# PARAMETRIC ARCHITECTURE FOR IMPLEMENTING MULTIMEDIA ALGORITHMS

*N. Kavvadias and S. Nikolaidis*
Electronics and Computers Div., Department of Physics
Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece
nkavv@skiathos.physics.auth.gr, snikolaid@physics.auth.gr

**ABSTRACT**

Multimedia applications are characterized by high computational demands related to data transfer and storage operations. Multimedia algorithms in their majority consist of regular repetitive loop constructs. In this paper, a novel control architecture for implementing such loop intensive algorithms is described. The proposed control unit takes advantage of the regularity of computations in order to serve as high performance parametric controller of multimedia datapaths. The control unit cooperates with datapath modules and their corresponding controlling FSMs. Algorithmic flow dependencies which determine the appropriate loop sequencing are mapped on a LUT. For another algorithm to execute, LUT context and FSM configurations only have to be reprogrammed. Thus, partial reconfiguration possibilities for implementing multimedia algorithms on programmable platforms can be exploited. For demonstration purposes, a matrix multiply algorithm implementation case is investigated. Compared to a software realization on ARM7 processor, significant performance improvements are reported.

## 1. INTRODUCTION

The popularity of multimedia systems used for computing and exchanging information is rapidly increasing. With the emergence of portable multimedia applications (mobile phones, laptop computers, video cameras, etc) the power consumption has been promoted to a major design consideration [1]. Consequently, there is great need for power and performance optimized architectures that can introduce large savings compared to conventional approaches.

Multimedia applications require increased performance combined with low power consumption figures, for efficiently manipulating large amounts of data in real time. Two general implementation approaches exist, to meet this demand. The first is to use embedded instruction set processors. This choice offers programmability but requires increased power while achieving relatively poor performance. The second is to involve Application Specific Integrated Circuits (ASICs) (or Application Specific Instruction set Processors - ASIPs). The second solution leads to high performance and reduced power consumption, however lacks flexibility.

Targeting multimedia systems on reconfigurable architectures is a third option that is recently gaining significant design space. Fast reconfiguration and partial reconfiguration possibilities have increased the applicability of FPGAs in environments where various algorithms have to be supported. Configurable logic is also very effective in speeding up regular, repetitive computations [2]. Loop-intensive programs fall within this class of computations. In [2], the issue of efficiently mapping the flow of execution of loop-dominated algorithms on parametric control hardware is discussed.

Previous research efforts take advantage of the regularity of loop intensive algorithms to reduce instruction memory access overhead. For this purpose, a small intermediate caching layer, between the on-chip main cache and the execution core, namely the loop cache, is used in many cases [3][4]. A methodology for introducing a loop table which stores frequently executed loops is presented in [5]. In this approach, application programs are profiled in order to acquire the necessary information to update the loop table. Some DSP processors provide hardware loop mechanisms for zero overhead branching to and between loops. E.g. in the ST120 [6] up to three loops can be stored, which may be fully nested or independent to each other. Finally, in [2] techniques for mapping linear and nested loop constructs on reconfigurable hardware are developed. Linear loop operations are mapped onto a sequence of configurations that are used to execute these tasks in order to minimize execution time. Nested loop optimizations are the result of refinement of the datapath by transformations. However, the concept of mapping the whole algorithm is not considered.

In this work, an architectural solution for generating a control structure that succeeds dramatic performance improvement and eliminates instruction memory load is introduced.

The proposed control unit can be used for the execution of structured algorithms for any combination of loops. For the loop (*for*) statement indexing, initial value, step and final value are given as parameters. As the controller runs, full access to the values of the indexes is provided, at any time, so that any function requirement can be satisfied. Also *for* statements with variable indexes are supported.

## 2. MULTIMEDIA ALGORITHM CHARACTERISTICS

The vast majority of multimedia algorithms can be viewed as small data-processing kernels comprising of loop statements that consist of regular, repetitive

computations. Such constructs are considered as strong candidates for performance enhancement using configurable hardware [2].

More complicated algorithmic structures can be resulted after the application of transformations on the multimedia code. A methodology for the derivation of such transformations has been proposed in [1][7] in order to reduce power consumption of data-dominated applications. These are data-reuse transformations that introduce additional loops in the original algorithm code, which imply the existence of memory hierarchy layers closer to the processor.

A general form of a multimedia algorithm is shown in Fig. 1a. It consists of a loop structure where data processing tasks are positioned. These tasks are placed either in innermost or between adjacent loops. We distinguish two types of data processing tasks. The first type noted as asterisk in Fig. 1 corresponds to tasks which close a loop. These tasks control the indexes of the loops and are designated as backward (bwd) tasks. Bwd tasks are always placed for closing a loop even when no processing task is implied. Tasks of the second type are quoted with a circle and do not participate in the algorithmic flow control decision. Such tasks are termed as forward (fwd) and have no effect on the loop indexes.

In the worst case when all possible tasks exist in an algorithm with $n$ loops, there is a maximum of $2n+1$ total tasks. There may be up to $n$ fwd tasks and $n+1$ bwd tasks.

Although the operation of the proposed controller was verified for complex loop structures, it was also evaluated for a popular multimedia algorithm used in many applications in image/video processing and computer graphics, namely the blocked matrix multiplication algorithm. The algorithm is of a 5-loop fully nested structure. The algorithm topology together with accompanying tasks are indicated in Fig. 1b. Pseudocode for the specific implementation is given in Fig. 7. Fwd and bwd tasks of loop 4 involve memory writes while bwd of inner loop is a multiply-and-accumulate operation.

## 3. CONTROL UNIT ARCHITECTURE

For implementing loop-intensive multimedia algorithms, the control unit architecture of Fig. 2, is proposed. First, the computing necessities for implementing the target algorithm as outlined in Fig. 1, are divided in control and data path requirements. Data path modules have to be included in the system architecture to perform the appropriate operations on data as manifested by the algorithm description. The form of datapath depends on the executed tasks. Datapath may consist of one or more modules. At any time, datapath processes a task of the algorithm and is controlled by a corresponding FSM. The main task of the control unit is to direct the datapath controlling procedure so that the appropriate FSM undertakes control of the datapath.

For a specific datapath unit to execute, its associated FSM has to be activated in order to produce the processing enable signals. Bwd and fwd FSMs are situated in the corresponding *task FSM* modules in the control unit. Every FSM and its related task in the algorithm are assigned with an address. This address is combined with the information on whether the current loop has finished, and *task FSM* module selection (fwd only) to specify the following loop. For selecting the next loop, two cases are always possible, the one is to stay resident on the current loop and the other is to close this loop and enter the following as determined by the loop dependencies. These dependencies which determine loop succession consist of the necessary loop sequencing information and are mapped on a Look-Up Table (LUT).

In Fig. 6, the address and data word formation of the LUT are given. The address word incorporates all necessary information that is to address the current loop (*loop_addr*), identify the task type (*FSMsel*), select the specific fwd FSM of a given loop (*fwdsel*), and determine if the current loop has finished (*loop_end*). The data word consists of the *FSMsel*, *fwdsel* and *loop_addr* fields.

For iterating a loop with our control structure, first it has to be explicitly defined. For this reason, the loop index values are stored prior execution in memory blocks (ROMs or register files). The *loop count unit* which incorporates the LUT, addresses the *for* statement parameter blocks. These parameters are input to the *FOR structure* module as shown by the heavy connection in Fig. 2. The *FOR structure* calculates the index values for the addressed loop. The new index issued becomes valid and is stored in a local register file, on a control signal from the corresponding bwd FSM. This is indicated as the *load* input of the *FOR structure* in Fig. 2. A special control unit, the *index control*, generates a selection output for controlling the multiplexing within the *FOR structure*. Logic implementation details of the *index control* slice are depicted in Fig. 4.

The *FOR structure* is exhibited in Figure 3. An adder is used to calculate the current index by adding the step to the initial value. Multiplexers, controlled by either the *index control* or the comparator outputs, are placed for passing correctly the initial, final or stepped values as needed. An internal register file stores the current index values for all loops and is controlled by a load signal issued from the corresponding bwd FSM. When the bwd FSM ends its cycle, a pulse is provided to the *load* input and the new stepped index value is written. The register file is multi-port in order for all indexes to be available for address or data computations. The comparator unit, outputs a pulse when the final iteration value of an index is issued. This signal anded with the *FSM_end* returned from the bwd FSM module, enables loading the ascending loop address from the *loop count unit*.

As seen in Fig. 4, a toggle flip flop (Tff) is used for storing characteristic information for each loop. For logic zero (‘0’), the output selects the initial index in the FOR structure, whereas for logic one (‘1’) the stepped value is selected. Initially, the Tff output is low. If the given FSM is selected, the *FSMsel* is on. When the

FSM terminates, *FSM_end* is on too, and a high value is passed to the multiplexer input. If the loop has not finished, *sel* input is low and this logic one sets the Tff to a high output. When the loop reaches its final iteration, the *loop_end* is high and *sel* changes to '1'. The current high output passes to the Tff input and toggles its state, driving it back to '0'. In this way when the next iteration of the same loop is started, its initial value will be selected.

The corresponding bit-slice of a *fwd task FSM* module is shown in Figure 5. Selecting the respective FSM within the FSM module is done by demultiplexing the *FSM_start* input by the *loop_addr* and *fwdsel* select inputs. The *loop_addr* selects the current loop and the *fwdsel* the specific fwd in the given loop. The *FSM_end* signals from both types of FSM control modules are inputs to a multiplexer which output enables the LUT register that holds the next loop ROM data word. The end signals from bwd FSM have an additional functionality as multiplexed together form the *load* signal to the *FOR structure*.

By this concept, the algorithm execution flow is encapsulated in the matter of suitably routing FSM execution from the current to the appropriate following task, as determined by the structure of the algorithm. This information is stored within the control unit. An arbitrary number of iterative constructs can be mapped as a set of dependencies and encoded into a LUT.

The control architecture is described in a hardware description language (VHDL) and its functionality is verified via simulations. As a real case application demonstrator, image filtering by the blocked matrix multiplication algorithm has been used. Loop index update and loop termination are performed within a single clock cycle. This is advantageous towards conventional microprocessors that translate a C-level FOR instruction in a number of assembly-level instructions e.g. 5 (ARM7). These instructions then need a minimum of one clock cycle each to execute. The second main source of performance increase is the adaptation of task-specific datapath units. The MAC used for the loop 5 bwd task, requires one cycle for its operation, while on ARM7 due to complex addressing, same functionality is performed in 13 instructions.

For a frame size of 176x144 and square blocks of 8x8, code executing on ARM7 takes 7.5 million cycles while on the proposed architecture only 300,000 cycles. The equivalent performance improvement factor is higher than 20. If no datapath optimizations are made, performance gains restrict to the implementation of the *for* instructions, and the associated factor is close to 5. Higher percentage savings are expected for the power dissipation since the power cost of a clock cycle in a general purpose processor (including memory accesses) is higher than in custom design.

## 4. CONCLUSIONS

In this paper, a parametric control architecture for implementing multimedia algorithms is introduced. This is able to execute structured algorithms for any combination of loops. While it operates, indexes of all loops are accessible so that data or address requirements can be satisfied. The proposed architecture is documented in an HDL and its correct functionality is verified by simulation. Drastic performance improvement is imposed by our approach over conventional implementation of the examined case on ARM7 processor. Every loop structured algorithm is supported and short-term research is focused on executing texture analysis and motion estimation algorithms on the proposed architecture.

## REFERENCES


[1] F. Catthoor, S. Wuytack et al., *Custom Memory Management Methodology*, Kluwer Academic Publishers, Boston, 1998.

[2] K. Bondalapati, "Modeling and Mapping Dynamically Reconfigurable Hybrid Architectures", Ph.D. Thesis on Computer Engineering, University of Southern California, August 2001.

[3] L. H. Lee, B. Moyer, J. Arends, "Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops", International Symposium On Low Power Electronics and Design, San Diega, CA, August 1999.

[4] A. Gordon-Ross, S. Cotterell, F. Vahid, "Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example", to appear in IEEE Computer Architecture Letters, January 2002.

[5] F. Vahid, A. Gordon-Ross, "A Self-Optimizing Embedded Microprocessor using a Loop Table for Low Power", International Symposium on Low Power Electronics and Design, Huntington Beach, CA, August 2001.

[6] STMicroelectronics, *ST120 DSP-MCU Core Reference Guide*, Release 1.2.

[7] S. Kougia, A. Chatzigeorgiou, N. Zervas and S. Nikolaidis, "Analytical Exploration of Power Efficient Data-reuse Transformations on Multimedia", International Conference on Acoustics, Speech and Signal Processing, Utah, May 2001.


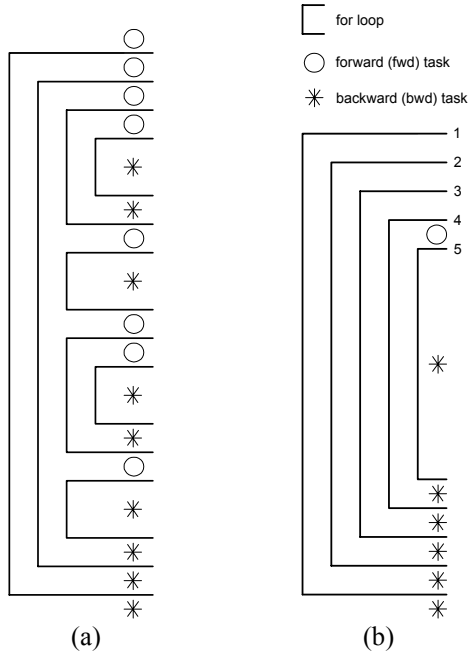


**Figure 1.** (a) Data Processing task placing in a generic loop-dominated algorithm, (b) The same applied for the blocked matrix multiplication algorithm

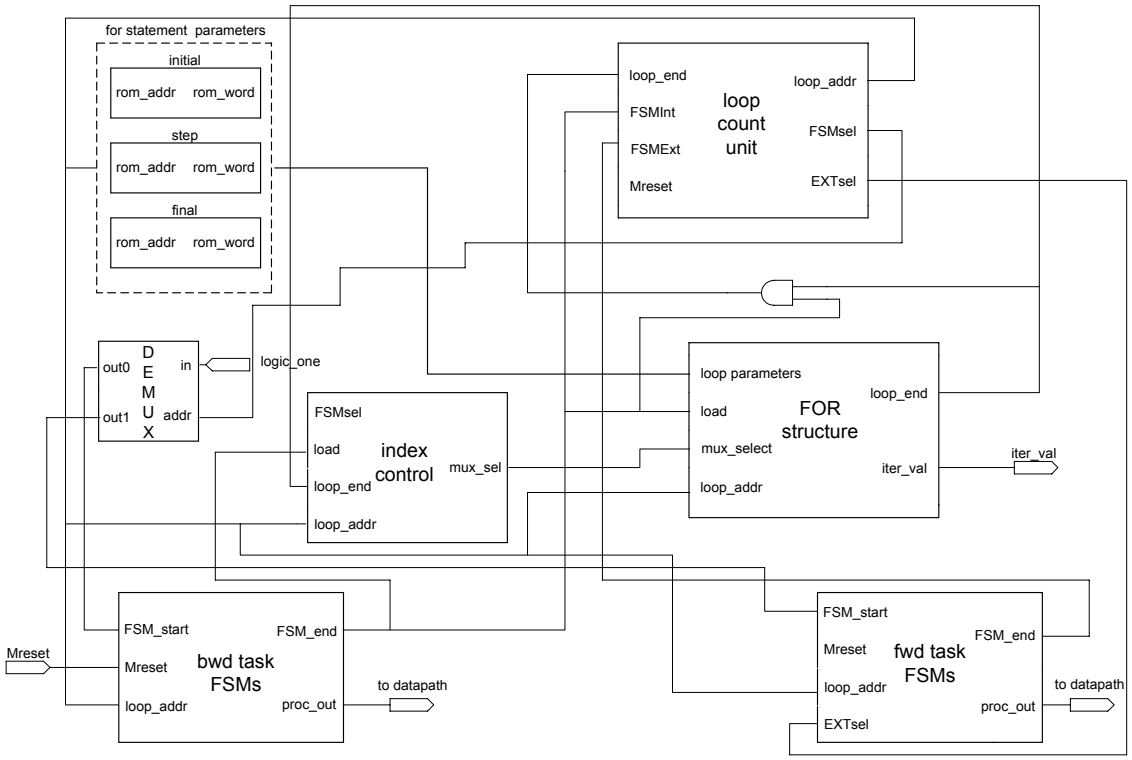


**Figure 2.** Control Unit Architecture

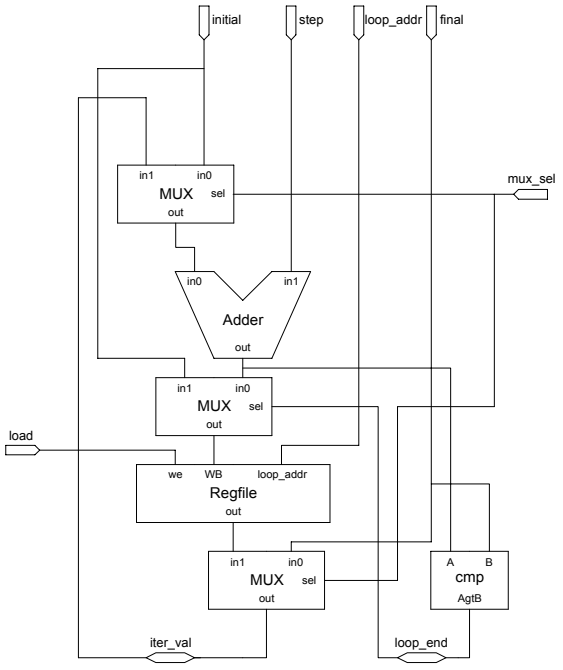


**Figure 3.** FOR structure block diagram

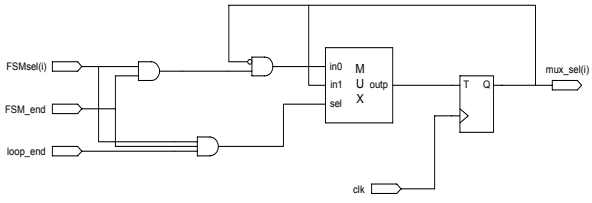


**Figure 4.** Index control bit slice

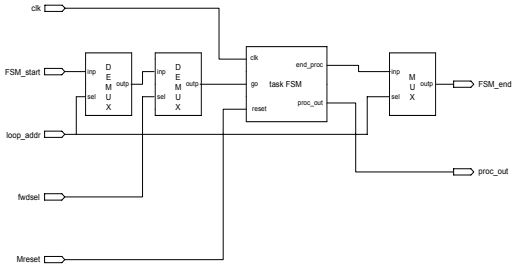


**Figure 5.** Fwd task FSM bit-slice. For bwd tasks, the fwdsel demultiplexing is omitted.

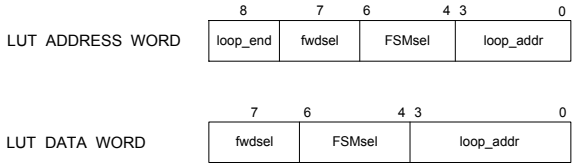


**Figure 6.** LUT address and data word formation

```
for i=0 to H-1, step B
  for j=0 to W-1, step B
    for k=0 to B-1, step 1
      for l=0 to B-1, step 1
        temp = 0
        for m=0 to B-1, step 1
          temp = temp + coeff[k,m] * image_in[i+m,j+l]
        end for
        image_out[i+k,j+l] = temp
      end for
    end for
  end for
end for
```

**Figure 7.** Pseudocode for the blocked matrix multiplication algorithm (H=144, W=176, B=8)