A Reconfigurable Multi-function Computing Cache Architecture *

Hue-Sung Kim, Arun K. Somani, and Akhilesh Tyagi
Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50011
E-Mail: {huesung, arun, tyagi}@iastate.edu

ABSTRACT

A considerable portion of a chip is dedicated to a cache memory in a modern microprocessor chip. However, some applications may not actively need all the cache storage, especially the computing bandwidth limited applications. Instead, such applications may be able to use some 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. In this paper, we present a cache architecture to convert a cache into a computing unit for either of the following two structured computations, FIR and DCT/IDCT. In order to convert a cache memory to a function unit, we include additional logic to embed multi-bit output LUTs into the cache structure. Therefore, the cache can perform computations when it is reconfigured as a function unit. 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). In addition, the area overhead of the reconfigurable cache module for FIR and DCT/IDCT is less than the core area of those functions. Our simulations indicate that a reconfigurable cache does not take a significant delay penalty compared with a dedicated cache memory. 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 memories.

1. INTRODUCTION

The number of transistors on a chip has increased dramatically in the last decade. Within the next 5-10 years, we will have 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 at a time. Such applications result in low utilization of the cache memory. Moreover, sometimes this hardware can be utilized for computation in some applications.

Availability of a large number of transistors on a next generation processor chip has motivated several researchers to study the use of reconfigurable logic for on-chip coprocessors [1; 2; 3]. Such logic can accelerate the execution of applications by providing results to the host processor or storing them into a cache. Thus, it improves the performance of the applications and reduces the bottleneck of off-chip communications. For example, if the processor employs an FPGA (Field Programmable Gate Array) or coprocessors to accelerate applications, the need for data transfer between the processor and those coprocessors increases the requirement for communication bandwidth, and eventually results in a bottleneck.

In Garp architecture [1], programmable logic resides on processor chip to accelerate some computations in a conventional processor. These computations are expected to be used frequently in the architecture. If an application does not need the logic, these functions remain idle. PipeRench [4] tries to reconfigure the hardware every cycle to overcome the limitation of hardware resources. XC6200 [5] from Xilinx has a different architecture from the previous FPGA series to allow concurrent and partial reconfiguration. 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 us to design a reconfigurable module which works as a function unit as well as a cache memory. Our goal is to develop such reconfigurable cache-function unit modules to improve the overall performance with low area and time overhead. The expectation is that significant logic sharing between the cache and function unit would lead to relatively low extra logic for a reconfigurable cache. 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.

We first describe the concept of a multi-bit output LUT used in the proposed architecture in Section 2. Section 3 de-

*This work was funded by Carver Trust Grants, Iowa State University and NSF Grant #CCR-9900601.

scribes the architecture of a reconfigurable module with the function unit and cache operations. The configuration and schedule of the module are described in Section 4. Section 5 depicts experimental results on the reconfigurable module. We conclude the paper in Section 6.

2. MULTI-BIT OUTPUT LUTS

In most FPGA architectures, a Look-up table (LUT) usually has four inputs and one output to keep the overall operation and routing efficient. However, an SRAM-based single output LUT does not fit well with a cache memory architecture because of a large amount of overhead for the decoders in the cache with a large memory block size. Instead of using a single output LUT, we propose to use a structure with multi-bit output LUTs. Such LUTs produce multiple output bits for a single combination of inputs and are better suited for a cache than the single output LUTs. Since a multi-bit 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 4x2 constant coefficient multiplier (all need the same, up to 6-bit output, size of LUT) are depicted in Figure 1(a) and (b), respectively.

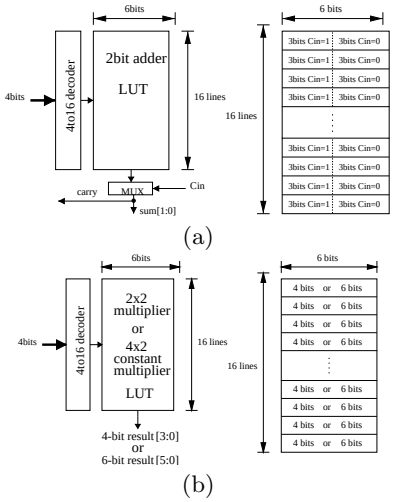


Figure 1: Multi-output LUTs : (a) A 2-bit adder ; (b) A 2x2 or a 4x2 constant coefficient multiplier

If a multi-bit output LUT is large enough for a computation, no interconnection (for example, to propagate a carry for an adder) may be required since all possible outputs can be stored into the large memory. In addition, unlike a single output LUT, a multi-bit 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, multi-bit 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 multi-bit output LUTs in Figure 1. Figure 2(a) depicts an 8-bit adder consisting of two 9-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, 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 time to finish an addition. By employing the concept of carry select adder as depicted in Figure 2(b), a faster adder using 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 the 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. $4 \times 2^8 \times 5 = 5120\ bits$).

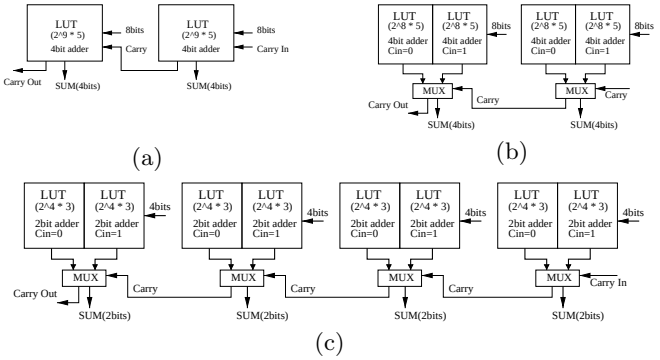


Figure 2: 8bit adder using (a) two 9-LUTs ; (b) two 8-LUTs; (c) four 4-LUTs

To make an area efficient adder, a 4-LUT with 6-bit outputs can be employed (Figure 2(c)). The same carry propagation scheme as in Figure 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 that for an 8-LUT since it has a smaller decoder and shorter data lines for memory reading. We, therefore, recommend the design in Figure 2(c) for adding subtract operations as well.

3. RECONFIGURABLE CACHE MODULE ARCHITECTURE

3.1 Overview of the processor with reconfigurable caches

In a reconfigurable cache module architecture (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 256KB to 1MB Level-1 data cache within next 5-10 years. Each cache module in our design is 8KB giving us 32 - 128 cache modules. A reconfigurable cache module can behave as a regular cache module or as a special purpose function unit as dedicated logic.

Figure 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. 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 dynam-

ically between the active cache modules with the use of address bound registers to specify the address range that is cached in that cache module. The details of this architecture are being developed in [6]. Through RCMA simulation on real multimedia programs, [6] 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 Figure 3 can be converted to function units, for example, to carry out functions such as FIR filter, DCT/IDCT (MPEG encoding/decoding function), Encryption, and General computation unit like a floating point multiplier, respectively. When one cache is used as a function unit (one of the above functions), 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 units for a function unit to provide input and receive output at the speed of the function unit. The configuration of the RCs is completed by the host processor because the processor has the information about cache mappings to take care of dynamic nature of cache functionality whether this reconfiguration would be interrupt driven or a parallel activity is an issue to be resolved.

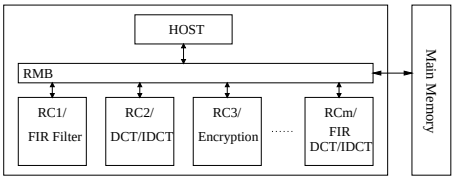


Figure 3: Overview of a processor with multiple reconfigurable cache modules

When a processor employs several such modules that can perform as caches as well as function units, then the data transfer between the processor and these modules increases the requirement for communication bandwidth. This eventually may result in another kind of bottleneck. We therefore adopt an efficient communication unit to satisfy the bandwidth needs. The modules can communicate with each other as well as the main memory and the host processor using the Reconfigurable Multiple Bus networks (RMB) [7]. If the simulations in [6] show that the inter-RC communication is highly localized, we may be able to get away with less expense, localized communication networks. Since reconfigurable modules can be converted into specific function units, interconnections for each function unit in the module are fixed to be a super set of the communication needs of the supported functions.

3.2 Organization and operation of a reconfigurable cache module

Since we target the applications that are compute-intensive and have a regular structure, we first partition them at coarse-level in which the basic computations are repeated regularly. A function in each level can be implemented using the multi-bit output LUTs as described in Section 2. We 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.

Figure 4 depicts a possible organization of the module. The cache can be viewed as a two-dimensional matrix of LUTs. Each LUT has 16 rows to support 4-LUT function and as many multi-bits in each row as required to implement a particular function. In the function unit mode, 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 mode, the least significant 4 bits of the address lines are connected to the row decoders dedicated to each LUT. The rest of 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 memory mode. Therefore, no matter what the value of the upper bits in the address is, 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.

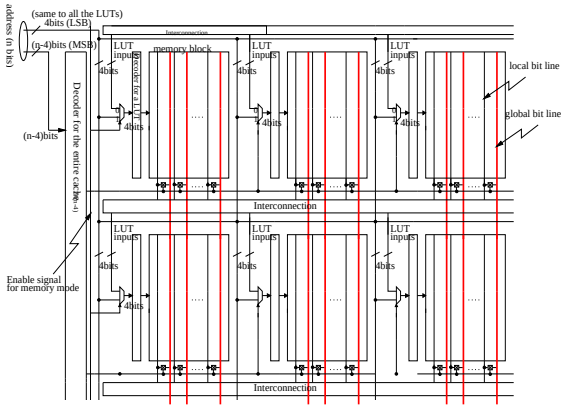


Figure 4: Cache architecture in the reconfigurable module

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 Figure 4. Thus, 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.

3.3 Access time for cache operations

Since each LUT, with its own row decoder for addressing in the reconfigurable module, is much smaller than a large synthesized memory block, the decoding time of a LUT is faster than the decoding time of the large memory block. Real dedicated caches may have a similar or more efficient parallel decoding structure with segmented bit lines. The

access time of reconfigurable cache is slightly slower than that of the real caches due to stretched bit lines caused by inserting the interconnections between LUT rows in the RC. Based on the SPICE model parameters for $0.5\mu m$ technology in [8] 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, the access time overhead due to the stretched line is about 1% of the overall cache access time. Since we assume a base cache with the parallel decoding and segmented bit lines, the area overhead of RC with respect to the base cache consists of only the interconnect area. The area overheads for FIR and DCT/IDCT function modules are given in Section 5.2.

4. CONFIGURATION AND SCHEDULING

4.1 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. Thus, the distance between two consecutive bits of a word is equal to the number of words in a block. However, for a LUT application, we need to use multi-bits in a single LUT. This implies that we cannot store one entry of a multi-bit output LUT by writing one word in a cache. For example, if a 4-LUT produces an n bit-wide output for a function, 16×n words are required to be written to the LUT in the cache. However, since other LUTs in the same row can also be programmed simultaneously, no more than 16×n words are required to fill up the contents of all LUTs in one row. In addition, if the width of a LUT is larger than the number of words in a cache block, multi-bit writing is performed into each LUT in a LUT row. This places a restriction that the width of a multi-bit 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 2. Since all cache structures are based on a power of 2, it is more convenient to make all LUT parameters (length and width) a power of 2 to avoid a complicated controller and an arbitrary address generator. This may result in under utilization 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.

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 either available in an on-chip cache or an off-chip memory. 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 through the cache data lines mentioned earlier.

4.2 Scheduling and controlling data flow

A cache module can also be used to implement a function which has more 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, $S1, S2, \ldots, S_k$, such that each set S_i can be realized by a cache module. If all S_i's are similar, then we can adapt data caching as described in [9] to store the partial results of the previous stage as input for the processing by the next configuration. The 'similar' here means that the LUT contents may change, but the interconnection between stages is the same. This happens, for example, in a 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 a data caching scheme, we place all input data in a cache and process them using the current configuration. At the end of that, the cache module is configured for the next step. 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 to be fed for all the computation, we have to keep the two caches as they are. 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, the number of input elements, data caches, and the reconfigurable cache. The data caches to hold the input and the intermediate results are also allocated as resources by the host processor. The controller establishes the connections between the reconfigurable cache and the data caches using a bus architecture like RMB. 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.

5. EXPERIMENTAL RESULT

We have experimented with two applications, Convolution and DCT/IDCT. In this section,we describe how we construct the applications onto the reconfigurable cache (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 5.2. Next, we compare the execution time of these application on RCs with the execution time on General-Purpose Processor (GPP) in Section 5.3. The main advantage of the RCs is on-chip processing, which implies faster processing time and no off-chip bottlenecks, and the balance/utilization of on-chip caches between storage and computation.

5.1 Experimental setup

5.1.1 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 8KB size cache with 128 bits per block/16-bit wide words implementing 4-LUTs with 16-bit output.

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 Figure 5(a). The input data is double pipelined in one stage for the appropriate computation [4]. An 8×8 constant coefficient multiplier can be implemented using two 4×8 constant coefficient multipliers and a 12-bit adder with appropriate connections [10]. A 4×8 constant coefficient multiplier is implemented using 12 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 Figure 5 (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.

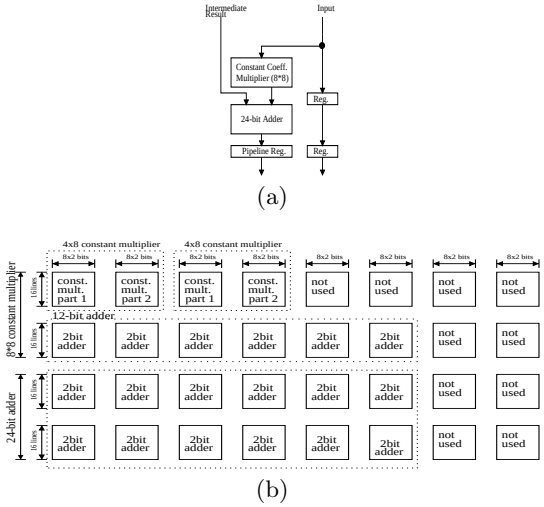


Figure 5: (a) One stage of Convolution; (b) Array of LUTs for one stage of convolution

The concept of a carry select adder is employed for an addition using the LUTs described in Section 2. Therefore, we need a 6-bit wide result for a 2-bit addition, three bits when carryin=0 and three bits when carryin=1 from a LUT. An n-bit adder can be implemented using $\lceil \frac{n}{2} \rceil$ 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. The final placement of LUTs is shown in Figure 5 (b). A few LUTs in the figure are not used for the computation. In Figure 5(b), pipeline registers and interconnections are not shown. For an 8KB reconfigurable cache, we have 32 rows of LUT which can be used to implement 8 taps of the convolution algorithm.

5.1.2 DCT/IDCT

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 [18]. 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×N 2-dimensional DCT/ IDCT in [18] is defined as

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

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

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

This N×N 2-D cosine transform can be partitioned into two N point 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], an N×N matrix multiplication for the N×N 2-D cosine transform defined in (1) and (2) can be partitioned into two N/2×N/2 matrix multiplications of 1-D DCT/IDCT with additions/subtractions before the DCT process and after the IDCT process.

In our implementation of the RC, the distributed arithmetic [16; 18] instead of multiply-and-accumulation (MAC) is employed for the DCT/IDCT function to avoid frequent reconfiguration of coefficients required for every input. 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.

$$\begin{aligned} y &= \sum_{i=0}^{N-1} a_i x_i = \sum_{i=0}^{N-1} a_i(-b_{i0} + \sum_{r=1}^{W_d-1} b_{ir}2^{-r}) \\ &= \sum_{r=1}^{W_d-1} \left[\sum_{i=0}^{N-1} a_i b_{ir}\right] 2^{-r} + \sum_{i=0}^{N-1} a_k(-b_{i0}) \end{aligned} \quad (3)$$

where $x_i = -b_{i0} + \sum_{r=1}^{W_d-1} b_{ir}2^{-r}$ with two's complement form of an input word length W_d and a_i (i = 0, 1, 2,..., N-1) is the weighted cosine factors. According to (3), the multiplication with the coefficients can be performed with a ROM containing 2^N pre-calculated partial products ($\sum_{i=0}^{N-1} a_i b_{in}$) in a bit-serial fashion.

One processing element (PE) contains a ROM and a shift accumulator for the partial summations of corresponding data bit order as shown in 6(a). In this configuration, each inner product is completed in the number of clock cycles that is the same as the word length of input elements. With N PEs, N-point DCT can be completed in parallel. Using the symmetry property the number of contents of a ROM can be reduced by $2^{\frac{N}{2}}$ with pre/post-adders and subtracters mentioned earlier.

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

The 2-D transform processed by two 1-D transforms requires two additional memories for input and intermediate

data. One PE implemented in the reconfigurable cache is depicted in Figure 6(b). In the figure, the ROM is placed in the middle of a LUT row to reduce the number of routing tracks. To make the DCT/IDCT implementation compatible with the convolution function unit, we place 4-LUTs with 16-bit output in an 8KB size of cache. Only 20 LUT rows (16 for PEs and 4 for pre/post-process) out of 32 LUT row in the 8KB cache are used for the implementation. However, the LUTs not used in this function still remain in the RC module for the compatibility.

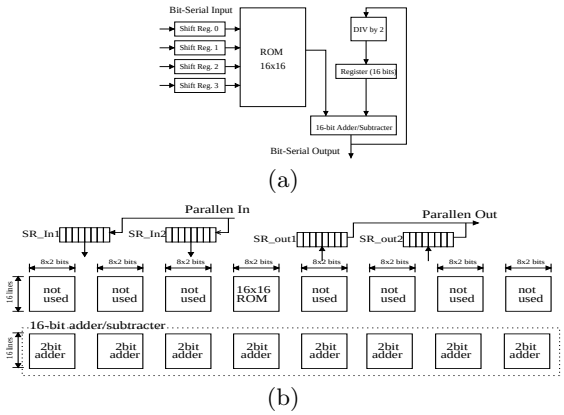


Figure 6: (a) A DCT/IDCT processing element; (b) Array of LUTs for DCT/IDCT processing element with the input registers

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 Equation (3), 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 (6bits for adder and 6bits 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/post-process 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 pre-process unit and before the post-process unit. This means only one set of shift registers are necessary since all the PEs compute using 4bits 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 in PEs by appropriate multiplexing of the dual shift registers. Adding shift registers makes 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 4.2 handles and controls the computation procedure.

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 pre-addition/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 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 3.2 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, adding more lines on the same layer along with the global bit lines does not affect negatively in the fabrication stage. This implies that we have enough vertical routing tracks in this architecture. This computation is repeated as many as 8 times, 8 rows for 8×8 images. 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, a 2-D DCT/IDCT is completed with two additional memories as the convolution function unit does.

5.1.3 Reconfigurable cache combined with multi-context configurations

Since we implement Convolution and DCT/IDCT in the same reconfigurable cache frame, it allows us to merge two functions into one reconfigurable cache. With the concept of multi-context configurations into multi-bit output LUTs and individual interconnection, the reconfigurable cache can be converted to either of the two function units when necessary. A combined reconfigurable cache with two functions takes less area than the sum of the areas of two individual function units because only interconnections are required to implement the functions. The required interconnection for each function is placed independently together in the combined reconfigurable cache. As described in Section 3.1, we use fixed interconnection since it takes less area and propagation time than those of programmable interconnection with a number of switches. The actual area of the reconfigurable cache framework and interconnection is shown in the last part of Section 5.2.

5.2 Area

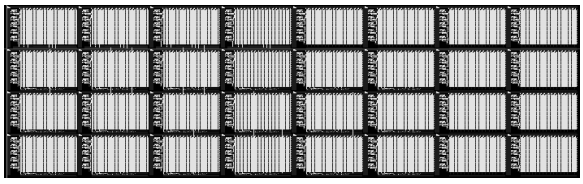


Figure 7: A possible layout of reconfigurable cache for one stage of the convolution

To measure the actual area overhead, we experimented with a possible layout of the reconfigurable cache with only Convolution, only DCT/IDCT and with both functions. Figure 7 represents one stage of convolution unit in the reconfigurable cache described above. The pipeline registers are not shown in the figure. According to this layout, the total

Table 1: Area comparison of FIR Filters and RC overhead for FIR

	Yoshino et al. [11]	Hatamian-Rao [12]	Ishikawa et al. [13]
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	**3.45** mm^2	**4.37** mm^2	**7.77** mm^2

Table 2: Area comparison of DCT/IDCT chips and RC overhead for DCT/IDCT

	Masaki et al. [16]	Madisettii-Willson [17]	Uramoto et al. [18]
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	**1.51** mm^2	**2.68** mm^2	

area of the reconfigurable module including the pipeline registers with an FIR Filter, which supports up to 256 taps, is 1.12 times the area of *the base cache memory array* descibed in Section 3.3. A normal cache operation requires additional hardware for row/column decoders, tag/status-bit part, and sense amplifiers not included in the base cache area.

For the DCT/IDCT function unit on an RC, the required interconnection is again fixed like 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 1.09 times the area of the *base cache memory array* including the accumulating registers and the shift register at the in/out port.

In Table 1 and Table 2, the area overhead of FIR Filter and DCT/IDCT in the RC is compared with designs for these functions previously reported in literature, respectively. As we explained in Section 3.3, the area overhead of the RC consists of the RC-specific interconnect and the required registers. Some of these designs include pads area. 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. The core area of design in [13] shown here is estimated in [11]. In Table 2, the core area of 1-D IDCT in design [16] 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 [12], 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. Although only 8 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. In addition, the area per tap in the RC is smaller than others.

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 presented in [14; 15]. Since constant coefficient multipliers are used in most DSP and multimedia applications, we implemented a 16×16 constant coefficient multiplier. 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 3. 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.

Table 3: Area comparison of Multiplier-Accumulator's and RC overhead for MAC

	Izumikawai et al. [14]	Lu-Samueli [15]
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	**0.08** mm^2	**1.35** mm^2

The area of the previous designs for DCT/IDCT in Table 2 is larger than the proposed DCT/IDCT cache module. The 2-D DCT/IDCT functions are implemented with a similar procedure as in the DCT/IDCT cache module - two 1-D DCT steps. Although the DCT function is implemented using hardwired multiplier in [17], the area is larger than the cache module. The DCT function in [18] 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 RC overhead.

In the combined multi-function 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 area for Convolution and DCT/IDCT. According to our experimental layout of the combined cache, the total area of the RC with two functions is 1.21 times the area of *the base cache memory array* with all the required registers.

Since the decoders for LUTs account for most of the area in the reconfigurable caches, adding more interconnection does not add area much in the combined RC. The actual area of the combined cache module is shown in Table 4. The base cache described in Section 3.3 consists of a dedicated 4-to-16 decoder, 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. This implies that we can add multiple functions in the existing reconfigurable cache with a small area overhead. The interconnection area for individual function is also listed in Table 4.

Table 4: Area overhead of the combined reconfigurable cache

Function	FIR, DCT/IDCT	
Technology	$0.6\mu m$	$0.8\mu m$
Interconnection & registers FIR	**1.94** mm^2	**3.45** mm^2
Interconnection & registers DCT/IDCT	**1.51** mm^2	**2.68** mm^2
Area Overhead	**3.45** mm^2	**6.13** mm^2
Technology	$1.0\mu m$	$1.2\mu m$
Interconnection & registers FIR	**5.39** mm^2	**7.77** mm^2
Interconnection & registers DCT/IDCT	**4.19** mm^2	**6.04** mm^2
Area Overhead	**9.58** mm^2	**13.81** mm^2

The fixed interconnection for the functions can be efficiently routed and does not take much area. The placement & routing of the reconfigurable cache has been done manually with CAD tools. We can expect the area overhead to reduce further if an automated algorithm realizing an optimal solution for the placement & routing is used.

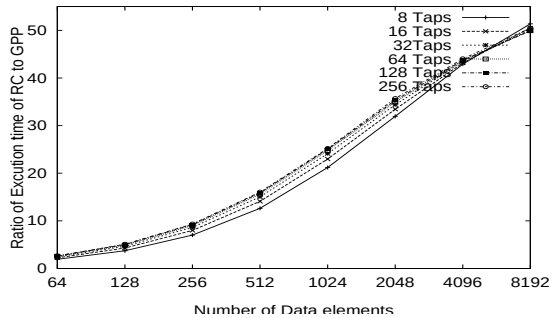
5.3 Execution time

5.3.1 Convolution

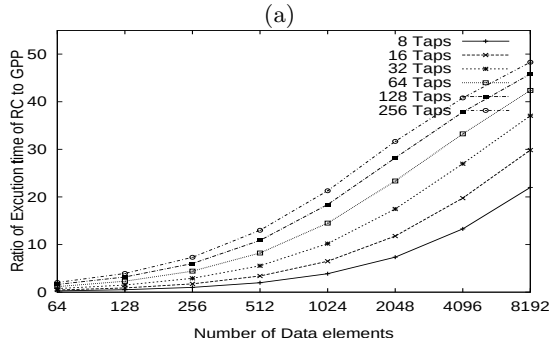
We achieved the following performance improvement results for our simulation experiment for the convolution. We compare the execution time of the FIR Filter using a reconfigurable cache(RC) to a conventional general purpose processor(GPP) using a conventional convolution algorithm. Since the reconfigurable cache may have to be flushed, we show the results for the two cases here. 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 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 to be added in the configuration time. The actual parameter values to compute the times are given in Table 5. We choose the values to be as conservative as possible with respect to a SPARC processor cycle time at 269.8 MHz on which the simulation is running. In the table, the computation time of one stage/PE in the RCs is chosen by the following factors. Each stage in the convolution function unit requires 3 LUT reads with additional time for propagation through a number of multiplexers while each PE in DCT/IDCT unit does 2 LUT reads with additional time for multiplexers. We use read time for LUTs of 8ns and cache access time of 12ns. The execution time for one stage and PE also includes the propagation time for multiplexers. The expressions for the times are presented below.

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



(a)



(b)

Figure 8: Ratio of execution time of RC and GPP for Convolution: (a) without memory flush; (b)with memory flush before converting into the function unit

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 Figure 5(a). In addition, we separate the configuration time for adders and multipliers. The reason for this is that only one

Table 5: Parameters for the RCs

Parameter	Description	Actual value
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 rows (cache blocks) in the cache	512
L_{LUT}	Number of rows (cache blocks) in a LUT	16
W_n	Number of words per cache block	8
a	Number of bits required to configure a content of LUT for a 2-bit adder and 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
S	Number of taps/PEs implemented in the RC	8
Parameters for Convolution		
TAP	Number of taps	8 - 256
X	Number of data elements	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	8×8 - 1920×1152

set of data for a LUT is necessary when reconfiguring the LUTs for adders because the contents of all the LUTs are the same while a different configuration data is necessary for multipliers. This can be done by a main processor as a simple instruction. 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 comparison of execution times between RC and GPP are shown in Figure 8. We assume that all the input data fit into a data cache for the computation in both RC and GPP. We traced the number of cache misses in GPP for all the cases in Figure 8. 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 with floating point variables instead of integers in the simulation code of the convolution for faster processing in GPP.

Our results show that the reconfigurable cache for computing has a higher performance improvement over the execution time of the GPP as the number of data elements increases. The performance improvement is gained almost independent of the number of taps without memory flush in Figure 8(a), but the ratio of the computation time with less number of taps decreases with memory flush in Figure 8(b) because the flush time affects the ratio of the total execution time more with the decrease in the number of taps.

5.3.2 DCT/IDCT

As described in Section 5.1.2, the 2-D DCT/IDCT can be completed by two 1-D transforms. This procedure 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 to GPP executing the fast DCT algorithm described in Section 5.1.2. As considered in the previous example, the two cases of cache '*flush time*', no cache flush and cache flush, are shown 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 an adder. 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 5. The expressions for the execution times are presented below.

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

The cache '*flush time*' is the same as earlier. Configuration data needs 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 8×8 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 2 sets of configuration data. Since only one of the pre/post-adders (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 result of the execution time comparison of GPP and RC is shown in Figure 9. The assumption regarding the cache misses of data mentioned in Section *5.3.1* 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, 8×8, 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 necessary to be 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.

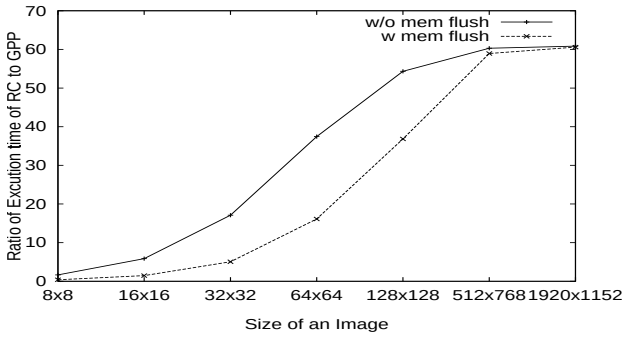


Figure 9: Ratio of execution time of RC and GPP for DCT/IDCT with and without '*flush time*'

According to the result, the reconfigurable cache for DCT/IDCT has a higher performance improvement over the execution time of the GPP as the size of an image processed increases. The performance improvement is roughly independent of the 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, 768×512 (TV-image) and 1920×1152 (HDTV), do not rely on the flush time. For MP@HL (Main Profile at High Level) decoding, the maximum time to process a macroblock is 4.08μs [16]. The result shows that it is possible to process a block in 2.30μs.

5.3.3 Multi-context 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.

6. CONCLUSION

We have presented a reconfigurable 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 overhead for this reconfiguration is about 10-20% of the base cache memory array area with 1-2% increase in the cache access time. Since applications which have a regular structure may be implemented in a reconfigurable module, we are currently developing similar mappings for other functions. Pseudo-programmable interconnection with limited programmability, but with less area overhead, to support more general functions 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-memory" type of capability.

7. REFERENCES

[1] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor",in Proc. of the IEEE Symp. on FCCM, Apr. 1997.

[2] André DeHon, "DPGA-coupled microprocessor: Commodity ICs for the early 21st century", In D. a. Buell and K. L. Pocek, editors, Proc. of IEEE workshop on FPGAs for Custom Computing Machines, pp. 31-39, Apr. 1994.

[3] S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, "The Chimaera Reconfigurable Functional Unit", IEEE Symposium on FPGAs for Custom Computing Machines, pp. 87-96, 1997.

[4] S. Cadambi et al., "Managing Pipeline-Reconfigurable FPGAs", in Proc. ACM/SIGDA 6th Intl. Symp. on FPGA, February 1998.

[5] Xilinx Inc., "Introducing the XC6200 FPGA Architecture", available on www.xilinx.com/apps/6200.htm.

[6] Abhishek Singhal, "Reconfigurable Cache Module Architecture", Master Thesis in Dept. of Computer Science at Iowa State University, in preparation.

[7] H. ElGindy et al., "RMB – A Reconfigurable Multiple Bus Network," in Proc. of Second HPCA Symposium, February 1996, pp. 108-117.

[8] Wafer Electrical Test Data and SPICE Model Parameters, available on www.mosis.org/Technical/Testdata

[9] Deepali Deshpande, Arun K. Somani, and Akhilesh Tyagi,"Configuration Scheduling Schemes for Striped FPGAs", in Proc. of FPGA99, Feb. 1999 pp. 206-214.

[10] M. Wojko and H. ElGindy, "Self Configuring Binary multiplier for LUT addressable FPGAs", in the 5th Australasian Conference on Parallel and Real-Time Systems 1998.

[11] T. Yoshino et al., "A 100-MHz 64-tap FIR Digital Filter in 0.8-μm BiCMOS Gate Array", IEEE Journal of Solid-State Circuits, Vol. 25, No. 6, Dec. 1990, pp. 1494-1501.

[12] M. Hatamian and S. Rao, "A 100 MHz 40-tap programmable FIR filter chip" in Proc. IEEE Intl. Symp. Circuits Syst.,, 1990, pp. 3053-3056.

[13] Ishikawa et al., "Automatic layout synthesis for FIR filters using a silicon compiler", in Proc. IEEE Intl. Symp. Circuits Syst., 1990, pp. 2588-2591

[14] M. Izumikawai et al., "A 0.25-μm CMOS 0.9-V 100-MHz DSP Core", IEEE Journal of Solid-State Circuits, Vol. 32, No. 1, Jan. 1997, pp. 52-61.

[15] Fang Lu, and Henry Samueli, "A 200-MHz CMOS Pipelined Multiplier-Accumulator Using a Quasi-Domino Dynamic Full-Adder Cell Design", IEEE Journal of Solid-State Circuits, Vol. 28, No. 2, pp. 123-132, Deb. 1993.

[16] T. Masaki et al., "VLSI Implementation of Inverse Discrete Transformer and Motion Compensator for MPEG2 HDTV Video Decoding", IEEE Transaction on Circuits and Systems for Video Technology, Vol. 5, No. 5, Oct. 1995.

[17] A. Madisetti, and A. N. Willson, "A 100 MHz 2-D 8×8 DCT/IDCT Processor for HDTV Applications", IEEE Transaction on Circuits and Systems for Video Technology, Vol. 5, No. 2, Apr. 1995.

[18] S. Uramoto et al., "A 100 MHz 2-D discrete consine transform core processor", IEEE Journal of Solid-State Circuits, Vol. 27, pp. 492-499, Apr. 1992.