On Reconfiguring Cache for Computing*

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

Akhilesh Tyagi
Department of Computer Science
Iowa State University, Ames 50011
E-Mail: tyagi@iastate.edu

Abstract

The number of transistors on chip has dramatically increased within the last decade. A considerable portion of a chip is dedicated to a cache memory in a modern microprocessor chip. However, some applications may not need all the caches for storages. In addition, some applications have embedded computations with a regular structure. The behavior of the applications is static, which implies that a specialized function unit could be beneficial for the application. This presents an opportunity to explore the use of a part of a cache for performing these regular computations. In this paper, we show one such design to convert a cache into a function unit to improve the performance of an application. A reconfigurable cache takes less area than the area of a cache and a function unit and imposes no time overhead. In order to convert a cache memory to a function unit, we mapped 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 an application with a large number of data elements by a large factor (as high as 63). The cache, of course, also works as a normal cache with the access time similar to a normal cache when the function unit is not required.

1 Introduction

The number of transistors on a chip has increased dramatically in the last decade [1]. Within the next five years, we will have billions of transistors on a chip. In modern microprocessor, more than half of the transistors are used for cache memories. This trend is likely to continue. However, not all applications will use all the capacities of a cache at a time. This results in low utilization of the cache memory when running the applications which do not require the entire cache capacities.

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 [2, 3, 4, 5, 6, 7, 8, 9]. Such logic can accelerate the execution of applications and then provide results to the host processor or store 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 or a coprocessor to accelerate applications, the data transfer between the processor and those coprocessors increases the requirement for communication bandwidth and eventually results in a bottleneck.

In Garp architecture [2], programmable logic resides on processor-chip to accelerate computations in a conventional processor. The computations are expected to be used frequently in the architecture. If an application does not need the logic, these functions remain idle. PipeRench [10] tries to reconfigure the hardware every cycle to overcome the limitation of hardware resources. XC6200 [11] from Xilinx has a different architecture from the previous FPGA series to allow an on-line reconfiguration. An advantage of this architecture is that a number of smaller configuration memory blocks can be combined to obtain a larger memory. However, the fine-grained memory cannot be synthesized efficiently in terms of area and time. In particular, providing a decoder for small chunk of memory requires a large number of decoders, which is expensive.

This observation motivates us to design a reconfigurable module which works as a function unit as well as a cache memory that reduces the extra logic. A general purpose processor may not perform well for compute-intensive applications such as image processing, encryption, and signal processing. We, therefore, propose to create special-purpose logic for these applications on the processor chip using a reconfigurable memory/function unit module. For this purpose, we partition the cache into several smaller caches. Each cache then can be converted to carry out a specialized dedicated compute-intensive function unit.

Our goal is to develop a mechanism to convert a cache into a function unit to improve the performance of an application by implementing a reconfigurable module with low area and time overhead. We first describe the concept of a multi-bit output LUT used in the proposed architecture in Section 2. Section 3 describes the architecture of a reconfigurable module, which talks about the function and cache operations. The configuration and schedule of the module are mentioned in Section 4. Section 5 shows 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 [12]. However, a SRAM-based single output LUT does not fit well to a cache memory architecture because of a large amount of overhead for the decoders in the cache with a large

*This work was funded by Carver Trust Grants, Iowa State University.

block size. Instead of 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 LUT. Since a multi-bit output LUT has the same inputs for all output bits, it is less flexible in implementing functions. A 2-bit adder and a 2-bit multiplier or a 4x2 constant coefficient multiplier (all need the same size of LUT) are depicted in Figure 1 (a) and (b), respectively.

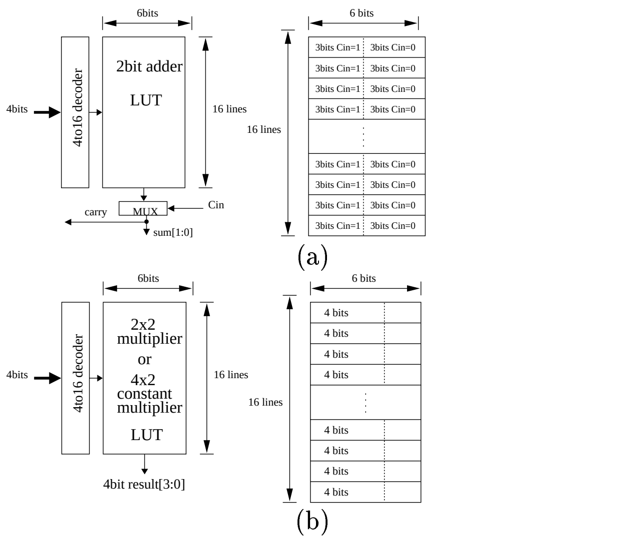

Figure 1: Multi-output LUTs : (a) A 2-bit adder ; (b) A 2-bit or a 4x2 constant coefficient multiplier

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 large LUT requires only one big decoder or a multiplexer with multiple inputs. Thus, the area for decoders reduces. However, the overall memory requirement increases. The required memory size of one large LUT increases exponentially when the number of inputs increases. 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 depicted 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 5-bit outputs 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 only after the previous 4-bit addition in one LUT is completed (i.e. a ripple carry). Since each LUT should 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), this results in a faster adder with 8-LUTs because 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 reading time for one 8-LUT and the propagation time for two multiplexers. If we have more input bits, the modified adder is much faster. However, this adder is not area efficient because it still requires a large amount of memory (i.e. $4 \times 2^8 \times 5 \;=\; 5120\ bits$).

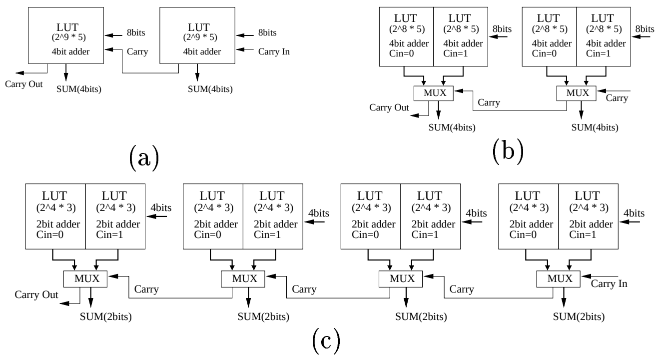

Figure 2: 8-bit adder using (a) two 9-LUTs ; (b) two 8-LUTs; (c) four 4-LUTs

To make an area efficient adder, a 4-LUT with 6-bit outputs can be exploited (Figure 2(c)). The same carry propagation scheme as in Figure 2(b) can be applied to the 4-LUTs to implement an 8-bit adder, but four 4-LUTs are used. The total time of the adder with the 4-LUTs might be larger than that with the 8-LUTs because it has twice the number of multiplexers to be propagated. However, the read time for a 4-LUT is much faster than for an 8-LUT since it has a smaller decoder and shorter lines for reading memory. We, therefore, propose to employ the 4-LUT with multiple outputs to implement such a function.

3 Reconfigurable Cache Module

In design of reconfigurable cache modules, we assume that multiple cache modules are attached to a processor on a chip. Each of these cache modules can be programmed to cache a particular part of the address space [13]. They can also be programmed to act as function units. We also assume that the number of cache units is sufficiently large so that use of a few cache units as function units does not adversely affect the cache operation. Alternately, if the cache modules are organized with k-way set associativity, removing one cache module will convert the cache into a (k-1)-way set associative cache. In either case, it is possible to pull a cache module out of operation to convert it into a function unit.

3.1 Processor with reconfigurable caches

Figure 3 shows the overview of the processor with reconfigurable caches (RCs). RC1, RC2, RC3,…, RCn in Figure 3 can be converted to function units, for example, to carry out functions such as FIR filter, IDCT(mpeg decoding function), Encryption, and General computation unit like a floating point multiplier,

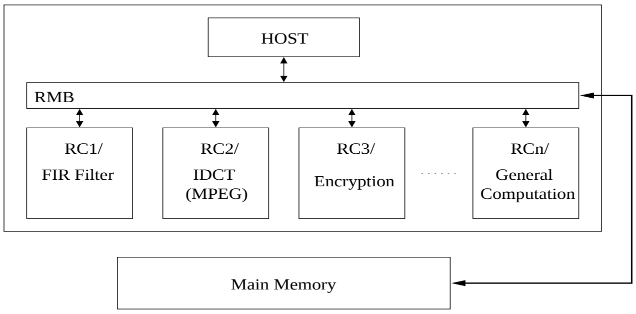

Figure 3: Overview of a processor with multiple reconfigurable cache modules

respectively. All the applications to be embedded into caches are compute-intensive. We assume that each cache module is designed to carry out one specific dedicated function. Other caches not shown in the figure may or may not be reconfigurable. When one cache is used as a function unit (one of the above functions), the other caches continue to operate as storage units as usual. It is also possible to reconfigure some cache modules to become data input and output units for a function unit to provide input and receive output at the speed of function unit. For example, if cache 1 performs FIR filtering, some other cache (cache 2) may initially contain input data and cache 3 may be empty to receive output data. To adapt to the various sizes of data for different applications, we may also use reconfigurable sizes of cache memory modules. Each cache can have a different depth and width and appropriate decoding according to the demand of applications [14, 15, 16]. The configuration of memory is completed by the host processor because the processor has the information about cache mappings to take care of dynamic nature of cache functionality.

When a processor employs several such modules than 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) [17]. RMB makes the communications among cache modules efficient by providing all the required paths within the available resources of RMB. RMB may be well suited to the communication needs between the reconfigurable cache and data caches or the host processor. We will not discuss the actual operation of RMB here and the interested reader is referred to [17].

3.2 Single/Multiple functions in a module

Although a reconfigurable cache module can be reconfigured to realize more than one type of function unit, we propose that each cache module be designed to reconfigure to one specific function unit. Since each reconfigurable module can be converted into only one specific function unit, interconnections for each function unit in the module are fixed. By making the interconnections and the function fixed in a module, we have several advantages and some disadvantages as described below.

Advantages:

1. The fixed interconnection is less complex, takes less area, and allows faster communications than a reconfigurable interconnection. Unlike in FPGAs, it also reduces the routing overhead and communication time between LUTs.

2. With a fixed function in each cache module, we know what the configuration data is and how they must be programmed when the module is converted. Therefore, it is possible to make the reconfiguration faster by implementing a special logic which allows multiple writes to multiple reconfigurable LUTs for identical functionalities or direct connections to LUTs.

Disadvantages:

1. Only a few specific applications can be implemented during the fabrication stage. We can, therefore, choose only the most common functions to be implemented. Since the functions might be application-specific and thus such a set cannot be determined for general purpose computing.

2. The proposed reconfigurable module would take less area than a general purpose reconfigurable module. However, if we implement a number of functions into several reconfigurable modules, the total area of all the modules may be higher than a single general purpose reconfigurable module.

In this paper, each reconfigurable module has a dedicated and fixed interconnection for its own dedicated function implemented using the LUTs described in Section 2. The interconnections between LUT rows do not affect the operations when the module is used as a cache memory since data is read/written through global bit lines described in the next section.

3.3 Organization and operation of module

Since we target the applications that are compute-intensive and have a regular structure, we first partition them at coarse-level. A function in each level can be implemented using the multi-bit output LUTs as described in Section 2. 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 computations can be performed in a pipelined fashion. Each basic computation in one pipeline stage, such as an addition or a multiplication, can also be pipelined. This pipeline increases the number of pipeline stages, but improves the throughput of the pipeline by shortening the cycle time of one pipeline stage. However, it requires more pipeline registers, which increases the area overhead. In this paper, to reduce the area overhead, we add registers to each stage, but not to each LUT.

Figure 4 depicts a possible cache 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 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 becomes input 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 from the big decoder. 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.

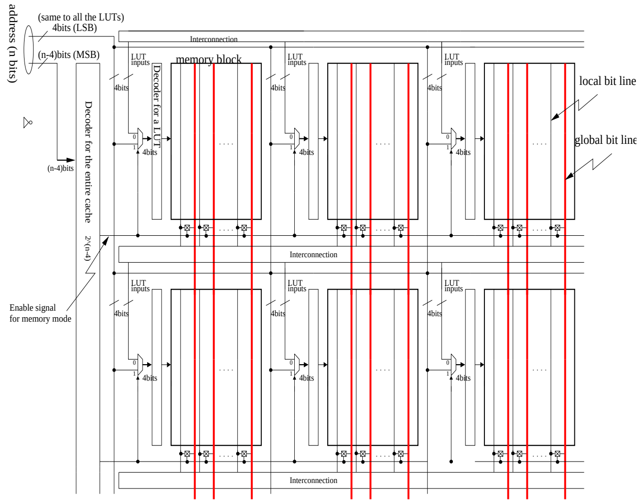

Figure 4: Cache operation in the reconfigurable Module

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 rest of the $(n-4)$ address bits decoded by the big 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 like 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.

In summary, cache operations are possibly slowed by the additional logics for a function unit. We estimate this time penalty in the next section.

3.4 Access time for cache operations

Each bit line in a normal cache is replaced by the global line in the proposed architecture. Since there are no obstructive switches or logic on the global line path except the connections (contacts) with the switches to the LUTs, the reconfigurable module does not have higher delay due to the global line. However, the signal may be degraded by the increased line capacitance due to the contacts for switches and longer global lines than a normal cache. It, therefore, may require additional amplification.

On the other hand, the cache with the reconfigurable structure may have a faster address decoder than a normal cache. Since each LUT, with its own row decoder for addressing in the reconfigurable module, is much smaller than a large synthesized memory block in a conventional cache, the decoding time of a LUT is much 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 are 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 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 which 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 special driver for the least significant 4 bits to reduce the delay. However, the drivers will not affect the size of reconfigurable module much as we can put a driver into the space saved due to the reduced size decoder for higher order bits. A conventional cache may also employ such a hierarchical decoding mechanism.

The lengths of word and bit lines in the reconfigurable cache are more than those in a normal cache due to the dedicated row decoders and the interconnections between LUTs. Therefore, the delays of propagating a signal from the big decoder through the word lines and a data signal from memory cell through the bit lines in the module are higher than those in a normal cache. Other delays are similar in both cases.

To estimate the access time, we used a cache simulator, CACTI with $0.8\mu m$ technology, which was developed by Steven J.E Wilton and Norman P. Jouppi [18]. The access time consists of five parts, Decoder, Word line, Bit line, Sense Amplifier, and Data out driver for tag and data parts. We modified the simulator slightly to fit into our structure for a different decoding scheme (without the driver) and longer lines only for the data part. We computed the access time of data part for 8KB cache with 128 bits block size, which is used to implement our function example in Section 5. Table 1 shows the access time of a normal cache and the reconfigurable module. As we described above, it turns out that the overall decoding time is decreased and the delays of word and bit lines are increased. Since we reduce the decoding time significantly, the total cache access time, sum of those five factors, in the reconfigurable module is less than in a normal cache.

Table 1: Comparison of access time for an 8KB cache

	Normal cache (ns)	Reconfigurable Module (ns)	Comments
Decoder	2.87779	Max(2.3828, 2.66958) = 2.66958	decreased
line	1.14371	1.24335	increased
Bit line	0.461652	0.547231	increased
Sense Amplifier	0.58	0.58	same
Data out driver	0.598526	0.598526	same
Total time	5.66168	5.57030	decreased

3.5 Area overhead

In the new architecture, additional areas are required for the dedicated interconnections, row decoders for LUTs, additional control units, and pipeline registers. Other resources such as the row decoder for upper bits and memory cells are required for both function and cache operations. In this section, we compare the areas of a normal cache module and a reconfigurable cache module.

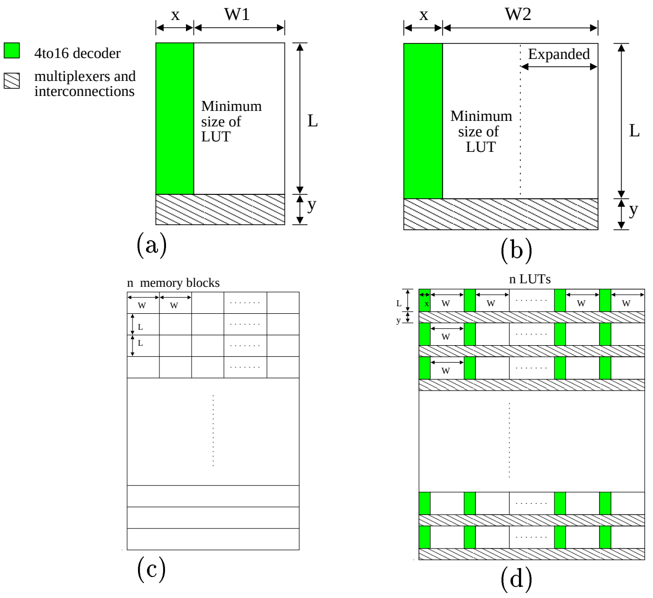

Figure 5: (a) a multi output LUT with decoder and interconnection overhead; (b) an expanded width LUT; (c) Structure of a normal cache ; (d) Structure of a reconfigurable cache

Figure 5 shows the areas for various components of a reconfigurable cache compared to a normal cache blocks. In the figure, the size of a LUT in the reconfigurable cache is $(W_1+x)(L+y) = W_1L + W_1y + Lx + xy$ while the size of the original cache block is W_1L. W and L are the width and the depth of a memory block to be used for a LUT, respectively. x is the additional width due to the dedicated decoder while y is the additional depth due to multiplexers and interconnections for the reconfigurable cache. The area overhead is as much as $\frac{W_1y+Lx+xy}{W_1L}$. If $x = aW_1$ and $y = bL$, the overhead factor is $a + b + ab$. The area overhead depends on the fraction of the additional lengths vertically and horizontally due to the 4-to-16 decoder and the fixed interconnection, respectively.

If we expand the size of LUTs with respect to the width of memory cells, $W_2 = cW_1$ in Figure 5(b), the fraction of the additional width will decrease by $x^{'} = \frac{a}{c}W_2$ in the expanded LUT. The fraction of total area overhead in the new size of LUT is $\frac{a}{c} + b + \frac{ab}{c}$. By increasing the width of the LUT, the fraction of area overhead is reduced by $(a + b + ab) - (\frac{a}{c} + b + \frac{ab}{c}) = a(1 - \frac{1}{c}) + ab(1 - \frac{1}{c})$, where $a, b < 1$,and $c > 1$. For example, if $a = \frac{1}{2}, b = \frac{1}{4}$ and $c = 2$, the overhead factor for a normal size LUT is $a + b + ab = \frac{7}{8}$ while for an expanded LUT is $\frac{a}{c} + b + \frac{ab}{c} = \frac{9}{16}$. In addition, using the expanded LUT reduces the number of row decoders for LUTs in a given row. Section 4.2 describes how we determine the number and the size of LUTs efficiently in terms of area and configuration time. In a complete reconfigurable cache, we have to add the area for the pipeline registers to the area overhead. The area overhead of an entire reconfigurable cache without the registers is $\frac{n \times area\ of\ one\ LUT}{the\ area\ of\ the\ original\ cache} = \frac{n \times WL \times (overhead\ for\ a\ LUT)}{n \times WL} = a + b + ab$, where n is the number of LUTs implemented in the cache. Figure 5(c) and (d) show structures of a normal and a reconfigurable cache, respectively. The total area overhead is the sum of the above results and the area of pipeline registers, which is $a + b + ab + (fraction\ of\ the\ area\ overhead\ due\ to\ pipeline\ registers)$.

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 consecutive 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. 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 blocks in a cache.

4.2 Number and size of LUTs

The following conditions determine the number and size of LUTs with the parameters described in Table 2. The size and number of LUTs must be a power of 2. After determining the most efficient number of LUTs in a row, we can reduce the area overhead by using dedicated decoders in the LUTs.

Condition1: *total # of bits/block* $\geq$ (*# of LUTs*) $\times$ (*# of bits required per decoded entry in a LUT*)

$\Longrightarrow N_{b/w} \times N_{w/B} \geq x \times a$

Condition2:

 # LUTs required for a function in a row $\leq \frac{\text{total \# bits/block}}{\text{the width of LUT}}$

$\Longrightarrow x \leq \frac{N_{b/w} \times N_{w/B}}{m \times N_{w/B}}$ *or* $m \leq \frac{N_{b/w}}{N_{w/B}}$

These conditions imply that we can write $m \leq \frac{N_{b/w}}{x}$ configuration bits into each LUT in a row with writing of one word in the cache. Also, the actual number of LUTs implemented in a row is equal to $\frac{N_{b/w}}{m}$.

Table 2: Parameters for the number and size of LUTs

a	Number of bits required in a row of LUT
x	Number of LUTs required in a LUT row
m	Number of bits to be written into a LUT by one word (power of 2)
$N_{b/w}$	Number of bits per word
$N_{w/B}$	Number of words per block

4.2.1 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 either available in an on-chip cache or an off-chip memory. Loading time for the configuration data in the latter case will be much 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.

Since we have a dedicated function in each memory and each row of LUTs has its own local decoder, a reconfigurable decoder can also be constructed to assist the reconfiguration for the specific function to allow multiple writes to the LUTs, one in each row with the same contents. The implementation of the reconfigurable decoder, which allows multiple writes into memory bits in a column, can be realized by adding multiplexers. Choosing all the parameters as a power of 2 also results in an easier implementation of the reconfigurable decoder.

Parts of the cache unit as a function unit can be 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 next set of 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 reconfiguring at the same time. Since the number of computation data is usually much larger than the number of reconfiguration data, the reconfiguration may not occur frequently. Therefore, a partial reconfiguration usually does not take much time. For example, in a convolution application with 16 taps, we need to reconfigure part of the module 16 times. It is possible to perform both computations and reconfigurations simultaneously if we have separate data lines for computation data and configuration data. This is expensive, and therefore, we do not recommend it.

4.2.2 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 describe in [19] 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 following step. We have to store the intermediate results from the sets 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 addressed-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 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 includes 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

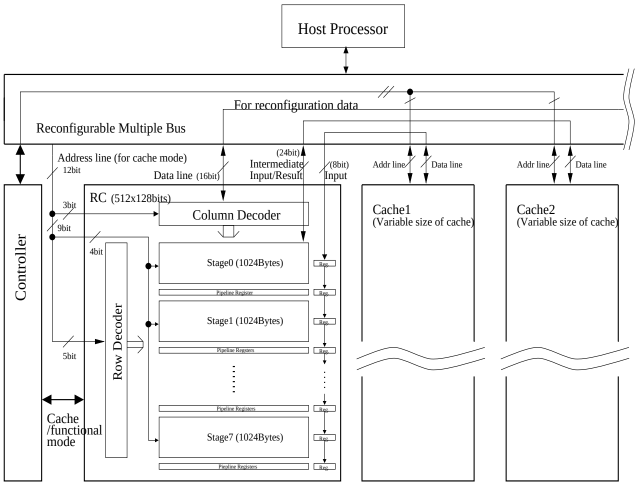

Figure 6: On-chip reconfigurable cache architecture

of inputs, 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 a 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. The controller also monitors the computation and initiates the next step when the current step is completed. Figure 6 shows some of these details.

5 Experimental Result

5.1 Experimental setup

We experimented with a design of the reconfigurable cache converted to perform a convolution function. 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. A conventional convolution algorithm is shown in Figure 7.

```
for i=1 to NumberOfIN {
    Y[i] = 0;
    for j=1 to NumberOfTAPs {   Y[i] += X[i+j-1]*W[j];   }   }
```

Figure 7: Convolution algorithm

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 8(a). The input data is double pipelined in one stage for the appropriate computation [10]. An 8×8 constant coefficient multiplier can be implemented using two 4×8 constant coefficient multipliers and a 12-bit adder with appropriate connections [20]. A 4×8 constant coefficient multiplier is implemented using 12 4-LUTs with single output from each LUT on FPGAs. In our simulation, we split the 4×8 LUT of the conventional constant coefficient multiplier into two 2×4 LUTs with multiple outputs as shown in Figure 8 (b).

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. The structure of LUT for the 2-bit addition is the same as that of the 2×4 constant coefficient multiplier since that also produces a 6-bit output. An n-bit adder can be implemented using $\lceil \frac{n}{2} \rceil$ such LUTs and a carry propagation scheme. The output is selected based on the input carry.

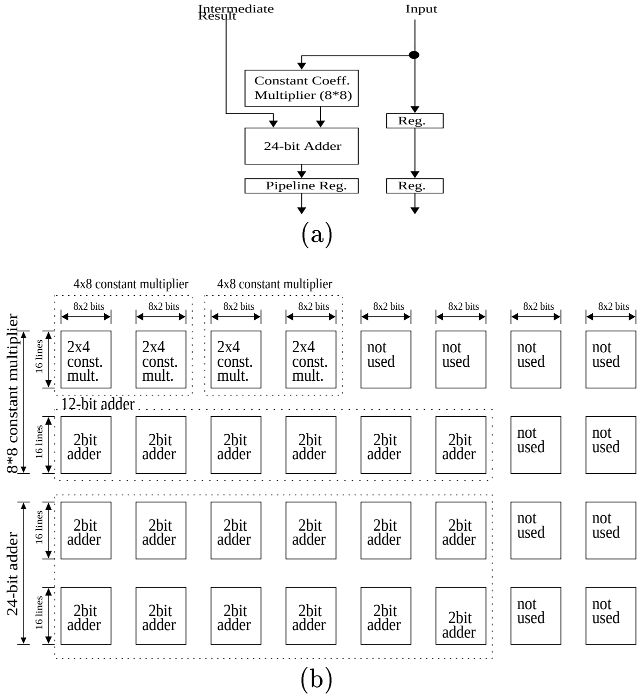

Figure 8: Multi-output LUTs : (a) A 2-bit adder ; (b) A 2-bit or a 4x2 constant coefficient multiplier

One stage of convolution can be implemented with 22 LUTs. To keep the number of LUT rows a power of two, we put 6 LUTs in each LUT row and have 4 LUT rows to use 22 (out of 24) required LUTs. According to the conditions in Section 4.2, we can have $m = 2$ ($m \leq \frac{N_{b/w}}{x}$) and the actual number of LUTs per row turns out to be 8 ($\frac{N_{w/B}}{m}$). The final placement of LUTs is shown in Figure 8(b). A few LUTs in the figure are not used for the computation. In Fig-

Table 3: Parameters for the RC

Parameter	Description	Actual value
T_{cpu}	1 cpu cycle time	4ns
TAP	Number of taps	8 - 256
X	Number of data	64 - 8192
S	Number of taps implemented in the RC	8
T_{1_stage}	The time to complete the computation in one stage	24ns
$R_{mem/cpu}$	Ratio of no. of cycles of 1 main memory access and 1 cpu cycle	20 (80n)
$R_{cache/cpu}$	Ratio of no. of cycles of 1 cache memory access and 1 cpu cycle	3 (12ns)
a	Number of bits required to configure a content of LUT	6
m	Number of bits to be written by one word when configuring	2
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

ure 8(b), pipeline registers and decoders for LUTs 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.

5.2 Area

To measure the actual area overhead, we experimented with a possible layout of a reconfigurable cache. Figure 9 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. we obtained $a = 0.281$ and $b = 0.179$, the parameters for area overhead in Section 3.5. The value for a takes into account the parameter c. In addition, the area overhead of the pipeline registers is roughly 5% including the interconnections. Therefore, the total area overhead is $a + b + ab + 0.05 = 0.560$. Therefore, the total area of the reconfigurable module is 1.56 times of a normal cache. Since the interconnections are fixed for the convolution, they can be efficiently routed and do not take much area. Since the placement and routing of the reconfigurable cache have 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 and routing is used.

5.3 Execution time

We achieved the following performance improvement results for our simulation experiment for the convolution. We compare the execution time of the function using a reconfigurable cache(RC) to a conventional general purpose processor(GPP) using the algorithm in

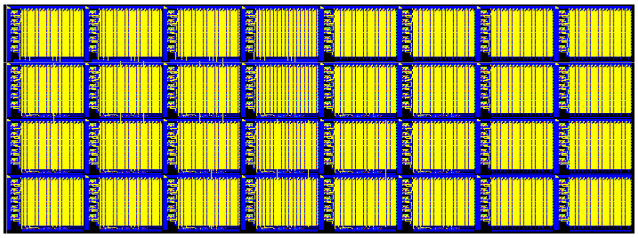

Figure 9: A layout for one stage of the convolution

Figure 7. 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. In the second case, the processor has to flush all the data in the cache before configuring it. The extra time is denoted by the '*flush time*'.

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 3. The expressions for the times are presented below.

- Config. Time for adder = $[\ (R_{mem/cpu})(\frac{a}{m})(L_{LUT}) + (R_{cache/cpu})(\frac{a}{m})(L_{cache} - 16 \times S)\ ] \times T_{cpu}$
- 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 = $[\ (\frac{TAP}{S}) \times (X + 2S - 1)\ ] \times T_{1\_stage}$

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 8(a). In addition, we separate the configuration time for adders and multipliers. The reason for this is that only one 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. Therefore, we can store each configuration data of a LUT for an adder into an on-chip register and write the data to all the LUTs for adders with cache access time. The time for storing and loading input and intermediate data can be overlapped with the computation time. Therefore, data accessing time for the computation is not added.

The ratios of execution times between RC and GPP are shown in Table 4. We assume that all the input data fit into a data cache for the computation in both

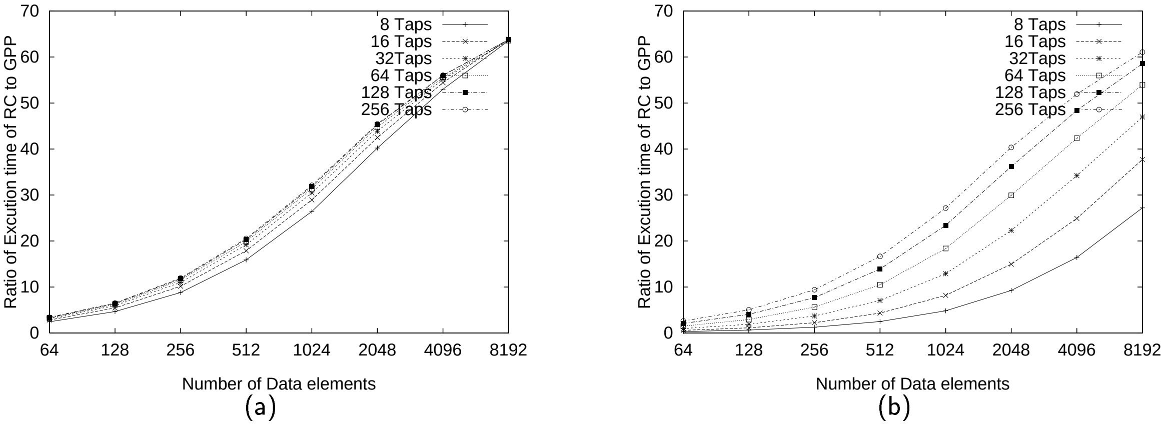

Figure 10: **Ratio of execution time of RM and GPP : (a) without memory flush; (b)with memory flush before converting into the function unit**

Table 4: Comparison of execution time between SPARC and RM (usec)

No. of Taps	No. of Data	SPARC (269.8 MHZ)	RM w/o memory flush config	RM w/o memory flush compute	Ratio	RM w/ memory flush config	RM w/ memory flush compute	Ratio
8	64	122.30	48.38	1.90	2.43	376.06	1.90	0.32
	128	241.64		3.43	4.66		3.43	0.64
	256	483.27		6.50	8.81		6.50	1.26
	512	970.48		12.65	15.90		12.65	2.50
	1024	1937.50		24.94	26.43		24.94	4.83
	2048	3937.51		49.51	40.22		49.51	9.25
	4096	7793.35		98.66	53.00		98.66	16.42
	8192	15589.69		196.97	63.54		196.97	27.21
16	64	234.11	79.10	3.79	2.82	406.78	3.79	0.57
	128	469.07		6.86	5.46		6.86	1.13
	256	936.01		13.01	10.16		13.01	2.23
	512	1866.74		25.30	17.88		25.30	4.32
	1024	3733.65		49.87	28.95		49.87	8.18
	2048	7567.92		99.02	42.49		99.02	14.96
	4096	15049.62		197.33	54.44		197.33	24.91
	8192	30210.64		393.94	63.87		393.94	37.73
32	64	461.93	140.54	7.58	3.12	468.22	7.58	0.97
	128	918.38		13.73	5.95		13.73	1.91
	256	1829.59		26.02	10.99		26.02	3.70
	512	3665.05		50.59	19.18		50.59	7.06
	1024	7318.92		99.74	30.46		99.74	12.89
	2048	14857.10		198.05	43.88		198.05	22.30
	4096	29515.00		394.66	55.15		394.66	34.21
	8192	59001.65		787.87	63.55		787.87	46.97
64	64	909.15	263.42	15.17	3.26	591.10	15.17	1.50
	128	1812.71		27.46	6.23		27.46	2.93
	256	3624.94		52.03	11.49		52.03	5.64
	512	7252.87		101.18	19.89		101.18	10.48
	1024	14521.47		199.49	31.37		199.49	18.37
	2048	29548.45		396.10	44.80		396.10	29.93
	4096	58472.42		789.31	55.54		789.31	42.36
	8192	116919.44		1575.74	63.57		1575.74	53.96
128	64	1809.67	509.18	30.34	3.35	836.86	30.34	2.09
	128	3613.38		54.91	6.41		54.91	4.05
	256	7237.46		104.06	11.80		104.06	7.69
	512	14460.41		202.37	20.32		202.37	13.91
	1024	28935.84		398.98	31.86		398.98	23.41
	2048	58954.22		792.19	45.30		792.19	36.19
	4096	116893.10		1578.62	55.99		1578.62	48.39
	8192	233513.89		3151.49	63.79		3151.49	58.55
256	64	3620.02	1000.70	60.67	3.41	1328.38	60.67	2.61
	128	7239.17		109.82	6.52		109.82	5.03
	256	14440.05		208.13	11.95		208.13	9.40
	512	28875.14		404.74	20.55		404.74	16.66
	1024	57732.70		797.95	32.10		797.95	27.15
	2048	117543.62		1584.38	45.47		1584.38	40.35
	4096	233012.31		3157.25	56.04		3157.25	51.95
	8192	465962.63		6302.98	63.80		6302.98	61.06

RC and GPP. We traced the number of cache misses in GPP for all the cases in Table 4. 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.

Our result shows 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. Figure 10 shows that the performance improvement is gained almost independent of the number of taps without memory flush in (a), but the ratio of the computation time with less number of taps decreases with memory flush in (b) because the flush time affects the ratio of the total execution time more with the decrease in the number of taps.

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 computing bandwidth for I/O bandwidth. We have analyzed it for a convolution. The reconfigurable cache for the computation of convolution improves the performance by a large amount (a factor of up to 63). The area overhead for this reconfiguration is about 60% without any 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. We are also studying if more than one function can be combined into one single reconfigurable module.

References

[1] Y.Patt, S. Patel, M. Evers, D. Friendly, and J. Stark, "One billion Transistors, One Uniprocessor, One Chip", IEEE Computer, pp. 51-57, Sep. 1997.

[2] John R. Hauser and John Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor",in Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, Apr. 1997.

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

[4] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units", in Proc. of the 27th Annual Intl. Symp. on microarchitecture, pp. 172-80. IEEE/ACM, Nov. 1994.

[5] A. Tyagi, " Reconfigurable memory queues as computing units architecture", in Proc. of the Reconfigurable Architecture workshop at 11th International Parallel Processing Symposium, Apr. 1997.

[6] 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.

[7] Hokie RISC processor, www.ee.vt.edu/courses/ee6504/rapid.html.

[8] Ralph D. Wittig and Paul Chow, "OneChip: An FPGA Processor With Reconfigurable Logic," in IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 1996

[9] National Semiconductor's Adaptive Systems on-a-Chip (ASC), www.national.com/appinfo/milaero/napa1000/.

[10] Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, Donald E. Thomas, "Managing Pipeline-Reconfigurable FPGAs", in Proceedings ACM/SIGDA Sixth International Symposium on FPGAs, Feb. 1998.

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

[12] J. Rose, R. Francis, D. Lewis, and P. Chow, "Architecture of programmable gate arrays: The effect of logic block functionality on area efficiency", IEEE Journal of Solid-State Circuits, v. 25, pp. 1217-1225, Oct. 1990.

[13] C. Wittenbrink and A. K. Somani, "Cache Tiling for High-Performance Morphological Image Processing", in the Proc. of Computer architecture for Vision Processing, France in Dec. 1991, pp.427-438.

[14] Steven J.E. Wilton, "Implementing Logic in FPGA Embedded Memory Arrays: Architectural Implications", in the IEEE Custom Integrated Circuits Conference, May 1998.

[15] Tony Ngai, Jonathan Rose, and Steven J.E. Wilton, "An SRAM-Programmable Field-Configurable Memory", IEEE Custom Integrated Circuits Conference , pp. 499-502, 1995.

[16] Steven J.E. Wilton, Jonathan Rose, and Zvonko G. Vranesic, "Architecture of Centralized Field-Configurable Memory", in ACM/SIGDA International Symposium on FPGAs, pp. 97-103, 1995.

[17] H. ElGindy, A. K. Somani, H. Schröder, H. Schmeck, and A. Spray, "RMB – A Reconfigurable Multiple Bus Network," in Proc. of Second HPCAS Feb. 1996, pp.108-117.

[18] Steven J.E Wilton and Norman P. Jouppi, "CACTI: an enhanced cache access and cycle time model", IEEE Journal of Solid State Circuits. v. 31 May '96 pp.677-88.

[19] Deepali Deshpande, "Configuration Scheduling Schemes for Striped FPGAs ", in Proc. of FPGA99, Feb. 1999

[20] Mathew Wojko and Hossam ElGindy, "Self Configuring Binary multiplier for LUT addressable FPGAs", in the 5th Australasian Conference on Parallel and Real-Time, Sep. 1998.