Efficient hardware looping units for FPGAs

<omitted for blind review>

***Abstract*—Looping operations impose a significant bottleneck to achieving better computational efficiency for embedded applications. To confront this problem in embedded computation either in the form of programmable processors or FSMD (Finite-State Machine with Datapath) architectures, the use of customized loop controllers has been suggested. In this paper, a thorough examination of zero-cycle overhead loop controllers applicable to perfect loop nests operating on multi-dimensional data is presented. The design of such loop controllers is formalized by the introduction of a hardware algorithm that fully automates this task for the spectrum of behavioral as well as generated register-transfer level architectures. The presented algorithm would prove beneficial in the field of high-level synthesis of architectures for data-intensive processing. It is also shown that the proposed loop controllers can be efficiently utilized for supporting generalized loop structures such as imperfect loop nests.**

The performance characteristics (cycle time, chip area) of the proposed architectures have been evaluated for FPGA target implementations. It is shown that maximum clock frequencies of above 230MHz with low logic footprints of about 1.4% of the overall logic resources can be achieved for supporting up to 8 nested loops with 16-bit indices on a modestly-sized Xilinx Virtex-5 device.

I. Introduction

Recent embedded systems are required to execute data-intensive workloads such as video encoding/decoding. For this reason, the respective market is dominated by general-purpose processor (ARM [1], MIPS32 [2]) and DSP architectures featuring architectural characteristics suitable to portable platforms (mobile phones, handheld gaming consoles etc). More and more often, embedded RISC families and DSPs involve customized features to data-dominated domains, where the most performance-critical computations occur in various forms of nested loops. Following this trend, modern DSPs provide better means for the execution of loops, by surpassing the significant overhead of the loop overhead instruction pattern which consists of the required instructions to initiate a new iteration of the loop.

An advantage of the emerging category of soft-core processors targeted to FPGAs is their configurability to fit specific user or application demands. Configurable processors such as Xtensa-LX [3] and the freely accessible Xilinx Microblaze [4], Altera Nios-II [5] and SPARC V8 LEON-3 [6] present this adaptability. However, neither of these processors is optimized for executing loop nests without cycle overheads. Usually zero-overhead mechanisms for single innermost loops are present.

In this work, an architectural approach to designing efficient parametric hardware looping units (HWLUs) targeted to FPGAs is presented that provide zero-cycle overhead implementations for perfect loop nests. Our solution significantly extends previous work in the field [7]. An algorithm is presented that can be used for realizing a behavioral-style model of the hardware looping unit and can also be adapted in terms of a module generator for the same purpose. Further, potential uses and extensions of the HWLU design are discussed for the support of irregular loop structures such as imperfect loop nests. The hardware looping designs and generators presented in this paper are available as part of the Opencores “hwlu” project [8].

The remainder of this paper is organized as follows. Section II overviews previous research on the subject. In Section III, the HWLU architecture is presented from a hardware point of view. Section IV presents an algorithm for modeling and generating parameterized HWLU designs and Section V unveils potential extensions and uses of the basic architecture. Section VI discusses area and timing characterization of FPGA implementations for HWLU variants and provides comparisons to a well-known zero-overhead looping architecture. Finally, Section VII summarizes the paper.

II. Related work

In other approaches, looping cycle overheads are confronted by using branch-decrement instructions, zero-overhead loops or customized units for more complex loop nests [7], [9]–[11]. For the XiRisc processor [9], branch-decrement instructions can be configured prior synthesis. Some DSP cores support a configurable number of hardware looping units, which can handle the case of perfect loop nests with fixed iteration counts [10]. Configurable processors as Xtensa [3] incorporate zero-overhead mechanisms for single innermost loops only. Unfortunately, a specific processor template is provided to which alternate control-flow mechanisms cannot be added.

Closer to our work is the dedicated controller for perfect loop nests found in [7]. Its main advantage is that successive last iterations of nested loops are performed in a single cycle. As it is originally presented, only fully-nested structures are supported and the area requirements for handling the loop increment and branching operations grow proportionally to the considered number of loops. In our work, formalized algorithms are presented for designing a corresponding HDL model and a code generator for an extension of this controller.

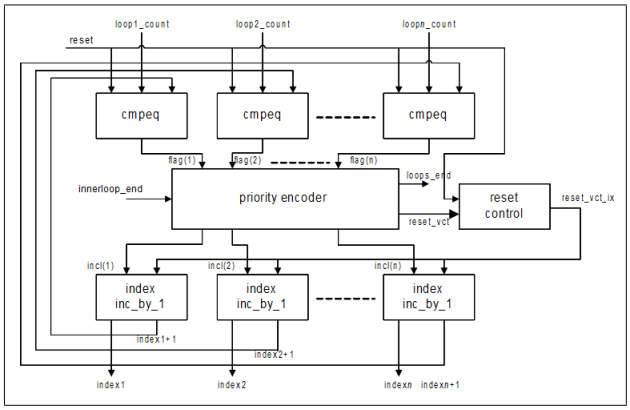


Fig. 1: Block diagram of the hardware looping unit.

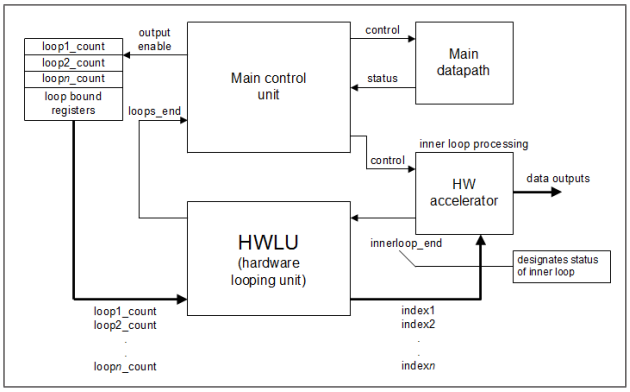


Fig. 2: Usage of the hardware looping unit in a programmable processor.

The proposed technique can be also contrasted to the ZOLC (Zero-Overhead Loop Controller) method [12], [13]. The main advantage of ZOLC is the accomodation of complex loop structures with multiple-entry and multiple-exit nodes while eliminating most cases for loop overheads. The ZOLC has been introduced and applied on both non-programmable architectures [12] and the XiRisc processor [9], [13].

The ZOLC and the proposed hardware looping unit, can be evaluated in terms of the tradeoff between better cycle performance (in favour of HWLU) and more efficient hardware use (for ZOLC). While with ZOLC, a complex loop structure with an arbitrary number and combination of loops can be controlled, by using a single process unit (one adder, one comparator etc), HWLU demands this hardware replicated for each loop. In this paper, we show that the HWLU has small hardware demands on modern FPGAs. Further, its exact performance benefits are evaluated in context of data-intensive algorithmic kernels of image and video processing standards. A number of extensions for the use of HWLU-enabled control automatons is also presented, with focus on controlling arbitrary loop nests and supporting polyhedral computation in hardware.

III. The hardware looping unit (HWLU) architecture

The hardware looping architecture (HWLU) naturally can incorporate any number of levels of loop nesting in hardware to eliminate branch instruction overhead for loop increments. The user can re-generate the corresponding files for modules `hw_looping` (structural) and `priority_encoder` (rtl) for a different number of supported loops. Fig. 1 shows the block diagram of the hardware looping architecture.

Loop index values are produced every clock cycle based on the loop bound values (possibly read from a lookup table) for each level of nesting. The initial value for the loop indices is provided by a reset mechanism, and the maximum value is equal to the loop bound minus one. In the following cycle of a last iteration for a specific loop, the loop index is reset to its initial value.

The priority encoder performs the actual control logic in context of the HWLU and operates asynchronously by detecting the equality comparators (`cmpeq`) outputs (bit-wise flag signals) and an external signal from the datapath (`innerloop_end`). This signal is produced by the corresponding hardware module that performs the inner loop operations, which may be a dedicated accelerator engine.

If a specific loop is terminating, this loop as well as all its inner loops are reset during the subsequent cycle. For a non-outermost loop, its immediate parent loop index is incremented. In case that none of the loops is terminating, then the inner loop is incremented. Signal `innerloop_end` guards this increment operation.

Finally, signal `loops_end` designates that processing in the entire loop structure has terminated, and is read by the main control unit of the microprocessor or FSMD.

A. Use case: The hardware looping unit within a programmable processor

Fig. 2 indicates a possible design of a control unit used in a programmable instruction set processor. It is implied that the register architecture of the processor is partitioned, so that the loop index registers are stored into dedicated registers (the register bank comprised by incrementer units in the simplest case) and a general-purpose register file (not shown here) is used for stiring other program variables.

As can be seen, control-dominated segments of the user program are implemented in the main datapath, which communicates through control and status channels with the main control unit. When appropriate, the main control unit activates the hardware acceleration datapath unit. Also, at that time, the output enable input to the loop bound register bank shown in the figure is active, so that the `loop_bound` value can be read by the HWLU. In our example, this unit performs all the inner loop processing. All index variables, are made available to the acceleration unit so that high-bandwidth data and address computation can be serviced as needed. When its operation terminates, the HWLU is acknowledged through the `innerloop_end` asynchronous flag. On an active `loops_end` signal, which occurs when the loop structure is exited, the main control unit pauses the HWLU e.g. by deasserting the output enable signal to the loop bound values lookup table.

```
IXGEN − B :
  local temp_index: temporary copy of index.
  parameter NLP: number of supported loops.
  parameter DW: index register width.
begin
  if innerloop_end equals 1 then
    for i in NLP downto 1 do
      if temp_index[i · DW-1:(i-1) · DW] less than
        loop_count[i · DW-1:(i-1) · DW] then
        if i less than NLP then
          initialize temp_index[NLP · DW-1:i · DW]
        endif
        increment temp_index[NLP · DW-1:i · DW] by stride
        exit for loop
      endfor
      if temp_index greater than or equal loop_count then
        clear temp_index[NLP · DW-1:0]
        loops_end ← 1
      endif
    endif
  endif
end
```

Fig. 3: Pseudocode for the IXGEN-B algorithm.

IV. Hardware algorithm for zero-overhead looping on perfect nests

In this section, a hardware algorithm is introduced for automating the design of compact and efficient hardware looping units that can be implemented as fully synchronous hardware. The looping units of this type are hereafter termed as 'index generators', and abbreviated to *IXGEN* which is also used when referring to the algorithm. These units can be also viewed as tuple generators, covering the entire space of d-tuples for d-dimensional data processing [14]. Such formal descriptions of micro-architectural looping operations can be exploited in the scope of high-level synthesis for FSMD-based processors.

The first form of the algorithm, named *IXGEN-B*, is directly applicable in context of a behavioral HDL model for any number of loops. The pseudocode semantics for implementing these mechanisms can be found in Fig. 3. The same I/O interface to the non-systematic design of Section III is also used here. Thus, *loop_count* and *index* are vectorized forms of the set of loop bound values and the current iteration vector, correspondingly.

When the data processing in the inner loop is completed, *innerloop_end* is asserted and a cascaded set of comparisons between index registers to their corresponding loop bound values is activated. The flow of comparisons is directed from outermost to their immediately innermost loops. If the index value is less than the loop bound for a given loop i, the index is incremented by a stride value, while all its outer loops are set to the initial index values. After the first successful comparison, the cascaded structure is prematurely exited in a form similar to the `break` statement of the C programming language. Given that the cascaded comparisons fail, an index value which is lexicographically larger or equal to *loop_count* signifies the end of processing in the loop nest.

The second form of the algorithm, named *IXGEN-R*, describes an HDL code generator of an equivalent index generation unit. Its main difference lies in the fact that it uses a priority encoded scheme that cannot be specified in a parameterized manner using natural HDL semantics. The pseudocode semantics of algorithm IXGEN-R can be found in Fig. 4. Here, the temporary signals *temp$n$_index* and *loop_count$n$* are used where n is the current loop enumeration. In the generated HDL code, these signals are defined as aliased names of elements of the *index* and *loop_count* vectors, respectively. It should be noted than all lines featuring a call to the `PRINT` routine illustrate emitted code.

```
IXGEN − R :
  local temp_index: temporary copy of index.
  alias temp_indexi: i-th segment of temp_index.
  alias loopi_count: i-th segment of loop_count.
  parameter NLP: number of supported loops.
begin
  PRINT(if innerloop_end = 1 then)
  for i in NLP downto 1 do
    if i equals NLP then
      PRINT(if temp_indexi <= loopi_count then)
      PRINT(increment temp_indexi by stride)
    else
      PRINT(elsif temp_indexi <= loopi_count then)
      for j in NLP downto i+1 do
        PRINT(initialize temp_indexj)
      endfor
      PRINT(increment temp_indexi by stride)
    endif
  endfor
  PRINT(clear temp_index)
  PRINT(loops_end ← 1)
  PRINT(endif)
  PRINT(endif)
end
```

Fig. 4: Pseudocode for the IXGEN-R module generation algorithm.

The VHDL description for the index generator of a triple perfect loop nest ($NLP = 3$) with unitary stride values is shown in Fig. 5.

V. Extensions

Potential extensions of the hardware looping units presented in this paper are highlighted in the following paragraphs.

A. Scanning integer points of polyhedra

Let us consider the three-dimensional polyhedron defined by the following set of inequalities. The corresponding implementation of a scanning routine either in software or in hardware would have to visit all the integer points that define the polyhedron.

$$0 \leq i \leq n$$
$$0 \leq j \leq n$$
$$0 \leq k \leq i + j$$

Typically, the scanning code could be automatically generated by an appropriate tool (Cloog [15]) in the form of three nested loops. It is easily seen that the upper bound for the inner loop is not static since it depends on the value of

```
signal temp_index : std_logic_vector(NLP*DW-1 downto 0);
alias  temp_index1: std_logic_vector(DW-1 downto 0) is
       temp_index(1*DW-1 downto 0*DW);
alias  loop1_count: std_logic_vector(DW-1 downto 0) is
       loop_count(1*DW-1 downto 0*DW);
...
  process (clk, reset, innerloop_end, temp_index, loop_count)
  begin
  ...
  elsif (clk'EVENT and clk = '1') then
    if (innerloop_end = '1') then
      if (temp_index3 < loop3_count) then
        temp_index3 <= temp_index3 + '1';
      elsif (temp_index2 < loop2_count) then
        temp_index3 <= (others => '0');
        temp_index2 <= temp_index2 + '1';
      elsif (temp_index1 < loop1_count) then
        temp_index3 <= (others => '0');
        temp_index2 <= (others => '0');
        temp_index1 <= temp_index1 + '1';
      else
        temp_index <= (others => '0');
      end if;
    end if;
  end if;
```

Fig. 5: Partial VHDL description of the index generation unit for NLP=3.

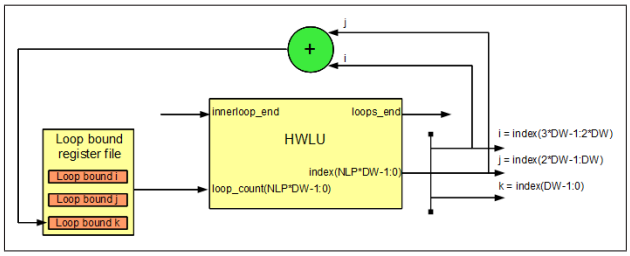


Fig. 6: Looping control hardware for scanning the aforementioned polyhedron.

indices i, j and thus it cannot be determined at compile time. In this case, the HWLU serves as part of the necessary control logic, requiring only limited additions. Fig. 6 illustrates an appropriate hardware implementation, utilizing an adder for computing the $i + j$ sum. The resulting value is then stored back in an external register bank for the loop bound entries.

This approach can be easily extended to more intriguing cases such as unions of polyhedra that are of certain interest in the field of high-level synthesis.

B. Using the hardware looping unit in general loop structures: Full Search Motion Estimation

The HWLU approach can be used for implementing the Full-Search Motion Estimation ($fsme$) algorithm. Motion estimation is used in MPEG video compression for removing the temporal redundancy in a video sequence. Compression is achieved by encoding only the displacement values of pixel blocks (motion vectors) between successive frames. The calculation of the motion vector is performed by a cost function minimizing the prediction error.

In Fig. 7, the pseudocode of the $fsme$ algorithm is shown. It consists of three double nested loops incorporating the data processing tasks of the algorithm, denoted by labels of the form Tnum, where num is a positive integer. The iteration

```
for (x = 0; x <= H-B; x += B) {
  for (y = 0; y <= W-B; y += B) {
T1: min = 255 * B * B;
    for (i = -p; i <= p; i++) {
      for (j = -p; j <= p; j++) {
T2:     dist = 0;
        for (k = 0; k <= B-1; k++) {
          for (l = 0; l <= B-1; l++) {
T3_1:       p1 = current[x+k, y+l];
T3_2:       if (p2 out of picture borders) {
              p2 = 0;
            } else {
              p2 = reference[x+i+k, y+j+l];}
T3_3:       dist = dist + abs(p1 - p2);}}
T4:     if (dist < min) {
          MVx[x, y] = i;
          MVy[x, y] = j;}}}}}
```

Fig. 7: C-like pseudocode for the full search motion estimation ($fsme$) kernel.

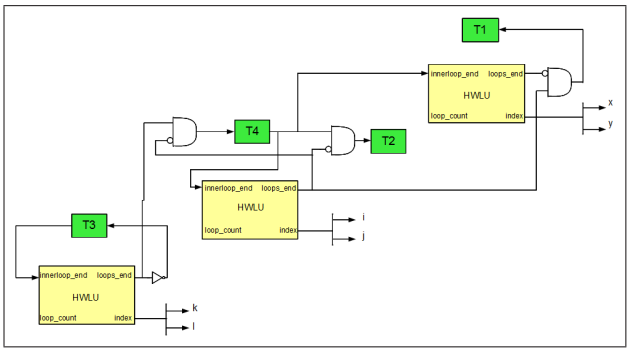


Fig. 8: Sketch of motion estimation hardware using three HWLU modules.

vector fully describing the loop nest is (x, y, i, j, k, l). The outer (x, y) loops select the block from the current picture for which the minimum motion vector is calculated. By iterating (i, j), each time a reference block is selected from the reference window. Initially, the dist variable is cleared, in order to accumulate the distance metric for the selected block. For each position in the search region, the distance kernel is executed, and this is performed for all (k, l) pixels in the current picture block.

The data processing tasks are summarized as follows:

T1: Initializes the min variable
T2: Initializes the dist variable
T3: Sum of absolute differences (SAD) criterion, divided into subtasks $T3_1$ to $T3_3$
T4: Updates the (i, j) motion vector when a new min value is found.

A high-level view of a motion estimator design using HWLU modules is shown in Fig. 8. Each double loop nest is assigned its dedicated HWLU instance. Updating the iteration vector is enabled by the termination of tasks T3 and T4 which are positioned at a closing position for a loop [13]. It can be seen that T1 and T2 do not affect the update of the iteration vector. The respective index registers that provide the physical implementation of the iteration vector are stored within the corresponding HWLUs.

VI. Performance evaluation of the hardware looping units

As it has been mentioned before, the proposed hardware looping units are easily adaptable to both programmable and FSMD-like architectures. In order to assess the performance of the HWLU, IXGEN-B and IXGEN-R hardware looping units for perfect loop nests, they are evaluated over the entire parameter set for the following value set; $NLP : 1 - 8$ and $DW : 8, 12, 16$. Since most of the benchmark applications deal with image or video manipulation, they usually operate on two-dimensional pixel data in multiple nested loop schemes. Further, the examined index register widths are realistic since they correspond to the size of the horizontal and vertical dimensions of digital images (e.g. as obtained from digital still cameras) which are typically within this range.

For each point in the parameter set, the timing (maximum clock frequency) and area requirements are measured for a representative FPGA process. The logic synthesis tool used is Xilinx Webpack ISE 9.2i.

Throughout the evaluations, the XC5VLX50 device (FF665 package and '-1' speed grade) which is one of the smallest available Virtex-5 devices. The maximum capacity of XC5VLX50 is 7,200 slices (which translates to 28,800 6-input LUTs), 96 18-kbit block RAMs (BRAMs) and 48 DSP48E datapath blocks. Both BRAMs and DSP48E blocks remain unused by the looping logic.

A. Speed measurements

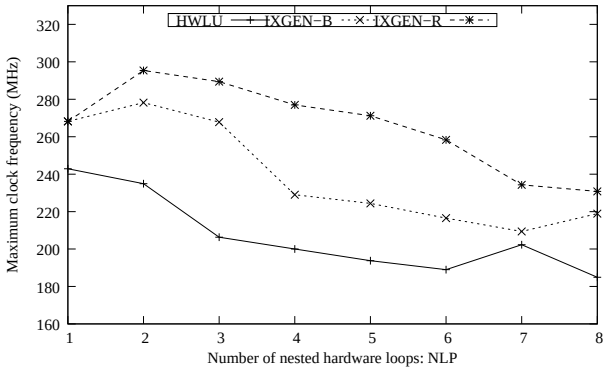
All three variants of the hardware looping architecture (HWLU, IXGEN-B, and IXGEN-R) have been designed in VHDL and synthesized for XC5VLX50. Fig. 9 depicts the maximum clock frequency estimates for different number of supported maximum number of loops (NLP=$\{1 \ldots 8\}$) and for different index register widths (DW = 8, 12 or 16).

It is obvious that the original IXGEN-R design is the most efficient in terms of maximum clock frequency: the corresponding performance margins are 20.3% against HWLU and 9.5% against IXGEN-B. Even more important is the fact that the IXGEN-R design achieves nearly unvarying performance for different index register widths. The latter is due to the fact that the synthesis tool efficiently balances the index increment logic for the prioritized cases, the evaluation of which has the same logic depth in an FPGA implementation. Both the HWLU and the IXGEN-B designs donnot scale gracefully with increased values of DW, since the synthesis tool infers cascaded logic.

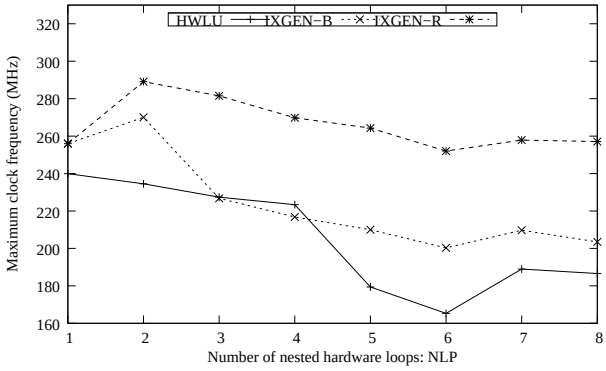
B. Chip area measurements

The chip area requirements are shown in Fig. 10.

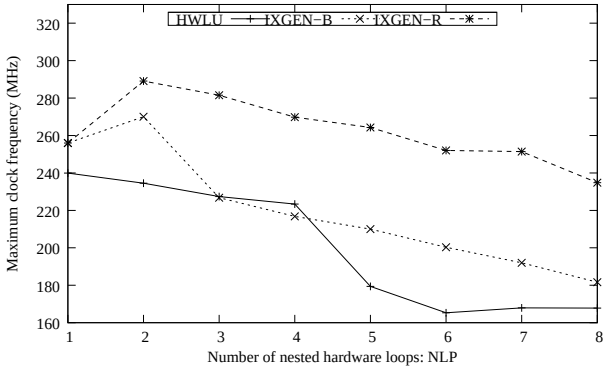
When compared regarding the area requirements, it is shown that HWLU presents the lowest demands for $DW = 16$ while the IXGEN-R design is better for smaller DW values. The HWLU is smaller by a factor of 32.9% than IXGEN-B for all cases and 18.3% than IXGEN-R for $DW = 16$. This observation can be explained by taking account the sparsely populated logic slices in the HWLU design for the small



(a) DW = 8 bits.



(b) DW = 12 bits.



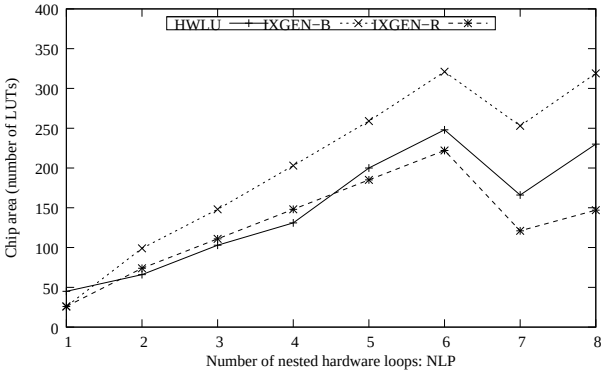
(c) DW = 16 bits.

Fig. 9: Maximum clock frequency for the hardware looping units (Xilinx FPGA XC5VLX50-FF665-1).

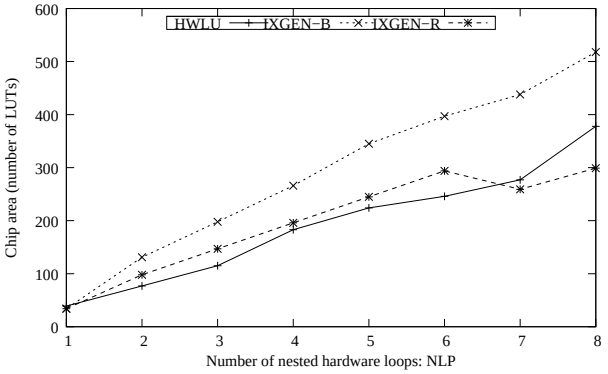
DW values. Many of these slices get populated when DW is increased and hardware exploitation for HWLU is significantly improved. On the contrary, the IXGEN-B and IXGEN-R designs feature more compact descriptions that leave no room for such behavior.

C. Comparison of the proposed hardware looping scheme against ZOLC

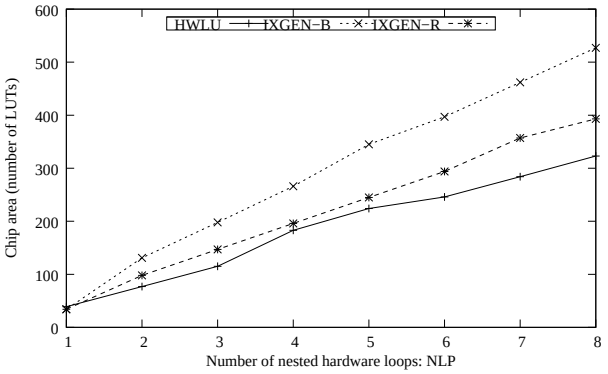
In addition to the full-search motion estimation (*fsme*) benchmark, an application set consisting of three more benchmarks was used for verification and performance comparisons of the proposed hardware looping units (HWLU) against ZOLC [13]. *fsme_dir* is a data layout optimized version of



(a) DW = 8 bits.



(b) DW = 12 bits.



(c) DW = 16 bits.

Fig. 10: Chip area in number of LUTs for the hardware looping units (Xilinx FPGA XC5VLX50-FF665-1).

fsme transformed by appropriate data-reuse transformations [16]. *matmult* is the block-based matrix multiplication, and *rcdct* the row-column decomposition DCT algorithm. All kernels were tested on CIF-compatible (352×288) frames extracted from YUV video sequences. For all applications, hand-optimized datapath units for each data processing task were designed in VHDL.

Table I summarizes the experimental results. It can be seen that the HWLU approach is competitive to ZOLC with a small (1.75%) performance advantage. It is assumed here that both looping units are parts of SoC-level designs, so that they donnot determine the critical path.

TABLE I: Performance results for the examined applications.

Benchmark	Number of loops	Cycles with HWLU	Cycles with ZOLC	%diff
fsme	6	68696549	70128467	2.04
fsme_dr	20	49215771	50759199	3.04
matmult	5	1926158	1940451	0.74
rcdct	18	6488100	6565753	1.18

VII. Conclusion

In this paper, a hardware looping architecture and its potential uses and extensions for data-intensive processing in embedded systems is introduced. The presented architecture is able to provide all necessary control means for executing perfect loop nests without any cycle overhead for updating the iteration vector. Out of three different variants of the architecture, the most efficient is denoted regarding timing and area characterization results. When speed comes in mind, the architecture adhering to the IXGEN-R algorithm is more efficient, while for larger index register widths, a lower-level mixed structural-RTL design is more area efficient (HWLU). The cycle performance of the proposed architecture is always better than the ZOLC [13] architecture due to simultaneous multiple-index update in perfect loop nests. While ZOLC has a much broader context, certain techniques can be applied that augment the use of HWLU-like architecture even on general loop nests.

References

[1] ARM ltd. [Online]. Available: http://www.arm.com
[2] MIPS technologies inc. [Online]. Available: http://www.mips.com
[3] R. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, March-April 2000.
[4] Xilinx home page. [Online]. Available: http://www.xilinx.com
[5] Altera home page. [Online]. Available: http://www.altera.com
[6] Aeroflex Gaisler research. [Online]. Available: http://www.gaisler.com
[7] D. Talla, L. K. John, and D. Burger, "Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements," *IEEE Trans. Comput.*, vol. 52, no. 8, pp. 1015–1031, August 2003.
[8] N. Kavvadias. Hardware looping unit. [Online]. Available: http://www.opencores.org/project,hwlu
[9] F. Campi, R. Canegallo, and R. Guerrieri, "IP-reusable 32-bit VLIW RISC core," in *Proceedings of the 27th European Solid-State Circuits Conference*, September 2001, pp. 456–459.
[10] M. Kuulusa, J. Nurmi, J. Takala, P. Ojala, and H. Herranen, "A flexible DSP core for embedded systems," *IEEE Des. Test. Comput.*, vol. 3, no. 4, pp. 60–68, October 1997.
[11] J.-Y. Lee and I.-C. Park, "Loop and address code optimization for digital signal processors," *IEICE Trans. Fund. Elec., Comm. and Comp. Sc.*, vol. E85-A, no. 6, pp. 1408–1415, June 2002.
[12] N. Kavvadias and S. Nikolaidis, "Zero-overhead loop controller that implements multimedia algorithms," *IEE Computers and Digital Techniques*, vol. 152, no. 4, pp. 517–526, July 2005.
[13] ——, "Elimination of overhead operations in complex loop structures for embedded microprocessors," *IEEE Trans. Comput.*, vol. 57, no. 2, pp. 200–214, Feb. 2008.
[14] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations*, 2005.
[15] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, France, September 2004, pp. 7–16.
[16] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecapelle, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Boston: Kluwer Academic Publishers, 1998.