the function unit mode—in which the RC works as a special function unit, the output of each row of LUTs is manipulated to become inputs for the next row of LUTs in a pipelined fashion. In the cache memory mode—in which the RC works as a conventional cache memory, the least significant 4 bits of the address lines are connected to the row decoders dedicated to each LUT. The rest of

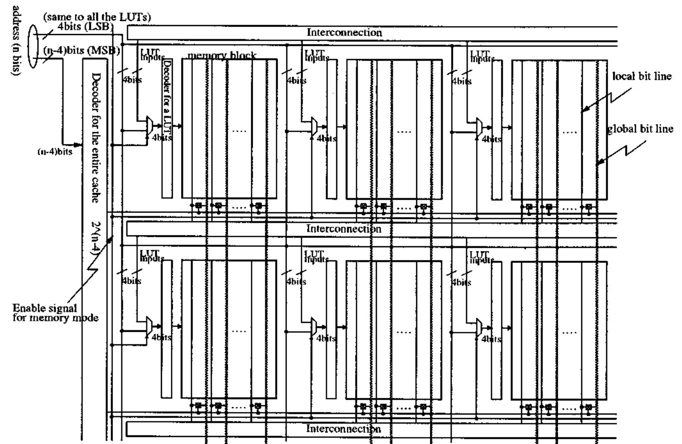

Fig. 4. Cache architecture in the reconfigurable module.

the address lines are connected to a decoder for the entire cache in the figure. In the cache memory mode, the LUTs take the 4-bit address as their inputs selected by the enable signal for the memory mode. Therefore, regardless of the value of the upper bits in the address, the dedicated row decoder selects a word line in each row of LUTs. This means one word is selected in each LUT row according to the least significant 4 bits.

Each LUT thus produces as many bits as the width of the LUT. These are local outputs of the LUTs. These outputs are available on the local bit lines of each LUT row. For a normal cache operation, one of the local outputs needs to become the global output of the cache. This selection is made based on the decoding of the remaining $(n-4)$ address bits decoded by the higher bit decoder. The local outputs of the selected row of LUTs are connected to the global bit lines. The cache output is carried on the global bit lines, as shown in Fig. 4. Thus, the output of any row of LUTs can be read/written as a memory block through global lines. We propose that these global lines be implemented using an additional metal layer. The global bit lines are the same as the bit lines in a normal cache.

Both decodings can be done in parallel. After a row is selected by both the decoders, one word is selected through a column decoder at the end of the global bit line, as in a normal cache operation. In the figure, the tag part of a cache is not shown, and a direct-mapped cache is assumed for the module. However, the concept of reconfigurable cache can be easily extended to any set-associativity cache because the tag logic is independent of the function unit's operations.

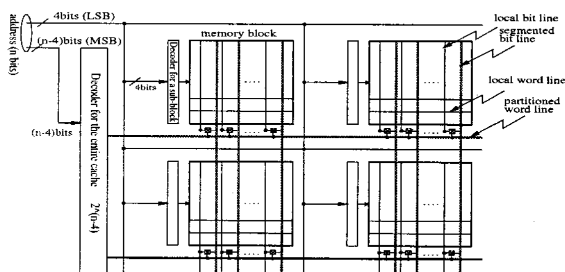

Fig. 5. Parallel decode cache architecture (base array cache) for faster cache access time.

D. Access Time for Cache Operations

We compare the access times for the RC with the access time for a fixed cache module of comparable size. The base fixed cache module from which a reconfigurable cache is derived comes in two styles. The first is a memory cell-only array cache with one address decoder and one data array. The second is a parallel decoding cache with segmented-bit lines and partitioned-word lines. The segmented-bit lines are divided every 16 cache blocks and enabled by the decoder for the high-order address bits with switches like the global bit lines in Fig. 4. The partitioned-word lines are divided into the decoding lines from the the high-order address decoder and local word lines in a submemory block from each dedicated decoder in Fig. 4. The local word lines select one block in every 16 cache blocks, and one of them is selected by the high-order address decoder. The base array cache is shown in Fig. 5.

The memory-cell-only array cache has single-level decoding, leading to low area and slow access time. An RC based on this design reduces access time by introducing hierarchical decoding at a cost of large area overhead. A base array cache structure, however, already incorporates access time advantage of hierarchical decoding, and hence also needs more area. An RC based on this design, hence, shows a slight degradation in access time with a very small area overhead. We analyze the RC access time for cache operations in terms of address decoding time and word/bit-line propagation time. Other components of access time, such as sense amplifier and column decoding, do not differ over the two cache organizations. The access times for an RC based on a memory cell array cache and the base array cache are estimated below, respectively.

1) Memory Cell Array Cache: The cache with the reconfigurable structure may have a faster address decoder than a memory cell array cache, which contains one main address decoder and a bunch of adjacent memory cells. Since each LUT, with its own row decoder for addressing in the reconfigurable module, is much smaller than a large synthesized memory cell array in a conventional cache, the decoding time of an LUT is faster than the decoding time of a large cache. As mentioned earlier, since two decoders can decode in parallel, possible word lines in a cache according to the least significant 4 bits may be ready to be read or written before the main row decoder even finishes decoding an address. The assumption here is that the main decoder has a larger number of address bits. Since the two decoding operations are independent, the delay of decoders is the maximum of two decoding times in the reconfigurable module. If there are many LUTs that take the same lower 4 bits in the module, we have to consider the increased capacitance due to the fan-out of the lower address bits. If the delay of decoding is higher, we may need a larger driver for the least significant 4 bits to reduce the delay. However, the drivers will not affect the size of the reconfigurable module much, as we can put a driver into the space saved due to the reduction in the decoder size for higher order bits.

Each bit line in a normal cache is replaced by the global line in the proposed architecture. Since the global line does not drive any gates (only the drain connections of the switches placed in an interleaved fashion—every 16 cache blocks), the reconfigurable module does not have higher delay due to the global lines. Although the global bit line in RC is stretched by inserting the interconnection between LUT rows, the number of drains—dominant capacitance in the bit line—is reduced by a factor of 16. Thus, the segmented global bit line in the RC has less capacitance than the bit line of a conventional cache. Additionally, the local bit-line discharge can be done in parallel with the higher address bit decoding and word-line propagation. This indicates that a data signal from a memory cell through the bit line in the module is propagated faster than a normal cache.

The word line in the reconfigurable cache is longer than in a memory cell array cache due to additional row decoders for each LUT. Therefore, the propagation delay of a signal from the higher bit decoder through the word line in the module is slightly higher than in a normal cache. However, the sum of two propagation times, word and bit lines, is smaller than in a conventional cache since the local bit line in RC starts discharging before the word line finishes the propagation.

As mentioned earlier, other delays are similar in both cases. In summary, the cache access time of RC is faster in decoding time and bit/word-line propagation time. Therefore, the RC is faster than a conventional memory cell array cache in read and write cache operations.

2) Base Array Cache: Recall that the base array cache performs parallel decoding with segmented-bit and partitioned-word lines. Cache implementations may have a similar or more efficient parallel decoding structure with segmented bit lines. Unlike the RC organization with vertical and horizontal partitions, some partitioned caches might employ only the vertical partition of cache blocks for less capacitance on the segmented bit lines because the stretched word line causes more delay than a large subblock. However, if we consider the word-line propagation time with the discharging time of local bit lines, the horizontal partition with the dedicated decoders to each LUT (submemory module) can make the word-line propagation faster. As described earlier, discharging the local bit line can start with charging the word line in RC. If we partition a cache block only vertically for segmented bit lines, one bit line of each bit-line pair in a cache block cannot be discharged unless the entire word line is fully charged (decoded) from the higher address-bit decoder. Although the entire stretched word-line propagation in RC is slightly slower due to the insertion of the dedicated LUT decoders, the parallel discharge/charge of the local bit/word line compensates the stretched word line (or makes it even faster). Therefore, we compare the access time for RC to the base array cache

partitioned vertically and horizontally with the segmented bit lines and partitioned word lines.

The access time of reconfigurable cache is slightly slower than that of a plain cache due to the stretched bit lines caused by the interconnections between LUT rows in the RC. Based on the SPICE model parameters for 0.5-μm technology in [14], the capacitance of the stretched bit line in the RC is increased by 11% over the segmented bit line in the caches. Since the bit-line access time constitutes 8% of the overall cache access time [15], the access time overhead due to the stretched line is about 1% of the overall cache access time. Since the word-line propagation time, the decoding time, and other components in RC are similar to those in the base array cache as described above, the overall cache access time, therefore, is slower than the base array cache by about 1%. The area overheads for FIR and DCT/IDCT function modules are given in Section IV-B with respect to both cache models.

III. Configuration and Scheduling

We explain how to store and place the configuration data in a cache memory based on a conventional cache architecture in Section III-A. Then, in Section III-B, we describe a way to load the configuration data initially and to load partial configuration data at run-time. The scheduling and controlling data flow for RC is described in Section III-C. Finally, we discuss compiler issues in Section III-D.

A. Configuration of a Function Unit

To reduce the complexity of the column decoding in a normal cache memory, data words are stored in an interleaved fashion in a block. The distance in bits between two consecutive bits of a word is equal to the number of words in a block. Due to the interleaved placement of data words in a cache block, we cannot write one entry of a multibit output LUT by writing one word in a cache. This implies that we can only write one bit into a LUT if the width of LUT is the same as the number of words in a cache block *or* we can write 2 bits simultaneously into an LUT if the width of the LUT is half the number of words in a cache block. For example, if a 4-LUT produces an n-bit-wide output for a function and the number of words in a cache block is n, $16 \times n$ words—16 for the number of entries and n for the width of LUT output (1 bit from each word)—are required to be written to the LUT in the cache. However, since other LUTs placed in the same cache blocks (LUT row) can also be programmed simultaneously, no more than $16 \times n$ words are required to fill up the contents of all LUTs in the entire LUT row. In addition, if the width of an LUT is larger than the number of words in a cache block, multibit writing is performed into each LUT in an LUT row, as mentioned above. This places a restriction that the width of a multibit output LUT be an integral multiple of the number of words in a cache block to allow an efficient reconfiguration of all LUTs in a row. The number of LUTs in a column—placed vertically—for a pipeline stage may also be required to be a power of two. Since all cache structures are based on a power of two, it is more convenient to make all LUT parameters (length and width) a power of two to avoid a complicated controller and an arbitrary address generator. This may result in underutilization of memory. However, the idle memory blocks for LUTs are not likely to be a problem when the module is used as a function unit due to availability of sufficient memory size in a cache.

B. Initial/Partial Reconfiguration

Initial configuration converts a cache into a specific function unit by writing all the entries of LUTs in the cache. The configuration data to program a cache into a function unit may be available either in an on-chip cache or an off-chip memory. Loading time for the configuration data in the latter case will be larger than in the former case. The configuration data may be prefetched by the controller or the host processor to reduce the loading time from off-chip memory. Using normal cache operations, multiple writes of configuration data to the LUTs are easily achieved.

An RC operating as a function unit can also be partially reconfigured at run-time using write operations to the cache. When a partial reconfiguration occurs, the function unit must wait for the reconfiguration to complete before feeding the inputs. Since computation data (input and output) and reconfiguration data (contents of LUTs) for a function unit share the global lines for data buses, we cannot perform both computing and partial reconfiguration at the same time. It is possible to perform both computations and reconfigurations simultaneously if we have separate data lines for computation data and configuration data. To process a large number of data elements, we do not need to reconfigure often. For example, in convolution application with 256 taps, we need to reconfigure a module implementing eight taps 32 times.

The time to configure initially from the normal cache memory mode to a function unit mode or to reconfigure a part of a function unit depends on the number of cycles to write words into a cache. Initial configuration time dominates the total configuration/reconfiguration time. The partial reconfiguration at run-time usually loads a small part of configuration. The targeted SIMD applications require small initial and partial configurations, and hence configuration has a small effect on overall execution time. The formulas for the configuration time and the simulated configuration times (including initial and partial) for FIR and DCT with various function parameters are shown in Section IV-C. With a smaller number of data elements, the configuration time dominates the total execution time. However, the total execution time is not dominated by the configuration time when the number of input data elements exceeds a threshold (which is true for SIMD applications).

C. Scheduling and Controlling Data Flow

A cache module can also be used to implement a function with a larger number of stages than what can be realized by the reconfigurable cache in one pass. In this case, we divide the function into multiple steps. That is, S stages required for a function can be split into sets, $S_1, S_2, \ldots, S_k$, such that each set S_i can be realized by a cache module. If all S_is are similar, then we can adapt data caching, as described in [16], to store the partial results of the previous stage as input for the processing by the next configuration. “Similar” here means that the LUT contents may change, but the interconnection between stages is the same. This happens, for example, in convolution application.

By changing the contents of LUTs, we can convert a stage in the cache block to carry out the operation of a different set of pipeline stages. In general, MAC is a very common function in many DSP and image-processing problems. The applications computing with MAC may have the same interconnection for all the computing stages with different LUT contents.

In a data caching scheme, we place all input data in a cache and process it for the first set of stages S_1. Following this, the cache module is configured for stages S_2. We have to store the intermediate results from the current set of stages into another cache and then reload them for the next set of computations. Therefore, we need two other cache modules to store input and intermediate data, respectively. These modules are address-mapped to provide efficient data caching for intermediate results. The role of the two caches can be swapped during the next step when a computation requires the intermediate results as inputs and generates another set of intermediate results. If both an input and an intermediate result are required by all the computations, the two caches cannot be swapped. The two caches must be large enough to hold input and intermediate results, respectively. Moreover, the reconfigurable cache must be able to accept an input and an intermediate result as its inputs.

The host processor needs to set up all the initial configurations, which include writing configuration data into LUTs and configuring the controller to convert a cache into a function unit. To do this, the host processor passes the information about an application to the controller, such as the number of stages and the number of input elements. The allocation and deallocation of RCs between cache memory mode and function unit mode, and the corresponding scheduling, are also the controller's responsibility. The controller is initiated and terminated by two new special instructions added to the conventional instruction set architecture (ISA). These instructions make the RC functions active and inactive in a code sequence. The data caches to hold the input and the intermediate results are also allocated by the host processor initially. The controller establishes the connections between the reconfigurable cache and the data caches with a dynamic bus architecture. The addresses for input, intermediate, and output data are produced by an address generator in the controller. These addresses are sequential within the respective cache units in regular computations. The controller also monitors the computation and initiates the next step when the current step is completed.

D. Compiler Issues

These reconfigurable caches can be employed under compiler control by adding new instructions, such as *FIR(parameters)/DCT(parameters)* or *rfu cache #, other parameters*. The first approach fixes *a priori* the functions supported by the architecture. The configuration in this case can be stored in the system memory as a part of the operating system initialization. These configurations can then be fetched by the microarchitecture in response to an instantiation of these instructions. The second approach *(rfu)* is more extensible through compiler analysis, in as much as it allows the compiler to map any suitable function on RCs. The configuration for a set of functions is predefined/precomputed (which is an input to the compiler in its target description). In another case, the compiler could be responsible for generating the configurations, which allows even more flexibility. The configuration data generated in either case (precomputed or compiler-driven) is based on RC framework (structure), such as the number of LUTs, the width of LUTs, the size of RC, and the interconnection.

IV. Experimental Results

We have experimented with two applications, convolution and DCT/IDCT. In this section, we describe how we map the applications onto RC. First, we map each application into RC separately; then we merge two applications into a single RC. We also compare the overall area of separated RCs and a combined RC in Section IV-B. Next, we compare the execution time of these applications on RCs with the execution time on a general-purpose processor (GPP) in Section IV-C.

A. Experimental Setup

1) Convolution (FIR Filter): A reconfigurable cache to perform a convolution function is presented in this section. The number of pipeline stages for the convolution in a reconfigurable cache depends upon the size of a cache to be converted. Our simulation is based on an 8-KB size cache with 128 bits per block/16-bit-wide words implementing four input LUTs with 16-bit output. A conventional convolution algorithm (FIR) is shown in the following:

$$y(n) = \sum_{k=0}^{M} w(k)x(n-k). \tag{1}$$

One stage of convolution consists of a multiplier and an adder. In our example, each stage is implemented by an 8-bit constant coefficient multiplier and a 24-bit adder to accumulate up to 256 taps in Fig. 6(a). The input data are double pipelined in one stage for the appropriate computation [6]. An 8×8 constant coefficient multiplier can be implemented using two 4×8 constant coefficient multipliers and a 12-bit adder with appropriate connections [17]. A 4×8 constant coefficient multiplier is implemented using twelve 4-LUTs with single output from each LUT on FPGAs. In our implementation, we split the 12-bit-wide LUT contents of a 4×8 conventional constant coefficient multiplier into two 16-bit output 4-LUTs (part 1, 2) with 6-bit-wide multiple outputs for a lower routing complexity of the interconnections, as shown in Fig. 6(b). The first six bits of each content are stored in LUT part1, while the last six bits are stored in LUT part2 to realize a 4×8 constant multiplier.

The concept of a carry select adder is employed for an addition using the LUTs described in Section II-A. Therefore, we need a 6-bit-wide result for a 2-bit addition, three bits when carry-in$=0$ and three bits when carry-in$=1$ from an LUT. An n-bit adder can be implemented using $\lceil n/2 \rceil$ such LUTs and the carry propagation scheme. The output is selected based on the input carry.

One stage of convolution can be implemented with 22 LUTs. To keep the number of LUT rows a power of two for cache operation, we put six LUTs in each LUT row and have four LUT rows to use 22 (out of 32) required LUTs. The final placement of LUTs is shown in Fig. 6(b). A few LUTs in the figure are

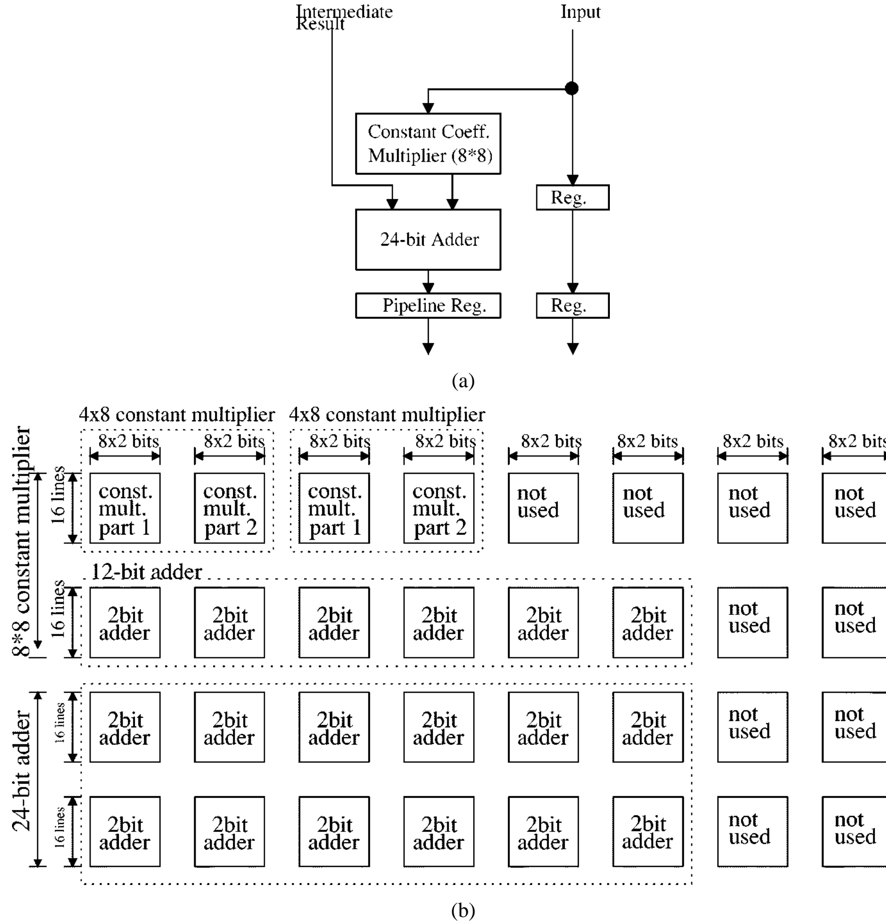

Fig. 6. (a) One stage of convolution and (b) array of LUTs for one stage of convolution.

not used for the computation. In Fig. 6(b), pipeline registers placed between stages and interconnections for LUTs are not shown. For an 8-KB reconfigurable cache, we have 32 rows of LUT that can be used to implement eight taps of the convolution algorithm.

2) DCT/IDCT (MPEG Encoding/Decoding): In this section, we show another reconfigurable cache module to perform a DCT/IDCT function, which is the most effective transform technique for image and video processing [25]. To be able to merge the convolution and DCT/IDCT functions into the same cache, we have implemented DCT/IDCT within the number of LUTs in the convolution cache module.

Given an input block $x(i,j)$, the $N \times N$ 2-D DCT/IDCT in [25] is defined as

$$X(u,v) = \frac{2}{N} C(u)C(v) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} x(i,j) \times \cos\frac{(2i+1)u\pi}{2N} \cos\frac{(2j+1)v\pi}{2N} \tag{2}$$

$$x(i,j) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u)C(v)X(u,v) \times \cos\frac{(2i+1)u\pi}{2N} \cos\frac{(2j+1)v\pi}{2N} \tag{3}$$

where $x(i,j)$ $(i,j = 0,\ldots,N-1)$ is a matrix of the pixel data, $X(u,v)$ $(u,v = 0,\ldots,N-1)$ is a matrix of the transformed coefficients, and $C(0) = (1/\sqrt{2})$, $C(u) = C(v) = 1$ if $u,v \neq 0$.

This $N \times N$ 2-D cosine transform can be partitioned into two N-point one-dimensional (1-D) transforms. To complete a 2-D DCT, two 1-D DCT/IDCT processes are performed sequentially with an intermediate storage. By exploiting a fast algorithm (the symmetry property) presented in [18] and [25], an $N \times N$ matrix multiplication for the $N \times N$ 2-D cosine transform defined in (2) and (3) can be partitioned into two $(N/2) \times (N/2)$ matrix multiplications of 1-D DCT/IDCT with additions/subtractions before the DCT process and after the IDCT process.

The 1-D DCT/IDCT process is an MAC, which can be represented as $y = \Sigma_{i=0}^{N-1} a_i x_i$. Although an MAC is already built in the reconfigurable cache in Section IV-A1, the distributed arithmetic [23] instead is employed in the RC for the DCT/IDCT function to avoid the run-time reconfiguration of coefficients required for the coefficient multiplier in FIR. Using this scheme, once the coefficients are configured into the RC, no more run-time reconfiguration is required.

The inner product of each 1-D transform (MAC) can be represented as follows:

$$y = \sum_{i=0}^{N-1} a_i x_i = \sum_{i=0}^{N-1} a_i \left(-b_{i0} + \sum_{r=1}^{W_d-1} b_{ir} 2^{-r} \right) = \sum_{r=1}^{W_d-1} \left[\sum_{i=0}^{N-1} a_i b_{ir} \right] 2^{-r} + \sum_{i=0}^{N-1} a_k(-b_{i0}) \tag{4}$$

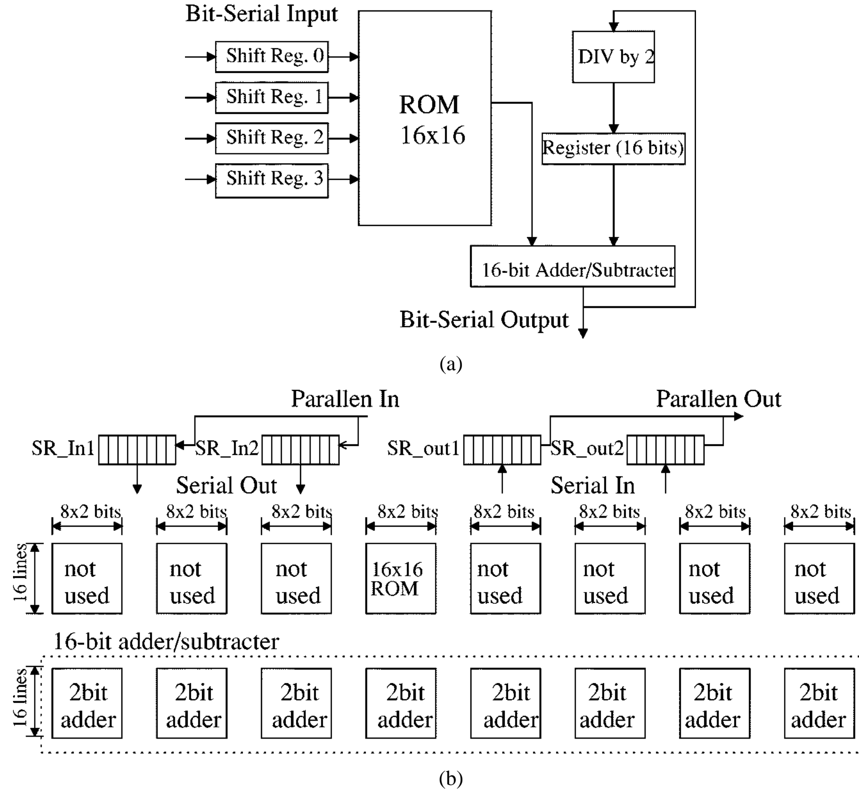

Fig. 7. (a) A DCT/IDCT processing element. (b) Array of LUTs for DCT/IDCT processing element with the input registers.

where $x_i = -b_{i0} + \Sigma_{r=1}^{W_d-1} b_{ir}2^{-r}$ with two's-complement form of an input word length W_d and a_i $(i = 0, 1, 2, \ldots, N-1)$ is the weighted cosine factors. According to (4), the multiplication with the coefficients can be performed with an ROM containing 2^N precalculated partial products $(\Sigma_{i=0}^{N-1} a_i b_{\mathrm{in}})$ in a bit-serial fashion. The inner product computes the sums of partial products corresponding to the same order bit from all the input elements processed in the current stage using a set of serial shift registers. For the output of the inner product, one more shift register is required. Therefore, one PE contains a ROM and a shift accumulator for the partial summations of corresponding data bit order. In this configuration, each inner product is completed in the number of clock cycles that is the same as the word length of input. With N PEs, N-point DCT can be completed in parallel. Using the symmetry property, the contents of a ROM can be reduced by $2^{N/2}$. However, it requires two sets of $N/2$ adders and $N/2$ subtracters before the DCT process and after the IDCT process.

Due to the coding efficiency and the implementation complexity, a block size of 8×8 pixels is commonly used in image processing [19]. We therefore have implemented an 8×8 2-D DCT/IDCT function unit by two sequential 1-D transform processes. In addition, the width of input elements is 8 bits. We also select the word length of the coefficients to be 16 bits for the accuracy of the DCT computation.

One PE with conventional architecture is depicted in Fig. 7(a). One PE implemented in the reconfigurable cache is depicted in Fig. 7(b). In the figure, the ROM is placed in the middle of an LUT row to reduce the number of routing tracks. In the given cache size, 8 KB, eight such PEs and the additional adders/subtracters for pre/postprocessing can be implemented. To make the DCT/IDCT implementation compatible with the convolution function unit, we place 4-LUTs with 16-bit output in an 8-KB sized cache. Only 20 LUT rows (16 for PEs and four for pre/postprocessing) out of 32 LUT row in the 8-KB cache are used for the implementation. However, the LUTs not used in this function still remain in the RC module for the compatibility with other functions.

A 16-bit carry select adder is configured as a shift accumulator with the registers not shown in the figure for the self-accumulation in each PE. According to (4), only one subtraction is necessary. This is done by the same adder, which can keep both addition and subtraction configurations in 12-bit data width (6 bits for adder and 6 bits for subtracter). The adder-subtracter shares the same input and output with the adder without requiring any extra logic. However, an extra control signal is needed to enable the subtraction. The additional adders and subtracters for the pre/postprocessing are implemented using the scheme for adder-subtracter described above since each pair of addition/subtraction needs the same input elements. In addition, the 1-bit shift of accumulated data can be easily done by appropriate connections from the registers to the input data lines of the adder. The input/output shift registers are added only to the in/out port of the actual DCT/IDCT function unit after the preprocessing unit and before the postprocessing unit. This means that only one set of shift registers are necessary since all the PEs compute using 4 bits out of the same set of input data in each transform of a row or a column.

In the actual implementation, we add one more set of shift registers to remove any delay due to loading or storing in/out data from other memories. All the loading/writing back from/to the storage can be overlapped with the computation cycle time

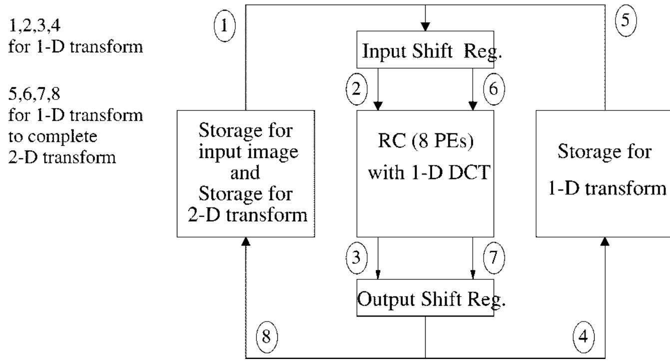

Fig. 8. Data flow of the computation process for 8 × 8 2-D DCT transform.

in PEs by appropriate multiplexing of the dual shift registers. Adding shift registers allows in/out data to be ready to be processed and written immediately after the previous computation, without any idle time. The controller described in Section III-C handles and controls the computation procedure. Once the host processor passes the required information to the controller, all the control signals are sent by the controller.

The computation process of an 8 × 8 2-D DCT is as follows. The function unit on the RC computes the 1-D transform for an entire row by broadcasting a set of input data after the preaddition/subtraction process to eight PEs in eight time units in a bit-serial fashion (i.e., a half set of data to four PEs and another half set of data to the other four PEs). A set of bit serial output from eight PEs is carried out to the output shift registers in the same fashion. The eight global bit lines described in Section II-C are used as input and output data lines. To avoid the delay of the global lines for the cache operations due to additional switches, we can place other routing tracks into the space between global bit lines, such as feedthrough. Since we have already added one additional metal layer for the global bit lines, this layer can be used to route additional lines. This implies that we have enough vertical routing tracks in this architecture. This computation is repeated eight times, once for each row, for eight rows of an 8 × 8 image. In the mean time during each computation, the next set of input data is fetched in another set of input registers and the previous output data is written into an additional memory. All the intermediate results from the 1-D transform must be stored in a memory and then loaded for the second 1-D transform, which performs the same computations to complete a 2-D transform. Therefore, 2-D DCT/IDCT is computed with two additional memories similar to the convolution function. A data flow diagram of the computation process for 8 × 8 2-D DCT is depicted in Fig. 8.

Several other opportunities for reconfigurable units exist in this architecture, as described later. Although the width of coefficients in the ROM configuration is fixed at 16 bits in this example, the coefficient width is flexible in this architecture between one and 16. Moreover, the width of input elements can be easily extended by adding more shift registers without modifying the current configuration. The emulation of an ROM in the RC does not imply fixed processing coefficients. Hence, different sets of coefficient values can be loaded using the conventional cache operation for the other distributed arithmetic operations.

3) Reconfigurable Cache Merged with Multicontext Configurations: Since we implement convolution and DCT/IDCT in the same reconfigurable cache framework, we can merge the two functions into one reconfigurable cache. With the concept of multicontext configurations mapped into multibit output LUTs and individual interconnections, the reconfigurable cache can be converted to either of two function units. A combined reconfigurable cache with two functions takes less area than the sum of the areas of two individual function units because the additional area cost is due to interconnections only. The logic is absorbed in the available cache memory-based LUTs. The required interconnection for each function is placed independently together in the combined reconfigurable cache, which implies that there is no sharing of interconnection between the two functions. As described in Section II-B, we use fixed interconnection since it takes less area and propagation time than the programmable interconnection. The actual area of the reconfigurable cache framework (base array cache) and interconnection is shown in the last part of Section IV-B.

B. Area

To measure the actual area overhead of cache array only for both *memory cell array cache* and *base array cache,* we experimented with layouts of the reconfigurable cache with only convolution, only DCT/IDCT, and both functions. As we compare the access time of RC for cache operations in two cache models, the *memory cell array cache* and the *base array cache* in Section II-D, the area overheads are estimated with respect to the two cache models.

According to our layout experiment, the total area of the reconfigurable module including the pipeline registers with an FIR filter, which supports up to 256 taps, is 1.53/1.12 times the area of data array in the memory cell array cache/base array cache without other logic components, respectively (described in Section II-D). To see the exact area overhead of memory array only, we consider the area overhead of RC with respect to the base area of only the data cache array, which does not include the additional cache logic—specifically, row/column decoders, tag/status-bit part, and sense amplifiers. The *percentage* of RC area overhead would appear to be even lower had we inflated the base area by including the area for these logic components. However, the actual area overhead remains the same.

For the DCT/IDCT function unit on an RC, the required interconnection is fixed just as in the convolution cache module. In the DCT/IDCT function, no complicated routing is required and the number of LUT rows in the RC is less than that for the FIR filter, while the number of registers is higher. Thus, according to our experimental layout for DCT/IDCT, the total area of the DCT/IDCT module is 1.48/1.09 times the area of data array in the memory cell array cache/base array cache, respectively, including the accumulating registers and the shift register at the in/out port. Again, those basic units, such as row/column decoders, tag/status-bit part, and sense amplifiers, are not included in this comparison as mentioned above.

In Tables I and II, the area overhead of FIR filter and DCT/IDCT in the RC is compared with designs for these functions previously reported in the literature. The designs we

TABLE I
AREA COMPARISON OF FIR FILTERS AND RC OVERHEAD FOR FIR

	[20]	[21]	[22]
Number of Taps	64	40	15,19
Coefficient Word-length	14 bits (fixed)	12 bits (programmable)	8 bits (fixed)
Technology	$0.8\mu m$ BiCMOS	$0.9\mu m$	$1.2\mu m$
Core Area	**49** mm^2	**22** mm^2	**80** mm^2
	FIR Filter in the RC (area overhead of the cache)		
Number of Taps	256 taps (with 8 physical taps)		
Coefficient Word-length	8 bits		
Technology	$0.8\mu m$	$0.9\mu m$	$1.2\mu m$
Area Overhead[1]	**11.28** mm^2	**14.28** mm^2	**25.39** mm^2
Area Overhead[2]	**3.45** mm^2	**4.37** mm^2	**7.77** mm^2
% for area overhead[1]	53% of area for array-only of the memory cell array cache		
% for area overhead[2]	12% of area for array-only of the base array cache		

[1] Area overhead of the "only" memory cell array
[2] Area overhead of interconnections and registers regarding the "only" base cache array described in II-D

TABLE II
AREA COMPARISON OF CHIPS AND RC OVERHEAD FOR DCT/IDCT

	[23]	[24]	[25]
Function	1-D IDCT	8×8 DCT/IDCT	8×8 DCT/IDCT
Technology	0.6 μm	$0.8\mu m$	$0.8\mu m$
Core Area	**9.4** mm^2	**10** mm^2	**21.21** mm^2
	8×8 DCT/IDCT in RC (area overhead of cache)		
Technology	$0.6\mu m$	$0.8\mu m$	
Area Overhead[1]	**5.9** mm^2	**10.5** mm^2	
Area Overhead[2]	**1.51** mm^2	**2.68** mm^2	
% for area overhead[1]	48% of area for array-only of the memory cell array cache		
% for area overhead[2]	9% of area for array-only of the base array cache		

compare with this paper are from the literature of the last ten years. We compare our result to the best area implementations in the literature. We have estimated the area for an RC in λ^2 by scaling λ from 0.25 to 12 μm (some of them are not shown in the tables). As we explained in Section II-D, the area overhead for RC is also estimated with respect to the two base cache architectures. For the memory cell array cache, the area overhead includes the dedicated decoders, switches, RC-interconnect, and required registers while the area overhead in the base array cache with the parallel decoding and segmented bit/word lines consists of only the interconnect and the registers. For a fair comparison, only the core sizes are listed in both tables by estimating the area of the core part of the entire chips. Also, the area overhead of RC in the tables is the area of only additional units to support the functions implemented. In other words, the original cache area is not included in the area overhead. The core

TABLE III
AREA COMPARISON OF MULTIPLIER-ACCUMULATORS AND RC OVERHEAD FOR ONE MAC STAGE

	Izumikawai et al. [26]	Lu-Samueli [27]
Size of In/Out	16b×16b/32bits	12b×12b/27bits
Technology	$0.25\mu m$	$1.0\mu m$
Area	**0.55** mm^2 (core)	**9.30** mm^2 (chip)
	MAC in the RC (area overhead)	
Size of In/Out	16b×16b/32bits	
Technology	$0.25\mu m$	$1.0\mu m$
Area Overhead[1]	**0.28** mm^2	**4.41** mm^2
Area Overhead[2]	**0.08** mm^2	**1.35** mm^2
% for area overhead[1]	51%	
% for area overhead[2]	11%	

area of design in [22] shown here is estimated in [20]. In Table II, the core area of 1-D IDCT in design [23] excludes I/O pads and buffer area. We scale the reported total area by the proportion of the reported core area to the reported total area. The area of FIR filter and DCT/IDCT in the RC includes all the required registers such as pipeline registers for FIR and accumulating/shift registers for DCT/IDCT.

Most of the reported FIR filter designs have fixed coefficients with as many physical MACs as the number of taps. Although coefficients are programmable in [21], only 40 taps can be supported for various types of filter. Besides, the time taken for run-time reconfiguration in a serial fashion is high due to the limited number of pins. The time of run-time reconfiguration of coefficients in the RC is much smaller because multiple LUT writes are achieved per cache write operation. For a fair comparison, the area per tap can be calculated roughly in each filter by dividing the core area by the number of taps. According to the area per tap, the area of a tap in the RC is larger than others with respect to the memory cell array cache while the area per tap in the RC is smaller than others with respect to the base array cache area overhead. Although only eight taps are implemented physically in the RC, the FIR cache module can support up to 256 taps with fast configuration not visible to the application.

Since some of the filters have a different word length, we compare the area of 16 × 16 constant coefficient multiplier and 32-bit accumulator (MAC) implemented in the RC with the same word length of MAC, as presented in [27]. Since constant coefficient multipliers are used in most DSP and multimedia applications, we implemented a 16 × 16 constant coefficient multiplier, one MAC stage for FIR. The MAC area is estimated based on the number of LUT rows used and interconnection in RC. In our experimental layout, the MAC (16 × 16) area in the RC is less than or equal to two times the area of one MAC stage of convolution (8 × 8) in the RC. This area is smaller than that of the existing MACs, as shown in Table III in both cases. This implies that an FIR filter with 16-bit word length can be easily implemented in the RC with a similar area overhead for four physical taps. However, it can still support up to 256 taps.

Note that the designs reported in [20] and [21] implement FIR with 14-bit and 12-bit coefficients, respectively, while we report RC area overhead for an 8-bit coefficient design. It is hard to develop a precise analytical model for area parameterized by

the number of bits in a coefficient. Some parts (such as multiplier) may scale nonlinearly with the number of coefficient bits depending on the algorithm. Some parts would scale sublinearly, such as control and global routing. For an approximate comparison, we assume that the area scales linearly in coefficient width. Hence, the 8-bit version of [20] would take area 28 mm^2 ($\frac{8}{14} \times 49$ mm^2), and for [21], the area would be 14.7 mm^2 ($\frac{8}{12} \times 22$ mm^2), which are comparable to the RC area overhead for FIR. The main advantage of RC for FIR is the reconfigurability, which allows the RC-FIR to have a virtually infinite number of taps unlike other customized FIR chips, with faster reconfiguration of taps. We therefore conclude that the area per tap for RC is comparable to that of the customized FIR chips.

The area of the previous designs for DCT/IDCT in Table II is larger than the proposed DCT/IDCT cache module except [24] with respect to area overhead of the memory cell array cache. The 2-D DCT/IDCT functions are implemented with a similar procedure as in the DCT/IDCT cache module—two 1-D DCT steps. Since the DCT function is implemented using a hardwired multiplier in [24], the area is smaller than the cache module with respect to the area overhead of the memory cell array cache. However, the area overhead with respect to the base array cache is smaller than all the previous designs shown in the table. The DCT function in [25] has two 1-D DCT units, so the area of one 1-D DCT unit is roughly half of the overall area, which is still larger than the RC overhead.

In the combined multifunction reconfigurable cache, each function needs a fixed interconnection topology. Therefore, the total area of interconnection occupied by the two functions in the combined RC is the sum of the individual interconnection areas for convolution and DCT/IDCT. According to our experimental layout of the combined cache, the total area of the RC with two functions is 1.63/1.21 times the area of data array in the memory cell array cache/base array cache, respectively, with all the required registers and without other components described above.

Since the decoders for LUTs account for most of the area overhead in the reconfigurable caches, adding more interconnection does not add much area in the combined RC. The actual area of the combined cache module is shown in Table IV. The base array cache described in Section II-D consists of dedicated 4-to-16 decoders, four address lines, and a number of switches to connect the local bit lines to the global bit lines. The area of combined reconfigurable cache is smaller than the sum of smallest areas in the existing FIR and DCT/IDCT function units in both cache models. This implies that we can add additional multiple functions in the existing reconfigurable cache with a relatively small area overhead. The interconnection area for individual functions is also listed in Table IV. Moreover, since some part of the area for routing tracks between the two functions is overlapped—for example, adders, constant multiplier, and ROMs—the area of interconnection in the combined RC may be less than the sum of two individual interconnection areas. The fixed interconnection for the functions can be efficiently routed and does not take much area. The placement and routing of the reconfigurable cache has been done manually as a first cut. We can expect the area overhead to reduce further if we place and route carefully.

TABLE IV
AREA OVERHEAD OF THE COMBINED RECONFIGURABLE CACHE

Function	FIR, DCT/IDCT	
Technology	0.6μm	0.8μm
Interconnection & registers FIR	**1.94** mm^2	**3.45** mm^2
Interconnection & registers DCT/IDCT	**1.51** mm^2	**2.68** mm^2
RC framework (base array cache)	**4.41** mm^2	**7.83** mm^2
Area Overhead[1]	**3.45** mm^2	**6.13** mm^2
Area Overhead[2]	**7.86** mm^2	**13.96** mm^2
Technology	1.0μm	1.2μm
Interconnection & registers FIR	**5.39** mm^2	**7.77** mm^2
Interconnection & registers DCT/IDCT	**4.19** mm^2	**6.04** mm^2
RC framework (base array cache)	**12.24** mm^2	**17.62** mm^2
Area Overhead[1]	**9.58** mm^2	**13.81** mm^2
Area Overhead[2]	**21.82** mm^2	**31.43** mm^2
% for area overhead[1]	63%	
% for area overhead[2]	21%	

C. Execution Time

1) Convolution: We compare the execution time of the FIR filter using an RC to a conventional GPP running the algorithm in (1). Since the reconfigurable cache may have to be flushed, we show the results for the following two cases. In the first case, no data in the cache needs to be written back to main memory before it is reconfigured as the function unit, for example, caches with write-through policy. In the second case, the processor has to flush all the data in the cache before configuring it (i.e., written back to the main memory). The extra time is denoted by the *flush time* and is required for write-back caches.

The total execution time of the convolution in the reconfigurable cache consists of configuration and computation times. The configuration time includes the times for adder and constant coefficient multiplier configuration. In addition, in the second case, the cache flush time is also added to the configuration time. The actual parameter values to compute the times are given in Table V. We chose the values to be as conservative as possible with respect to SPARC IIi processor cycle time at 270 MHz [28] (where the GPP simulation was performed). The access time for the data cache in a SPARC IIi processor is one cycle in a pipelined fashion (it is a 16-KB direct mapped cache with two 16-B subblocks per line). In a typical processor, this access time can be anywhere from one to two cycles. Hence, we chose three cycles for the cache access time in RC for a conservative model. Had we chosen a lower cache access time (one or two cycles), the RC execution time would appear to be even more favorable since other parameters, such as LUT read time in RC, were based on the cache access time—three cycles (12 ns). The main memory access time is 20 cycles. The parameters for the cache structure are based on an 8-KB size cache with eight words per block and 16 bits per word ($L_{\rm cache}, L_{\rm LUT}$, and W_n). Since eight words in a cache block are stored in the interleaved fashion, each bit of one word is stored every 8 bits. The first and ninth bits of an LUT content can be written in the LUT simultaneously by writing one word (parameter $m = 2$). The computation time of one stage/PE in the RCs is chosen by the

TABLE V
PARAMETERS FOR THE RCs

Parameter	Description	Values
T_{cpu}	1 cpu cycle time	4ns
T_{1_stage}	The time to complete the computation in one stage/PE	24ns/16ns
$R_{mem/cpu}$	Ratio of no. of cycles of 1 main memory access and 1 cpu cycle	20 (80ns)
$R_{cache/cpu}$	Ratio of no. of cycles of 1 cache memory access and 1 cpu cycle	3 (12ns)
L_{cache}	Number of cache lines in the cache	512
L_{LUT}	Number of contents in a LUT	16
W_n	Number of words per cache block	8
a	Number of bits required to configure a content of one LUT for a 2-bit adder with 3bits for carry=0 and 3bits for carry=1 & for the half of a 4x8 constant coefficient multiplier	6
r	Number of bits required to configure a content of LUT for a ROM	16
m	Number of bits to be written by one word when configuring	2 2
S	Number of taps/PEs implemented in the RC	8 8
	Parameters for Convolution	
TAP	Number of taps	8 - 256
X	Number of data	64 - 8192
	Parameters for DCT/IDCT	
W_d	The width of input elements	8 bits
N	The size of a basic block image	8
IMG	The size of an entire image	8x8 - 1920x1152

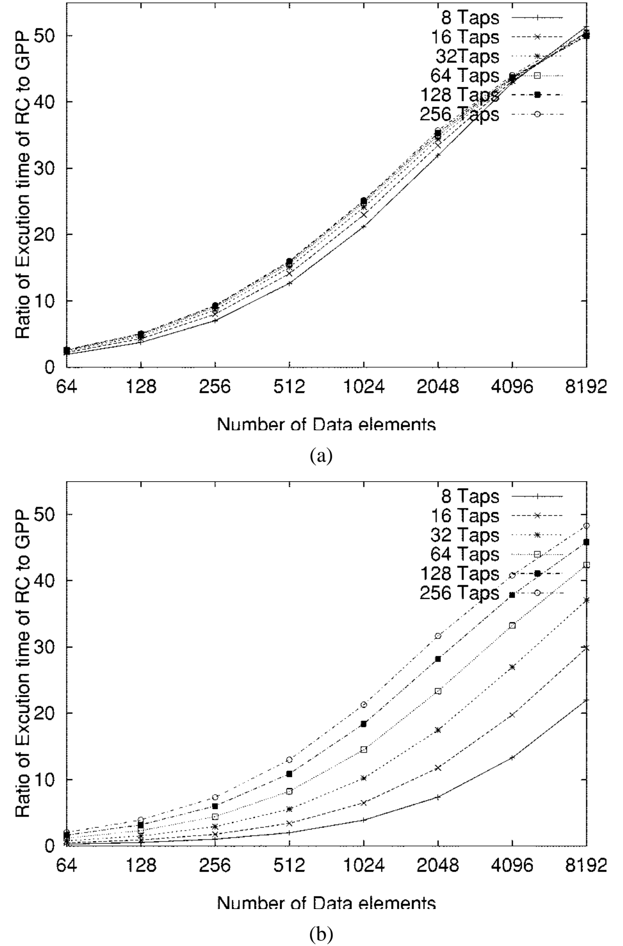

Fig. 9. Ratio of execution time of RC and GPP for convolution: (a) without memory flush and (b) with memory flush before converting into the function unit.

following factors. Each stage in the convolution function unit requires three LUT reads with additional time for propagation through a number of multiplexers, while each PE in DCT/IDCT unit does two LUT reads with additional time for multiplexers. We use read time for an LUT of 8 ns with the multiplexer propagation time—less than the cache access time because the LUT is much smaller and faster than the 8-KB cache memory. The expressions for the times are presented as follows:

1) ConFig. Time for adder $= [(R_{\text{mem/cpu}})(a/m)(L_{\text{LUT}}) + (R_{\text{cache/cpu}})(a/m)(L_{\text{cache}} - L_{\text{LUT}} \times S)] \times T_{\text{cpu}}$;
2) ConFig. Time for constant multiplier $= (R_{\text{mem/cpu}})(a/m)(L_{\text{LUT}})(\text{TAP}) \times T_{\text{cpu}}$;
3) Cache Flush Time $= (R_{\text{mem/cpu}})(W_n)(L_{\text{cache}}) \times T_{\text{cpu}}$;
4) Computation Time $= [(\text{TAP}/\text{S}) \times (X+2S-1)] \times T_{1_stage}$.

In the computation time, we add $2S$ instead of S for the initial pipeline steps because we exploit the double pipelined input data in each stage of the convolution, as shown in Fig. 6(a). In addition, we separate the configuration time for adders and multipliers. The reason for this is that only one set of data for an LUT is necessary when reconfiguring the LUTs for adders because the contents of all the LUTs are the same, while different configuration data are necessary for multipliers. The time for storing and loading input and intermediate data can be overlapped with the computation time. Therefore, data access time for the computation is not added.

The speedup of RC over GPP for convolution is shown in Fig. 9. We assume that all the input data fit into a data cache for both RC and GPP computations according to the following observation. We traced the number of cache misses in GPP for all the cases in Fig. 9. From the trace, we found that, regardless of the number of taps and data elements in the computation, the number of cache misses does not vary with the execution time. Therefore, we neglected the effect of the cache miss penalty in the comparison. We simulated convolution with floating-point variables instead of integers, which leads to faster processing in GPP. The choice between memory cell array and base array determines cache access time in GPP. As we explained in Section II-D, the RC based on memory cell array will give smaller access time in GPP even for other applications, while the RC based on base array will increase the cache access time by 1–2%. We have assumed the cache access time in GPP and in the processor with RC to be the same for both cache types (memory cell array or base array).

Our results show that the reconfigurable cache provides a better performance improvement than the GPP as the number of data elements increases. Fig. 9 shows that the performance improvement is almost independent of the number of taps without memory flush in Fig. 9(a). The ratio of the computation time with less taps decreases with memory flush in Fig. 9(b) because the flush time affects the ratio of the total execution time more with the decrease in the number of taps.

2) DCT/IDCT: As described in Section IV-A2, the 2-D DCT/IDCT can be completed by two 1-D transforms. This pro-

cedure is similar to the data caching scheme, which is adapted for the FIR filter module (i.e., two additional memories for processing with intermediate data). We compare the execution time of the 2-D transforms in RC and GPP executing the fast DCT algorithm described in Section IV-A2. As in the previous example, the two cases of cache *flush time,* no cache flush and cache flush, are considered in this section.

The total execution time of the DCT (IDCT) in the reconfigurable cache consists of configuration and computation times. The configuration time includes the writing times for the contents of ROMs and adders. In addition, in the case of cache flush, the cache flush time is also to be added in the configuration time. The actual parameter values to compute the times for this function used are the same as for the convolution in Table V. The expressions for the execution times are presented as follows:

1) ConFig. Time for accumulators and pre- (or post)adders/subtracters $= [(R_{\text{mem/cpu}})(2a/m)(L_{\text{LUT}}) + (R_{\text{cache/cpu}})(2a/m)((S+2) \times L_{\text{LUT}})] \times T_{\text{cpu}}$;
2) ConFig. Time for ROM $= [(R_{\text{mem/cpu}})(r/m)(S \times L_{\text{LUT}})] \times T_{\text{cpu}}$;
3) Cache Flush Time $= (R_{\text{mem/cpu}})(W_n)(L_{\text{cache}}) \times T_{\text{cpu}}$;
4) Computation Time $=$ [2 $\times$ (1-D transform)] $\times$ (Image size/Basic block size) $\times T_{1_stage} = [2 \times (N + W_d \times N)] \times (IMG/N \times N) \times T_{1_stage}$.

The cache *flush time* is the same as earlier. Configuration data need to be written to all the PEs once only because all the data elements in an image are processed with the same coefficients using the distributed arithmetic. The configuration procedure of the convolution in the previous section is applied to DCT/IDCT. As described earlier, the time of loading and writing all the in/out data from/to memories can be overlapped with the computation. Thus, only the initial loading and the final writing time, which is overlapped in the transition of data set, is added to the computation time of each 8 $\times$ 8 1-D transform for data access time. In this configuration, the adder is used as both a 16-bit adder and a 16-bit subtracter with two sets of configuration data. Since only one of the pre/postadders (subtracters) is necessary for DCT and IDCT, respectively, the configuration time of pre-(or post)adders/subtracter with the same configuration scheme is added in the execution time.

The speedup of RC over GPP for DCT is shown in Fig. 10. The assumption regarding the cache misses of data mentioned in Section IV-C1 has been applied to this simulation. Therefore, the main memory access time is not considered for in/out data of the computation. For a larger size of image than the basic block, 8 $\times$ 8, we partitioned the entire image into a number of basic block images. We assume that the cosine weighted factors are prestored as coefficients in an array when the GPP processes the DCT/IDCT, which means the actual cosine coefficient computation is not performed in GPP. It is much faster than the computation with the actual cosine factors. Again, floating-point variables are employed in our simulation of DCT/IDCT for faster processing in GPP.

According to the result in Fig. 10, the reconfigurable cache for DCT/IDCT has a better performance improvement over the execution time of the GPP as the size of input image increases. The performance improvement is roughly independent of the

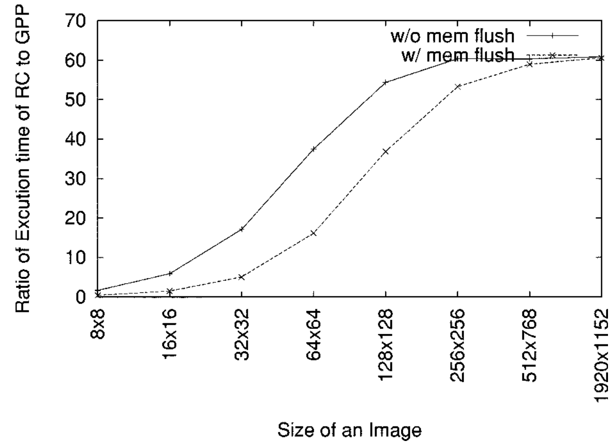

Fig. 10. Ratio of execution time of RC and GPP for DCT/IDCT with and without *flush time*.

memory flush in the larger size of images. Since the computation is ROM based, only the initial configuration is necessary. Thus, the larger sizes in the results, 512 $\times$ 768 (TV-image) and 1920 $\times$ 1152 (HDTV), do not rely on the flush time. For main profile at high level decoding, the maximum allowable time to process a macroblock is 4.08 μs [23]. The result shows that it is possible to process a block in 2.30 μs.

3) Multicontext Reconfigurable Cache: There is no difference between individual and combined caches in terms of the execution time. However, the combined cache may have a slightly higher propagation delay due to longer wires caused by the inclusion of interconnection, in our instance, 1.6% increase in cache access time. Therefore, we can assume that both individual and combined RCs have almost the same execution performance.

V. Conclusion

We have presented a reconfigurable cache module, which can perform both as a function unit and a cache. This allows a processor to trade compute bandwidth for I/O bandwidth. We have analyzed it for convolution and DCT/IDCT. The reconfigurable caches for the computation of convolution and DCT/IDCT improve the performance by a large amount (a factor of up to 50 and 60 for convolution and DCT/IDCT, respectively). The area penalty for this reconfiguration is about 50–60% of the memory cell cache array area with faster cache access time, and 10–20% of the base cache array area with 1–2% increase in the cache access time. However, we save 27% for FIR and 44% for DCT/IDCT in area with respect to memory cell array cache and about 80% for both applications with respect to base array cache if we were to implement all these units separately. We are currently developing similar mappings for other structured functions. Pseudoprogrammable interconnection with limited programmability, but with less area overhead, to support more general functions—a certain family of applications—is also being considered. Although we propose integrating the reconfigurable cache modules within Level-1 caches, these RC modules can also be used at Level-2 cache. Architecturally, Level-2 integration would be easier providing us with "Active pages" type of capability [12].

REFERENCES

[1] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, Apr. 1997.

[2] A. DeHon, "DPGA-coupled microprocessor: Commodity ICs for the early 21st century," in *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, D. A. Buell and K. L. Pocek, Eds., CA, Apr. 1994, pp. 31–39.

[3] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. 27th Annu. Int. Symp. Microarchitecture*, Nov. 1994, pp. 172–180.

[4] A. Tyagi, "Reconfigurable memory queues/computing units architecture," in *Proc. Reconfigurable Architecture Workshop 11th Int. Parallel Processing Symp.*, Apr. 1997.

[5] S. Hauck *et al.*, "The Chimaera reconfigurable functional unit," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1997.

[6] S. Cadambi *et al.*, "Managing pipeline-reconfigurable FPGAs," in *Proc. ACM/SIGDA 6th Int. Symp. Field Programmable Gate Arrays*, Feb. 1998.

[7] Xilinx Inc., Virtex 2.5 V field programmable gate arrays datasheet, .

[8] A. DeHon, "Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization)," in *Proc. FPGA'99*, Feb. 1999.

[9] S. K. Raman, V. Pentkovski, and J. Keshava, "Implementing streaming SIMD extensions on the Pentium III processor," *IEEE Micro*, vol. 20, no. 4, July/Aug. 2000.

[10] P. Ranganathan, S. Adve, and N. P. Jouppi, "Performance of image and video processing with general-purpose processors and media ISA extensions," in *Proc. 26th Int. Symp. Computer Architecture (ISCA'99)*.

[11] P. Soderquist and M. Leeser, "Memory traffic and data cache behavior of an MPEG-2 software decoder," in *Proc. 1997 Int. Conf. Computer Design*, Oct. 1997.

[12] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: A computation model for intelligent memory," in *Proc. 25th Int. Symp. Computer Architecture (ISCA'98)*.

[13] A. Singhal, "Reconfigurable cache module architecture," M.S. thesis, Dept. of Computer Science, Iowa State University, Ames, May 2000.

[14] Wafer Electrical Test Data and SPICE Model Parameters, .

[15] S. J. E. Wilton and N. P. Jouppi, "CACTI: An enhanced cache access and cycle time model," *IEEE J. Solid-State Circuits*, vol. 31, pp. 677–688, May 1996.

[16] D. Deshpande, A. K. Somani, and A. Tyagi, "Configuration scheduling schemes for striped FPGAs," in *Proc. FPGA'99*, Feb. 1999, pp. 206–214.

[17] M. Wojko and H. ElGindy, "Self configuring binary multiplier for LUT addressable FPGAs," in *Proc. 5th Australasian Conf. Parallel and Real-Time Systems*, 1998.

[18] W. H. Chen, C. H. Smith, and S. V. Fralick, "A fast computational algorithm for the discrete cosine transform," *IEEE Trans. Commun.*, vol. COM-25, pp. 1004–1009, Sept. 1977.

[19] M. Maruyama *et al.*, "VLSI architecture and implementation of a multifunction, forward/inverse discrete cosine transform processor," in *Proc. Visual Communications and Image Processing '90*, pp. 410–417.

[20] T. Yoshino *et al.*, "A 100-MHz 64-tap FIR digital filter in 0.8-μm BiCMOS gate array," *IEEE J. Solid-State Circuits*, vol. 25, Dec. 1990.

[21] M. Hatamian and S. Rao, "A 100 MHz 40-tap programmable FIR filter chip," in *Proc. IEEE Int. Symp. Circuits Syst.*, 1990.

[22] S. Ishikawa *et al.*, "Automatic layout synthesis for FIR filters using a silicon compiler," in *Proc. IEEE Int. Symp. Circuits Syst.*, 1990.

[23] T. Masaki *et al.*, "VLSI implementation of inverse discrete transformer and motion compensator for MPEG2 HDTV video decoding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 5, Oct. 1995.

[24] A. Madisetti and A. N. Willson, "A 100 MHz 2-D 8 $\times$ 8 DCT/IDCT processor for HDTV applications," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 5, Apr. 1995.

[25] S. Uramoto *et al.*, "A 100 MHz 2-D discrete cosine transform core processor," *IEEE J. Solid-State Circuits*, vol. 27, pp. 492–499, Apr. 1992.

[26] M. Izumikawai *et al.*, "A 0.25-μm CMOS 0.9-V 100-MHz DSP core," *IEEE J. Solid-State Circuits*, vol. 32, pp. 52–61, Jan. 1997.

[27] F. Lu and H. Samueli, "A 200-MHz CMOS pipelined multiplier-accumulator using a quasidomino dynamic full-adder cell design," *IEEE J. Solid-State Circuits*, vol. 28, pp. 123–132, Dec. 1993.

[28] Sun Microsystems, UltraSPARC-IIi-User's manual, .

Huesung Kim (S'99) received the B.E. degree in electrical engineering from Kangnung National University, Kangnung, Kangwon, Korea, in 1996. He is currently pursuing the Ph.D. degree in the Departments of Electrical and Computer Engineering, Iowa State University, Ames.

His research interests include VLSI design and computer architecture, especially in microarchitecture and cache memory design.

Arun K. Somani (S'83–M'83–SM'88–F'99) received the M.S.E.E. and Ph.D. degrees in electrical engineering from McGill University, Montreal, QC, Canada, in 1983 and 1985, respectively.

He is currently David C. Nicholas Professor of Electrical and Computer Engineering at Iowa State University, Ames. From 1985 to 1997, he was a faculty member at the University of Washington, Seattle. His research interests are in the area of fault-tolerant computing, computer communication and networks, wireless and optical networking, computer architecture, and parallel computer systems.

Prof. Somani is a distinguished speaker and distinguished tutorial speaker of IEEE. He has served on numerous conference program committees and as general chair, program chair, and tutorial chair.

Akhilesh Tyagi (M'88) received the B.E. degree (Honors) in electrical and electronics engineering from Birla Institute of Technology and Science, Pilani, in 1981, the M.Tech. degree in computer engineering from the Indian Institute of Technology, New Delhi, in 1983, and the Ph.D. degree in computer science from the University of Washington, Seattle, in 1988.

He was on the Faculty of the Department of Computer Science at the University of North Carolina at Chapel Hill and Iowa State University, Ames. He has been with the Department of Electrical and Computer Engineering at Iowa State University since August 1999. His research interests in VLSI include: complexity theory, design, synthesis, and computer architecture.