A portable specification of zero-overhead looping control hardware applied to embedded processors

Nikolaos Kavvadias and Spiridon Nikolaidis
Section of Electronics and Computers, Department of Physics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
Email: {nkavv,snikolaid}@physics.auth.gr

***Abstract*— Looping operations impose a significant bottleneck to achieving better computational efficiency for embedded applications. To confront this problem on embedded RISC processors, an architectural modification involving the integration of a zero-overhead loop controller (ZOLC) has been suggested, supporting arbitrary loop structures with multiple-entry and multiple-exit nodes. In this paper, a graph formalism is introduced for representing the loop structure of application programs, which can assist in ZOLC code synthesis. Also, a portable description of a ZOLC component is given in detail, which can be exploited in the scope of RTL synthesis, compiler optimizations or assembly level transformations for enabling its utilization. This description is designed to be easily retargetable to single-issue RISC processors, requiring only minimal effort for this task.**

I. Introduction

Recent embedded microprocessors are required to execute data-intensive workloads like video encoding/decoding. For this reason, the respective market is dominated by 32-bit RISC architectures (ARM, MIPS32) that continuously evolve their features providing among others deeper pipelines, compressed instruction sets and support for saturation arithmetic and subword parallelism. At the DSP end (Motorola 56300, ST120, TMS320C54x), architectures involve even more specialized architectural characteristics suitable to multimedia processing, i.e. that 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.

In this work, a portable specification of a zero-overhead loop control (ZOLC) unit is presented, which can be utilized for the automatic generation of related DSP enhancements on embedded processors, in addition to a region-based control-flow formalism for software synthesis of ZOLC initialization and configuration. The proposed ZOLC unit is used at the instruction fetch stage to eliminate the loop overheads and can be applied to an arbitrary combination of loops. The unit has already been incorporated to the XiRisc 32-bit configurable microprocessor [1], [2], but in this paper we focus in formalizing ZOLC mechanisms by deploying a reusable description in C-like pseudocode.

II. Related work

In recent literature, looping cycle overheads are confronted by using branch-decrement instructions, zero-overhead loops or customized units for more complex loop nests [3]–[6].

This approach is encountered in both academic [1], [7], and commercial processors and DSPs. For the XiRisc processor [1], 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 [7]. Configurable processors as the ARC and Xtensa [8] incorporate zero-overhead mechanisms for single e.g. innermost loops only. Unfortunately, a specific processor template is provided to which alternate control-flow mechanisms cannot be added.

The greatest disadvantage of these solutions is that they are only capable of handling canonical forms of nests consisting of single-entry static loops. Non-perfectly nested loops are not generally supported, while irreducible control-flow regions (loops with multiple entries due to explicit control transfers) are not detectable by natural loop analysis, while their conversion to reducible regions can be costly in terms of code size. Also, the latter techniques [9] can only be found in sophisticated compilers. Irreducible loops can be produced when compiling to C from flowchart specifications, for example in translating from process description languages like UML.

Closer to our work, a dedicated controller for perfect loop nests is found in [4]. Its main advantage is that successive last iterations of nested loops are performed in a single cycle. In contrast to our approach, 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. Also, this unit cannot be efficiently used with any datapath since a certain parallelism is assumed to perform several operations per cycle.

In our approach, a ZOLC method that accommodates complex loop structures with multiple-entry and multiple-exit nodes is introduced and applied on an existing RISC processor. With our method, complex loop structures with bound or stride values unknown at compile-time are supported, without regard for the related compiler optimization capabilities.

III. Task control flow graph representation of loop-intensive applications

For the task of code generation, the control-data flow graph (CDFG) of a given program is processed, which is the CFG [10] with its nodes expanded to their constituent instructions.

In order to let the ZOLC engine execute looping operations in the background, it is required to: a) remove the loop overhead instructions from the original CDFG, b) generate instructions for initializing the ZOLC storage resources and c) generate the ZOLC initialization sequence executed prior entering a selected loop nest, insert instructions for dynamic updates of loop bound and stride values, and handle dynamic control flow decisions occuring at outermost if-then-else constructs. To determine the parameter values of the optimal ZOLC configuration (number of loops, and entries/exits per loop) under area constraints, a design space exploration procedure is imposed, with inevitable iterations to evaluate these alternatives. To evaluate a specific configuration, the computational complexity associated with performing all the steps from a) to c) at the CDFG level can be reduced by performing b) and c) on a more convenient graph representation of the application program.

This representation should only model the control transfer expressions $(CTEs)$ among control-flow regions situated at loop boundaries. In our context, these regions are termed as the Data-Processing Tasks $(DPTs)$ of the algorithm. This graph structure is the Task Control Flow Graph (TCFG) and is defined as follows:

Definition 1: We call $TCFG(V \cup V', E)$ the directed cyclic graph representing the control flow in an arbitrarily complex loop nesting of an application program; each node V represents exactly one primitive DPT; each node V' represents one composite DPT that results from applying a hardware-dependent transformation operator on a DPT subset; the edges E represent control dependencies among data-processing tasks.

Tasks that do not include a loop overhead instruction pattern are designated as *primitive forward tasks*, while those including exactly one such pattern are termed as *primitive backward tasks*. The remaining tasks are attributed as *composite tasks*, and can be introduced as a result of graph transformations applying hardware-dependent rules. Such case is explained at the end of this section.

Constructing the TCFG requires loop analysis on the input program by the compiler. Only a few experimental compiler infrastructures seem to support direct multiple-entry loop detection [11].

As a motivational example, the full-search motion estimation kernel (*fsme*) is considered, which is used in MPEG compression for removing the temporal redundancy in a video sequence. The algorithm consists of six nested loops and its TCFG is shown in Fig. 1 (graph layout obtained by [12]). Primitive backward tasks are denoted as bwd_i, where i is the enumeration of the loop nest, starting from one as the zero-th level is preserved for the predecessor and successor statements to the nest. Composite backward tasks can be distinguished by a range notation with $bwd_m - bwd_n$ or a list notation with $\{bwd_m,\ldots,bwd_n\}$, the latter if they consist of non-consecutive backward tasks. Forward tasks are denoted as $fwd_i(j)$, where i is the loop number and j selects a specific task of this type from the i-th loop. This formulation to distinguish the task types is used for the internal data structures of a compiler pass for forming TCFGs. The frequently encountered loopend

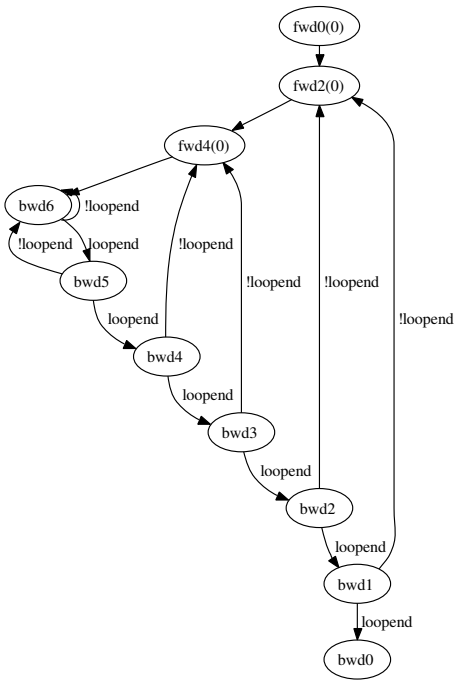

Fig. 1. TCFG for the full-search motion estimation kernel.

signal denotes a loop termination condition at the time execution resides in a bwd task. This signal would be produced by ZOLC to drive task switching.

An interesting application of the presented ZOL program control is its synergistic use with other hardware techniques. Such case is when supporting the update of multiple indices within a single clock cycle [4]. For the examined application (*fsme*), four consecutive backward tasks are merged to the 'bwd1-bwd4' composite node. Thus, a number of loop end signals equal to the number of loops in the kernel are required. In Fig. 2, the rules implemented by a primitive and the referred backward task are shown. In the latter case, the corresponding transformation unit is able to restructure a given TCFG of an algorithm with n loops by merging consecutive primitive backward tasks.

IV. RTL-LEVEL SPECIFICATION OF THE ZOLC MECHANISM

The purpose of ZOLC is to provide a proper candidate program counter (PC) target address to the PC decoding unit for each substituted looping operation. Typically, the design of the instruction decoder, the PC decoding unit and the register file architecture require modifications in presence of ZOLC hardware. ZOLC is composed from the task selection unit, which determines the appropriate next PC value when execution resides in a loop structure, the loop parameter tables where the loop bound and stride values are kept and the index calculation unit.

Two modes of operation are distinguished from the ZOLC side. In "initialization" mode, the ZOLC storage resources are initialized. In "active" mode, the ZOLC: a) determines the following task, b) it issues a new target PC value and a set of candidate exit values for the case of multiple-exit loops to PC

```
PrimitiveBackwardTask()
begin
  useful computation operations in this task;
  if m equals loop-final[m] then
    reset m to loop-initial[m]
  else
    increment m
end

CompositeBackwardTask()
begin
  useful computation operations in this task;
  if m equals loop-final[m] then
    m..n := loop-initial[m]..loop-initial[n]
  elsif m+1 equals loop-final[m+1] then
    increment m
    m+1..n := loop-initial[m+1]..loop-initial[n]
  ...
  elsif n equals loop-final[n] then
    increment n-1
    n := loop-initial[n]
  else
    increment n
end
```

Fig. 2. Index update in primitive and composite backward tasks, respectively.

decode, c) loop indices are updated and written back either to specified registers of the integer register file or to a separate index register bank. The task sequencing information is stored in a LUT within the task selection unit. On completion of a data processing task, a task end signal is issued from PC decode, and an entry is selected from the LUT to address the succeeding data processing task and the loop parameter blocks, based on which task has completed and the status of the current loop. It should be noted that data-dependent instruction latencies (e.g. for the MUL instruction on ARM7, 2 to 4 execution cycles are required for completion) complicate ZOLC operation. Thus, it is assumed that the compiler will either move the offending instruction from the last position in a given task, or that a NOP is inserted. The initial, final and step loop parameters are used to calculate the current index value and determine if a loop has terminated.

Pseudocode semantics for implementing ZOLC mechanisms in context of the instruction fetch stage of typical single-issue RISC processors (Fig. 3) can be found in Fig. 4. Such formal description of ZOLC micro-architectural level operations can be exploited in the scope of high-level synthesis for DSP-friendly ASIPs. The ZOLC behavior is accessed during the PC decoding operations to determine the successive PC value and can be decomposed into the $LoopParams$, $IndexCalculation$, and $TaskSelection$ procedures. Table I summarizes the notation used for signals and storage resources of ZOLC. The required storage resources are: the task selection LUT ($ttlut\_m$), the index register bank ($IXRB$), the dynamic flag register bank consisting of 2-bit entries that encode information for fwd tasks ($DFRB$), a status register for index calculation ($muxsel\_m$) and the loop parameter registers ($lparams\_m$). Also, $NUM\_ENTRIES$ and NUM_EXITS dedicated register files are required for determining the PC target for the following task and the candidate PC exit values for the multiple-exit condition, respectively.

TABLE I

NOTATION USED IN THE PSEUDOCODE OF FIG. 3-4.

Name	Description
{*name*}_*a*	Address signal
{*name*}_*d*	Data signal
{*name*}_[*ent—ext*]	Referring to loop entry/exit
{*name*}_*m*	Register (file)/memory resource
{*name*}_*t*	Temporary
*.{*name in caps*}	Field {name}
*.{*name in lowcase*}	Column select for 2D arrays

```
PCDecoding() // PC decoding operations
begin
  for i in 0..NUM_EXITS-1 do
    if PC equals PCext_m[task_d].i then
      taskend_t[i] := 1
    else
      taskend_t[i] := 0

  N = NUM_ENTRIES = NUM_EXITS
  zolc_trg := encoder-N-to-log2-N(taskend_t)
  taskend := selector-N-to-1(zolc_trg)

  if PC_task_ext equals PC and ZOLC enabled then
    LoopParams()
    IndexCalculation()
    TaskSelection()
    PC := PC_zolc
  elsif a branch or jump has occured then
    usual PC decoding operations
  else
    PC := PC + word-size-in-bytes
end
```

Fig. 3. PC decoding operations for a ZOLC-enhanced processor.

V. EXPERIMENTS

The portable RTL specification of ZOLC has been added to the instruction- and cycle-accurate models of both the XiRisc processor [1], [2] and an in-house ASIP written in ArchC [13]. The ZOLC was also incorporated in the VHDL description of the XiRisc and the reported execution time speedups were about 27% while the processor cycle time was not affected. Regarding the ASIP, instruction-set extensions as the 'absolute value difference and accumulation' ($dabsa$) have been added in order to accelerate the execution of performance-critical kernels, for example the SAD criterion in motion estimation. The average basic block and DPTG sizes were also reduced due to the complex instructions.

A small set of 2 synthetic and 3 real applications have been examined in the case of the ASIP architecture, and the corresponding results are shown in Table II. The average task size and ZOLC initialization cycles are also given. In case of variable loop bounds and other types of loop dependencies, additional operations require a number of ZOLC-update cycles, which however never have to occur in tasks within inner-most loops.

Performance speedup was found at 44.15% in average, while speedups of about 75% can be obtained for the extreme

TABLE II

PERFORMANCE RESULTS FOR THE EXAMINED APPLICATIONS.

Benchmark	Avg. task size	Init. cycles	Cyc. with ZOLC	Cyc. without ZOLC	%diff
loop4	1	48	12,786	52,092	75.45
loop8	1	92	1,368,237	5,658,686	75.82
fsme	2.3	58	51,791,075	76,144,025	31.98
matmult	1.6	47	5,184,473	8,992,559	42.35
fsp6c1	2.23	164	20,296,016	48,470,213	58.13

case of one useful computational operation per task for the considered ASIP (looping overhead instruction pattern executes in 4 cycles).

VI. CONCLUSION

In this paper, a zero-overhead loop controller suited to embedded RISC microprocessors is introduced. The presented architecture is able to execute complex loop nests, even incorporating irreducible control-flow regions, with no cycle overheads incurred for task switching. A novel graph representation, namely the data-processing task graph has been described, that can be used for ZOLC code generation. Until now, the proposed mechanisms have been documented in VHDL, ArchC, and RTL pseudocode, extensive tests have been applied and performance measurements have been obtained for representative target applications. Overall, execution time improvements of 27% and 44.1% have been observed for the XiRisc processor and an in-house ASIP, respectively.

REFERENCES

[1] 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.

[2] N. Kavvadias and S. Nikolaidis, "Hardware support for arbitrarily complex loop structures in embedded applications," in *Proc. Design, Automation and Test in Europe Conf.*, March 7-11 2005, pp. 1060–1061.

[3] STMicroelectronics, *ST120 DSP-MCU Core Reference Guide*, 1st ed.

[4] 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.

[5] J. Kang, J. Lee, and W. Sung, "A compiler-friendly RISC-based digital processor synthesis and performance evaluation," *J. VLSI Sig. Proc.*, vol. 27, pp. 297–312, 2001.

[6] 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.

[7] 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.

[8] R. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, March-April 2000.

[9] S. Unger and F. Mueller, "Handling irreducible loops: Optimized node splitting vs. DJ-graphs," *ACM Trans. Prog. Lang. Sys.*, vol. 24, no. 4, pp. 299–333, 2002.

[10] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison-Wesley, 1986.

[11] A. Brown, S. Z. Guyer, and D. A. Jiménez, "The C-Breeze compiler infrastructure," Department of Computer Sciences, The University of Texas, Austin, TX," Technical report, July 12 2004.

[12] GraphViz. [Online]. Available: http://www.research.att.com/sw/tools/graphviz/

[13] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC architecture description language and tools," *Int. J. Parallel Prog.*, vol. 33, no. 5, pp. 453–484, October 2005.


```
LoopParams() // accessing current loop parameters
Data: loop_a address.
Result: initial, step, final values for the current loop.
begin
  initial := lparams_m[loop_a].INITIAL
  step := lparams_m[loop_a].STEP
  final := lparams_m[loop_a].FINAL
end

IndexCalculation() // current loop index update
Data: loop parameters, muxsel flag, ttsel, taskend.
Result: IXRB, muxsel updated, loopend generated.
begin
  if not(muxsel_m[loop_a]) then
    index_t := initial + step
  else
    index_t := IXRB_m[loop_a] + step

  if index_t greater than final then
    loopend := 1
  else
    loopend := 0

  if not(loopend) and not(ttsel) then
    IXRB_m[loop_a] := index_t
  elsif loopend and not(ttsel) then
    IXRB_m[loop_a] := initial

  if taskend and not(ttsel) and not(loopend) and
  not(muxsel_m[loop_a]) then
    muxsel_m[loop_a] := 1
  elsif taskend and not(ttsel) and loopend then
    muxsel_m[loop_a] := 0
end

TaskSelection() // task switching mechanism
Data: ttsel, task_d, loopend.
Result: ttsel, task_d, loop_a updated,
PC_zolc, PC_task_ext generated.
begin
  if not(ttsel) then
    task_a := task_d concat loopend
  else
    switch on DFRB[task_d]
      when 0: task_a := task_d concat loopend
      when 1: task_a := task_d concat 0
      when 2: task_a := task_d concat 1

  for i in 0..NUM_ENTRIES-1 do
    PCent_a[i] := task_a
  endfor
  for i in 0..NUM_EXITS-1 do
    PCext_a[i] := task_a
  endfor

  if taskend then
    task_d := ttlut_m[task_a].TASK_DATA
    ttsel := ttlut_m[task_a].TTSEL
    loop_a := ttlut_m[task_a].LOOP_A

  PC_zolc := PCent_m[task_d].zolc_trg
  PC_task_ext := PCext_m[task_d].zolc_trg
end
```

Fig. 4. Pseudocode for ZOLC microarchitectural-level operations.