Elimination of overhead operations in complex loop structures for embedded microprocessors

Nikolaos Kavvadias, and Spiridon Nikolaidis, *Member, IEEE*

Abstract—Looping operations impose a significant bottleneck to achieving better computational efficiency for embedded applications. In this paper, a novel zero-overhead loop controller (ZOLC) supporting arbitrary loop structures with multiple-entry and multiple-exit nodes is described and utilized to enhance embedded RISC processors. 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 for enabling its utilization. This description is designed to be easily retargetable to single-issue RISC processors, requiring only minimal effort for this task.
The ZOLC unit has been incorporated to different RISC processor models and research ASIPs at different abstraction levels (RTL VHDL and ArchC) to provide effective means for low-overhead looping without negative impact to the processor cycle time. Average performance improvements of 25.5% and 44% are feasible for a set of kernel benchmarks on an embedded RISC and an application-specific processor, respectively. A corresponding 10% speedup is achieved on the same RISC for a subset of MiBench applications, not necessarily featuring the examined performance-critical kernels.

Index Terms—Microprocessors, Control design, Pipeline processors, Optimization, Real-time and embedded systems, Hardware description languages.

1 Introduction

Recent embedded microprocessors are required to execute data-intensive workloads like video encoding/decoding with a favorable power/performance tradeoff. Last years, the respective market is dominated by new 32-bit RISC architectures (ARM [1], MIPS32 [2]), and embedded DSPs as the Motorola 56300 [3], ST120 [4], and TMS320C54x featuring architectural and power consumption characteristics suitable to portable platforms (mobile phones, palmtop computers, etc). Continuously evolving embedded RISC families provide among others deeper pipelines, compressed instruction sets and support for subword parallelism to increase performance. At the DSP end, architectures involve even more specialized characteristics suitable to data-dominated domains such as media processing and OFDM radio, 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.

- *N. Kavvadias is with the Section of Electronics and Computers, Department of Physics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece. (e-mail: nkavv@physics.auth.gr).*
- *S. Nikolaidis is with the Section of Electronics and Computers, Department of Physics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece. (e-mail: snikolaid@physics.auth.gr).*

This research project is co-financed by E.U.-European Social Fund (75%) and the Greek Ministry of Development-GSRT (25%).
Manuscript received 24 Nov. 2005; revised 28 June. 2006, 31 May 2007; accepted 25 June 2007.

However, the proposed solutions are focused on canonical loops met in typical DSP kernels. A disadvantage of most DSPs in the market is that they are only capable of handling perfect fully-nested structures consisting of single-entry static loops [5]. Configurable processors as ARCTangent-A4 [6] and Xtensa-LX [7] are even more restrictive since they incorporate zero-overhead mechanisms for single e.g. innermost loops only. In these cases, a specific processor template is assumed to which alternate control-flow mechanisms cannot be added. Some recent efforts regard the hardware design of specialized units for eliminating loop overheads, as the single-cycle multiple-index update unit [8]. A relevant software solution, loop unrolling, is a compiler optimization technique that removes the overhead of short software loops with a small number of iterations [5], however not always applicable due to conditional control flow at certain nesting levels in the loop body and loop bounds unknown at compile time.

In this work, an architectural solution for zero-overhead loop control (ZOLC) is presented. This solution faces the problem of loop overhead instructions in its general form achieving the optimum performance gains. The proposed ZOLC unit is used at the instruction fetch stage to eliminate the loop overheads and can be applied to arbitrarily complex loop structures (which can comprise of combinations of natural and irreducible loops with multiple-entry and multiple-exit points). A portable specification of a ZOLC unit is also provided, which can be utilized for the automatic generation of the aforementioned enhancements to different embedded processors. In addition, a region-based task control-flow formalism for software synthesis of ZOLC initialization

and configuration is given, and a practical compilation flow for ZOLC code generation has been implemented based on open-source tools. For proof-of-concept, the unit has been incorporated to various RISC processors: XiRisc [9], a 32-bit configurable microprocessor, distributed as VHDL softcore, as well as to ADL (architecture description language) models of other processors (MIPS R3000 for example).

The remainder of this paper is organized as follows. Section 2 overviews previous research regarding the elimination of looping operations in embedded applications. In Section 3, the notion of a data processing task control flow graph (TCFG) for representing loop-intensive applications is explained. A set of motivating examples is unfolded in Section 4, each of them featuring a realistic case that the ZOLC approach would prove beneficial. The architectural template, the implementation details and a portable model of the ZOLC unit are presented in Section 5. Software and hardware integration issues for targeting ZOLC-aware processors are the subject of Section 6. In Section 7, performance results on different RISC processor configurations are discussed with focus on the case of XiRisc. Finally, Section 8 summarizes the paper.

2 Related work

In recent literature, looping cycle overheads are confronted by using branch-increment/ decrement instructions, zero-overhead loops or customized units for more complex loop nests [3], [4], [8], [10], [11]. These approaches are encountered in both academic [9], [12], and commercial processors and DSPs [3], [4], [6], [7]. For the XiRisc processor [9], branch-decrement instructions can be configured prior synthesis. Kuulusa et al [12] presented a DSP core supporting a configurable number of hardware looping units, which can handle the case of perfect loop nests with fixed iteration counts. The DSP56300 [3] supports seven levels of nesting using a system stack. There is a 5-cycle overhead for preparing a loop for this type of hardware control, which may be important for small number of iterations or for linear loops placed at a deeper nest level. In our work such overheads are eliminated since ZOLC initialization occurs outside of loop nests. Lee et al [11] provide software tool support to zero-overhead looping on the TMS320C54x DSP in the form of a loop-specific compiler pass. The corresponding pass is introduced to the *gcc* frontend prior to the actual intermediate representation (IR) generation. Notably, extensive transformations are required to derive the canonical form for nests consisting of single-entry loops [5].

For the majority of DSP architectures, 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 and their conversion to reducible regions can be costly in terms of code size. It has been proved recently [13] that no node splitting technique can avoid the exponential blowup in code size for converting an irreducible CFG to a reducible one. Also, the latter techniques [14], [15] 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.

Specific compiler optimizations for zero-overhead looping on DSPs are extensively discussed in [16]. In the referred paper, the authors regard using a zero overhead loop buffer (ZOLB, a kind of compiler-managed cache) for keeping frequently executed program loops without having to execute loop overhead instructions. In terms of cycle performance, their aim is comparable to ours. Also, they acknowledge as well, the importance of having the compiler automatically exploit potential cases for removing loop overheads. However, our technique presents the following differences to the work by Uh et al.: a) ZOLC can be applied to arbitrarily complex loop structures while the ZOL buffer regards each loop separately, b) the ZOLC approach can be applied independent to the actual local storage of program instructions (on-chip block RAM instruction memory, cache, or loop buffer) while their technique requires that loop bodies are copied to the ZOL buffer.

A typical case of RISC processor adaptation for fully nestable types of hardware-implemented loops with fixed number of iterations has been described in [10]. A more aggressive dedicated controller for perfect loop nests is found in [8]. 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 existing RISC processors. With our method, complex loop structures are supported, without regard for the compiler optimization capabilities. The presented method can be applied to structures with dynamic loops that have bound values only resolved at run-time. This particular case cannot be serviced by any of the previously discussed methods.

3 Representation of loop-intensive applications

The control-flow information of each function in a given program is captured within its control-flow graph (CFG) [17], which is a directed graph having vertices representing the function's basic blocks and edges denoting the direction of control-flow. For the task of graph-based

code generation, the control-data flow graph (CDFG) of the application will be processed, which consists of the CFG with its nodes expanded to their constituent instructions. In order to let the ZOLC engine execute looping operations at the background, it is required for the compiler to perform the following tasks: a) remove the loop overhead instructions from the original CDFG, b) generate instructions for initializing the ZOLC storage resources and c) 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 maximum 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 portions of the code situated at loop boundaries. The instruction lists that comprise each of these code segments are the *Data-Processing Tasks* ($DPTs$) of the algorithm. The CTEs model abstract control operations. In our case, CTEs either correspond to hardware signals of ZOLC or to conditions issued by the processor instructions as those by dynamic branches. 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 in Section 4.3.

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 [18]. The algorithm consists of six nested loops; the loop-nesting diagram for *fsme* is shown in Fig. 1(a), with the corresponding TCFG in Fig. 1(c) (graph layout obtained by [19]). Primitive backward tasks are denoted as bwd_i, where i is the enumeration of the loop nest, starting from one since 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 $bwd_m - bwd_n$ (as in Fig. 4) 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 $loopend$ 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.

To elaborate, backward tasks are noted with a star in Fig. 1(a)-Fig. 1(b) and can be placed at the innermost or closing position of a loop, with the final task, $bwd0$, marking the exiting position of the loop structure. The loop indices are updated during the execution of these tasks. Forward tasks are quoted with a circle and are placed in non-terminating positions of a loop. Such tasks can participate in control flow decisions and have no effect on the loop indices.

The TCFG representation of loop structures has been used for the following: a) to generate the control part of ZOLC for non-programmable processors [20] and the ZOLC initialization sequence for programmable processors, b) to apply formal graph transformations as those induced by loop transformations and c) to enable a design space exploration procedure for hardware-software partitioning with hardware DPT accelerators.

4 USE-CASES OF A ZOLC ENGINE

4.1 Case study: Blocked matrix multiplication on a ZOLC-enhanced DSP

In order to provide a typical usage example of ZOLC-controlled operation, we assume a hypothetical (with a working cycle-accurate model) DSP engine, hereafter termed hDSP. The hDSP is a signal processing ASIP (application-specific instruction-set processor) with heterogeneous storage (dual-memory banks and dedicated registers), single-cycle multiply-accumulation, advanced address generation capabilities, and a 5-stage pipeline inspired by the classical DLX design.

A plethora of multi-dimensional signal processing applications such as color space conversion, frequency-time domain transforms, and image filtering, involve processing on matrix-matrix multiplication primitives, which are implemented as triple-nested loops contained within outer loops for block scanning. Such performance-critical kernel is most probably written in native assembly which is deduced from the reference code of Fig. 2(a). Fig. 2(b) shows the hand-optimized code segment of the kernel where *AGU* denotes integer operations on the address-generation unit, '@' the current address for accessing an array variable in a memory bank (*X* or *Y*), '||' is used for concurrent micro-operations, *RI, RF, RX, RP* are the initial and final loop parameter, current index and generic parameter register classes, and *ACC* is the accumulator.

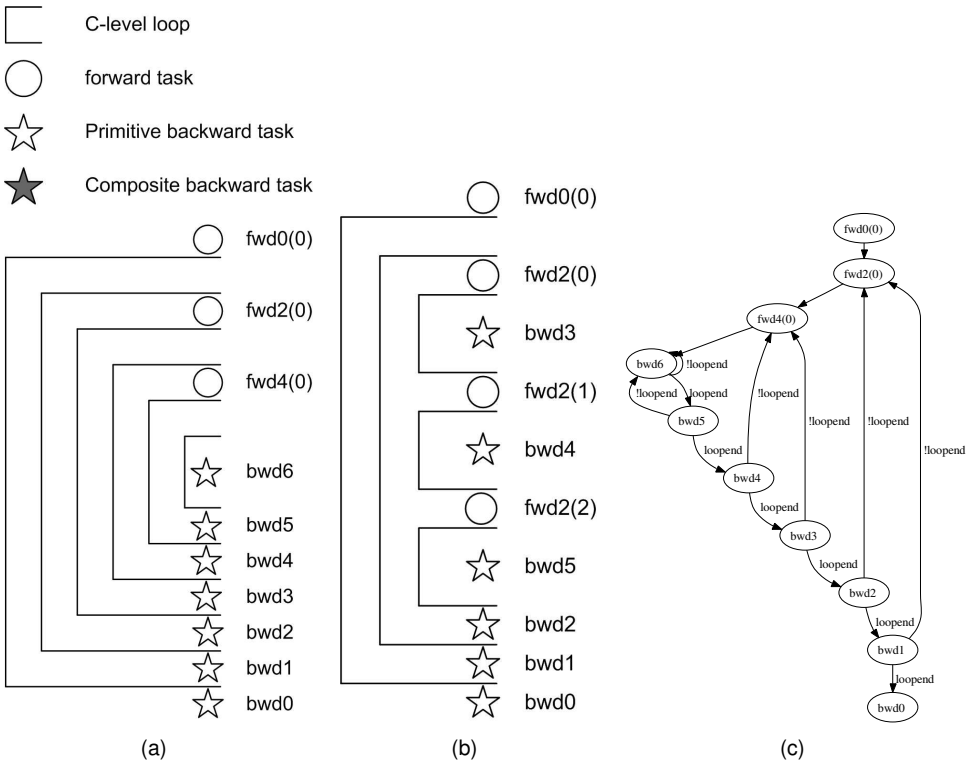


Fig. 1: Loop structures and TCFG views for loop-intensive algorithmic kernels. (a) Loop nesting diagram for a fully-nested structure (*fsme*). (b) A complex loop structure found in a biorthogonal wavelet filter. (c) TCFG for the *fsme*.

For branch operations of 3-cycle penalty, the overall cycle requirement for the inner loop of Fig. 2(b) is 10 cycles. The benefit from using ZOLC is that the increment and compare-and-branch operations (aggregating 4 cycles) can be removed and replaced by a task switching operation associated to the instruction fetch of the last useful instruction in the inner loop (*MAC ACC, @COEFF, @IMAGE*). When using ZOLC, the inner loop executes in 6 cycles, which scores an approximate 40% boost in cycle performance. In general, information not available at compile-time has to be updated in the ZOLC storage resources at run-time, which reveals the need for instruction extensions to the processors for this purpose. Such loops are termed semi-static and cannot be unrolled at compile-time. They are common in some applications e.g. the biorthogonal (2,2) Cohen-Daubechies-Feauveau wavelet filter (Fig. 1(b)) used in Wavelet-Scalar Quantization (WSQ) image compression [21].

For ZOL-controlled execution, prior entering the loop nest, the task-related information is stored in a set of local register files. The data attributed to each task consist of the following elements, also shown in the form of the *TaskContext* 'record' in Fig. 2(c):

- the PC entry address (absolute or relative, depending on the ZOLC implementation). For multiple-entry tasks there exist more than one such addresses
- the PC exit address, i.e. the address of the last useful computation instruction in the task. Similar to above, multiple-exit tasks have more than one possible PC exit addresses monitored
- the encodings for the possible subsequent DPTs; the decision is taken at run-time by inspecting the loop end condition (for *bwd* tasks) or possibly a dynamic forward branch condition (for *fwd* tasks)
- the loop address to which the subsequent possible tasks are contained
- the task type (*fwd/bwd*) for these tasks

Fig. 2(d) illustrates the context values for the most important DPTs in the *matmult* algorithm: $fwd_4(0)$, bwd_5 and bwd_4.

ZOLC initialization for preparing the processor to enter a state where all looping is controlled by ZOLC can be done by an instruction sequence or configuration stream produced by a separate compilation pass operating on the TCFG representation (Section 6.1). A few instruction extensions are also required which are discussed in Section 6.2. For the entire 5-loop *matmult* algorithm, the required additional cycles are:

- 16 instructions for storing the entry and exit PC absolute (or offset) address for marking the boundaries of a task
- 14 instructions for initializing the task switching information when residing in a specified task context. In this case, a task context is represented by the concatenation of the task enumeration and the

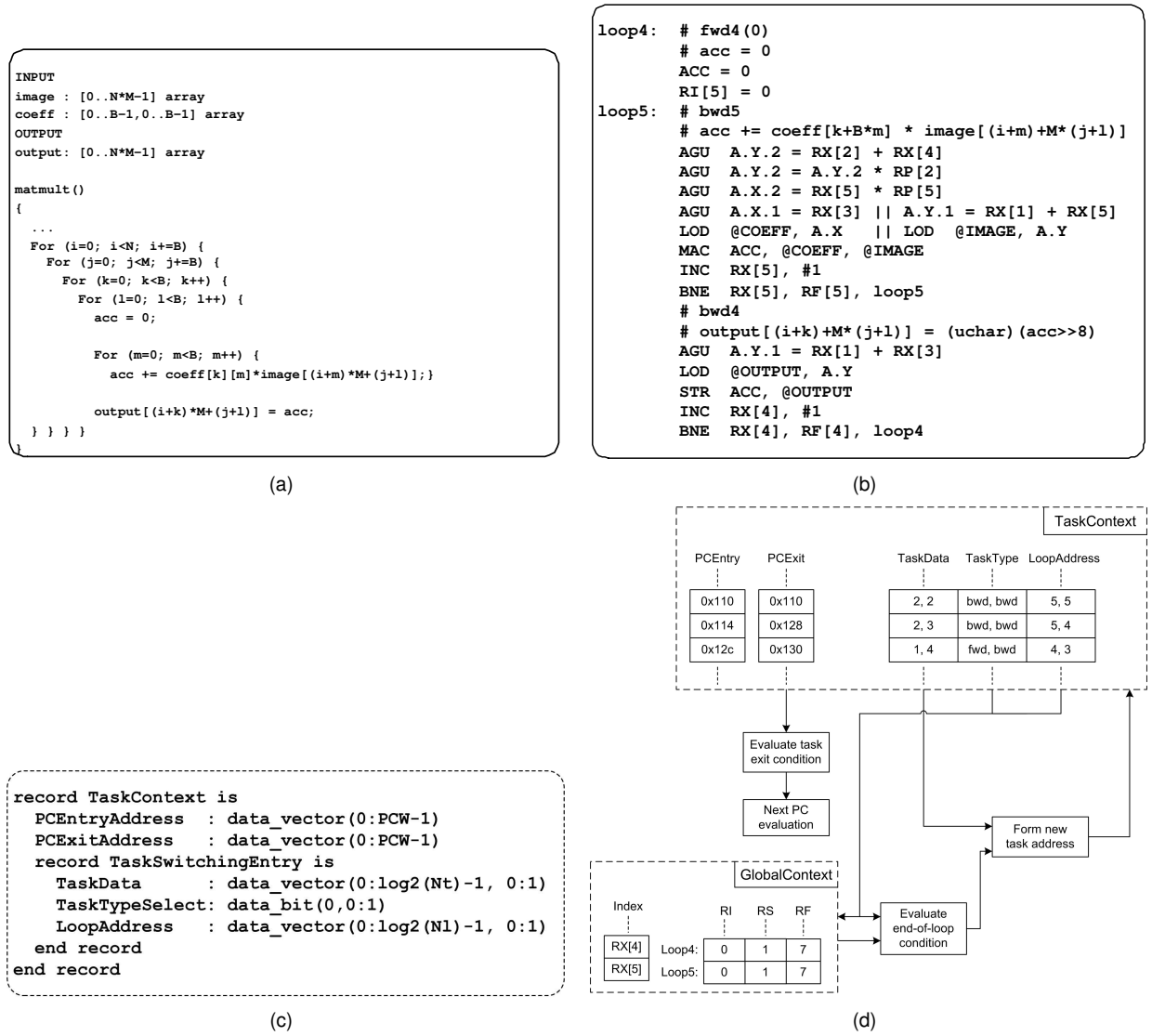


Fig. 2: ZOLC-aware application mapping for the blocked matrix multiplication algorithm (*matmult*). (a) ANSI C reference code. (b) Hand-optimized hDSP assembly. (c) The $TaskContext$ record encapsulating the intrinsic information attached to a data processing task. (d) $TaskContext$ and $GlobalContext$ values for the performance-critical tasks of the algorithm.

value of the end loop condition for the current loop (possibly with multiple-exits). The task switching information consists of the task enumeration, the task type (*fwd* or *bwd*) and the loop address for a possible subsequent task.

- 15 instructions for setting the initial (e.g. lower bound), step (stride) and final (e.g. upper bound) value for a loop.
- an additional instruction is needed for configuring a mask register used among others for configuring the induction expressions associated to each *bwd* task in the index calculation ALU. By default, addition and subtraction operations are assumed.

The overall number of cycles (46) is negligible when compared to the over 5.7 million required for *matmult* on CIF (352×288) images (for the Y-component of the image) with an 8×8 primitive. For a 20-loop algorithm (*fsme_dr* mentioned in Table 4), the overhead cycles are limited to 164, which are over three orders of magnitude less than any other method can sustain including [8]. Also, the ZOLC approach has more positive side-effects: stricter real-time constraints may be applied on the application since the best against worst case execution time difference (*BCET-WCET*) is reduced. In addition to these reasons, the usage of ZOL control is even more important for data-parallel architectures with short inner loops. For the MorphoSys architecture featuring a 2D-array reconfigurable datapath, the authors report speedups of 4 due to zero-overhead looping for frequent tasks containing block data movement operations [22].

4.2 Supporting multiple-entry loops for ZOLC operation

Multiple-entry loops are irreducible regions of the CFG, undetectable by the classic natural loop analysis algorithm [17] used by many retargetable optimizing compilers [23], [24], [25]. The consequence is that familiar code optimization techniques are not applicable on these regions. The main approach to overcome this is to use traditional node splitting [17] that theoretically increases code size exponentially and is avoided in the majority of research compilers. Optimized versions of node splitting have been suggested recently [15] that employ an elaborate graph type for attributing backedges in the CFG to their unique classes. A drawback is that a significant paradigm shift is imposed since extensive modifications are required to current compilers to take account their proposed representation.

In order to explore multiple-entry loops for potential of ZOLC operation, the TCFG representation of Fig. 3(b) suffices to accurately model the loop structure for the code in Fig. 3(a). Then, the proper instructions will be removed from all bwd tasks and from fwd tasks that are static loop initialization pattern containers. After that, simply by edge enumeration, the necessary task sequencing information for initializing the ZOLC is derived.

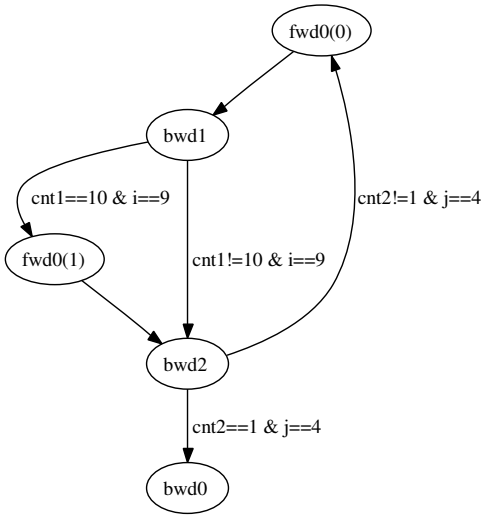
The TCFG sustains a uniform representation for complex loop structures. Compared to the best methods for handling irreducible loops, not only code bloat is avoided, but code size and dynamic instructions are reduced.

4.3 Supporting alternative models of hardware looping

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 [8], [26]. For the examined application (*fsme*), four consecutive backward tasks are merged to the 'bwd1-bwd4' composite node as shown in Fig. 4. Thus, a number of loop end signals equal to the number of loops in the kernel are required. In Fig. 5, 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 by merging consecutive primitive backward tasks.

```
L1: cnt1++;

for (i=1; i<10; i++) {
  a[i] = a[i-1] + 1;

  if (cnt1==10 & i==9)
    break;
  else if (cnt1!=10 & i==9)
    goto L2; }

cnt2++;

L2: for (j=0; j<5; j++) {
  b[j] = 2*a[j]-1;

  if (cnt2==1 & j==4)
    break;
  else if (cnt2!=1 & j==4)
    goto L1; }
```

(a)



(b)

Fig. 3: Illustrative C source code and the corresponding TCFG for a case of multiple-entry loops. (a) C source code for the example. (b) The corresponding TCFG.

5 Incorporating ZOLC to a programmable processor

5.1 Template view of the ZOLC-enhanced processor

A block diagram indicating how the proposed ZOLC architecture is incorporated in the control path of a typical RISC processor is shown in Fig. 6. 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 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 by processor instructions that employ a special paged-access format added to the original instruction set architecture. 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

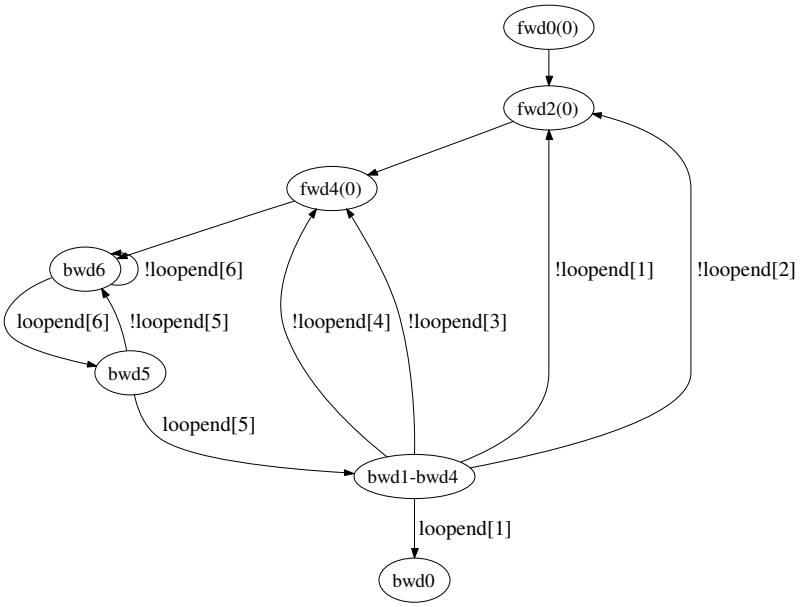


Fig. 4: TCFG for an *fsme* implementation using the index update hardware of [8].

for the case of multiple-exit loops to PC 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. The initial, final and step loop parameters are used to calculate the current index value and determine if a loop has terminated.

5.2 Architectural details for ZOLC components

As can be seen from Fig. 6, the ZOLC accepts a control vector from the modified instruction decoder, which consists of write enable signals for its register files, page selection and register address for selecting a specific register. This is the required control information that the task selection unit, the loop parameter tables, and the modified register file accept from the decoded instructions. For the example case of XiRisc, the instruction decode vector is extended to produce $k_{ent} + k_{ext} + 7$ additional bits dedicated to handle the ZOLC resources, where k_{ent} and k_{ext} are the maximum permissible number of entries/exits for each loop.

Regarding the index registers on a processor utilizing a ZOLC unit, there exist two possibilities: the indices can be allocated as a separate register class on the actual integer register file or an explicit index register bank should be introduced in the architecture. The former case is preferred since additional instruction extensions to the opcodes needed for ZOLC initialization, would not be required. In practice, we have used 16-register index class on the 64-entry general-purpose register file of the hDSP, and correspondingly 8 indices on the original XiRisc (32 entries) without negative impact on performance for the examined benchmarks as it is also endorsed by liveness analysis results on Machine-SUIF. If the index register bank approach is followed, 3 additional bits are required for the instruction encodings, which may not available. The loop parameter tables provide respective storage for the loop bound (initial, final) and stride (step) values. Compile-time unknown bound or stride values are handled with dynamic updates. Thus, a shadow path is designed for transferring the content of general-purpose registers to the corresponding paged registers. Simple move instructions are used to update the loop parameters at normal program flow.

The task selection unit incorporates a partitioned LUT for selecting the following task, loop address and task type. The number of entries to this LUT is $2 \times (n_t - 1)$ where n_t is the maximum number of tasks in the loop structure. In the most extreme case, $n_t = 2 \times n_l + 1$ where n_l is the maximum number of loops supported on a specific hardware manifestation of ZOLC processing. The data word length for these LUTs is $log_2(n_l) + 1$, $log_2(n_l)$ and 1, correspondingly, since it is pointless to specify task transitions with $bwd0$ as the source. Thus, the number of tasks able to drive task switching is given by $n'_t = 2 \times n_l$. For 16-bit unsigned immediates, an entry for each DPT of a 128-loop algorithm could be assigned with a single instruction.

Also, following a simple path to implementation, k_{ent} dedicated register files for determining the PC target for the following task and the candidate k_{ext} PC values for the multiple-exit condition have to be placed in the same unit. If no support for multiple-exit loops is required, then $k_{ent} = k_{ext} = 1$. The overhead of multiple PC storage being unused can be confronted with a more

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

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

Fig. 5: Index update in primitive and composite backward tasks, respectively.

challenging realization utilizing key entries for the PC entry/exit addresses of a task. Each task is assigned a double-key (key_{ent}, key_{ext}) notifying the number of entries and exits for the specified task. The actual PC addresses are then accessed in incremental addresses from the key. The PC target is determined among k_{ext} possible choices based on the value of the task entry PC selection vector, shown in Fig. 6. The simple equality comparators used for evaluating the outcome of the multiple-exit condition, reside to the PC decode unit.

The index calculation unit consists of a 2-state FSM to acknowledge the status of the running loop (first or subsequent iteration) and the combinational index update module. In general, this unit is an additional ALU, but for the examined applications, an adder-subtractor suffices since all index updates were simple additions to a constant or variable. Index update is performed only on the completion of a *bwd* task. Also, the integer register file is modified to support one read and one write port that are dedicated to loop index transfers.

The storage bit count for the ZOLC configurations can be easily derived from the summary of the modules introduced due to ZOLC in the XiRisc case study shown in Table 1. In the following expression, the storage bit count is calculated by summing up the contribution of each module given as the product of the number of instances (Quantity) by the number of entries (Num. entries) and bits per entry:

$$\sum_{Units} (\text{Quantity}) \times (\text{Num. entries}) \times (\text{Bits per entry}) = 2 \cdot (4 \cdot n_l + 1) \cdot (log_2(n_l) + 1) + 2 \cdot (k_{ent} + k_{ext}) \cdot n_l \cdot PCW + (3 \cdot DW + 1) \cdot n_l$$

where:

- n_l, k_{ent}, k_{ext} have already been defined in the text
- DW is the data word width of the processor
- PCW denotes the program counter width

5.3 RT-level specification of the ZOLC mechanism

In order to assist the application of ZOLC mechanisms we have developed pseudocode semantics for their representation shown in Fig. 7. A formal description of ZOLC micro-architectural level operations can be exploited in the scope of high-level synthesis for DSP-oriented ASIPs. Table 2 summarizes the notation used for signals and storage resources of the 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, $NUM\_ENTRIES = k_{ent}$ and $NUM\_EXITS = k_{ext}$ dedicated register files are required for determining the PC target for the following task and the candidate PC exit values if multiple-exit conditions need be supported, respectively.

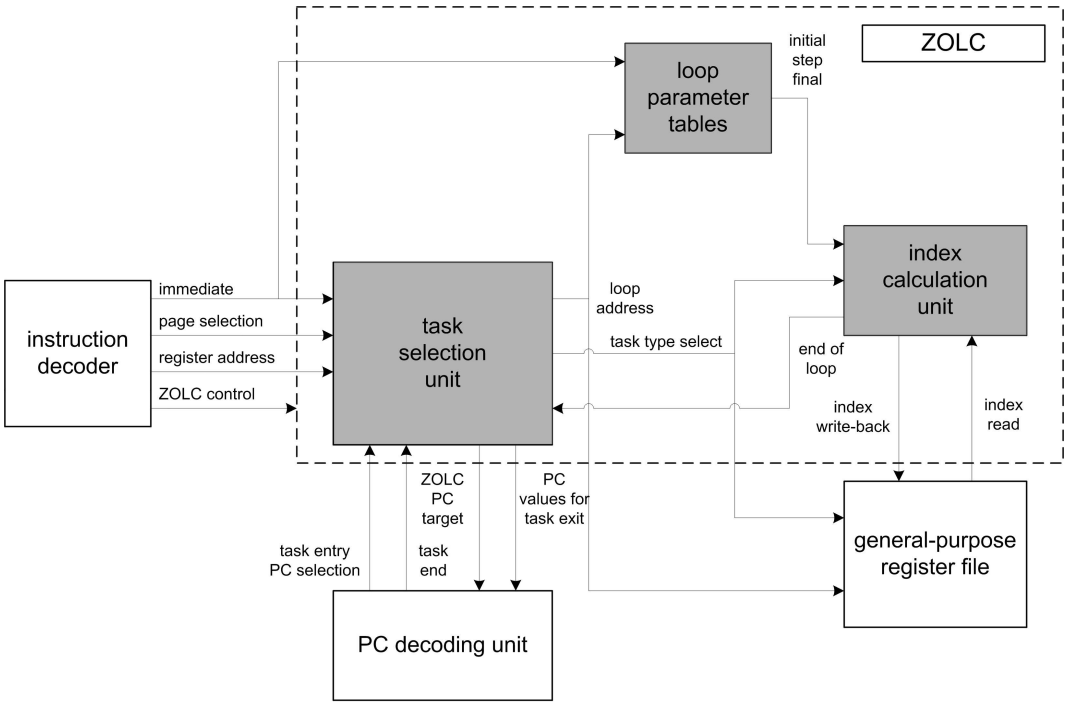


Fig. 6: Incorporating the ZOLC architecture to programmable RISC processors.

TABLE 1: Summary of additional storage resources for XiRisc introduced by ZOLC.

Unit	Component	Quantity	Num. entries	Bits per entry
Task selection unit	LUT (*task_d*)	1	$2 \cdot n'_t$	$log_2(n'_t)$
	LUT (*ttsel*)	1	$2 \cdot n'_t$	1
	LUT (*loop_a*)	1	$2 \cdot n'_t$	$log_2(n_l)$
	Output register	1	1	$log_2(n'_t) + log_2(n_l) + 1$
	PCent_m register file	k_{ent}	n'_t	PCW
	PCext_m register file	k_{ext}	n'_t	PCW
Loop parameter tables	Register files	3	n_l	DW
Index calculation unit	Mask register	1	1	n_l

During PC decoding with the ZOLC in operation, a certain $task\_d$ value has been issued and the PC value is compared to the $PCext\_m$ entries for the same $task\_d$ (for all the potential task exits). The $taskend\_t$ vector encodes the result of this parallel comparison, while $zolc\_trg$ is its binary encoding and $taskend$ is active when there is at least one match of a task exit address. In such case, the ZOLC behavior is accessed to determine the successive PC value and can be decomposed into the $LoopParams$, $IndexCalculation$, and $TaskSelection$ procedures. As the result of these evaluations occurs during a single clock cycle, the newly evaluated PC_zolc value is assigned to the PC and is available at the succeeding cycle.

The task of the $LoopParams$ process is to access the current loop parameters (initial, step, final value) from the corresponding entry of the loop parameters register banks addressed by $loop\_a$. $IndexCalculation$ is used for evaluating the current loop index value. For this purpose, the $muxsel\_m$ status register denotes for which loop its loop index must be initialized during the next index update. The $loopend$ signal, also calculated by the index calculation unit, signifies the final iteration of a loop. $TaskSelection$ describes the actual task switching mechanism. For backward tasks and trivial forward tasks (not exited by an explicit conditional branch), the task address ($task\_a$) for the subsequent task is formed by concatenating the $task\_d$ and $loopend$ signals. Forward tasks exited by dynamic branches require additional information for the branch target task encoded in the $DFRB$ status register. On the presence of an active $taskend$, the information for the following task is read from the task selection LUT. The $PCent\_m$ and $PCext\_m$ register files are addressed by $task\_d$. The specific entries for the $zolc\_trg$ 'column' address from the $PCent\_m$ and $PCext\_m$ provide the appropriate PC_zolc and the task exit PC for the following task, respectively. This approach demands that the task selection LUT can be read asynchronously.

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

continued on next column

```
continued from previous column
      taskend_t[i] := 0
    endif
  endfor

  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
  endif
end

// accessing current loop parameters
LoopParams()
begin
  initial := lparams_m[loop_a].INITIAL
  step := lparams_m[loop_a].STEP
  final := lparams_m[loop_a].FINAL
end

// current loop index update
IndexCalculation()
begin
  if not(muxsel_m[loop_a]) then
    index_t := initial + step
  else
    index_t := IXRB_m[loop_a] + step
  endif

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

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

  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
  endif
end

// task switching mechanism
TaskSelection()
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
    endswitch
  endif

  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
  endif

                                  continued on next column
```

```
continued from previous column
  PC_zolc := PCent_m[task_d].zolc_trg
  PC_task_ext := PCext_m[task_d].zolc_trg
end
```

Fig. 7: Pseudocode for ZOLC microarchitectural-level operations.

TABLE 2: Notation used in the pseudocode of Fig. 7.

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

6 Software and Hardware Considerations for ZOLC Support

6.1 Automating ZOLC optimizations within a practical compiler-assembly optimizer framework

To support the potential use of the ZOLC unit, proper development tools are required in order to ease code generation. For the best possible performance, an additional code generation pass should be executed for exploiting ZOLC, prior to register allocation and just after constructing the optimized CFG. Its purpose is to emit ZOLC initialization code and to insert move operations that are required for loops with dynamic bounds as in the case of loop tiling [5].

Regarding XiRisc, the available software toolchain is based on the *gcc* compiler. The effort required to devise and import a new internal compilation pass to *gcc* can be devastating, since it was initially designed for adding new machine targets but not new code transformation passes. A workaround is to post-process the resulting assembly code and this approach was taken for obtaining the performance measurements of Section 7.2.

As a proof-of-concept of supporting ZOLC in an automated compiler framework, we have developed an analysis and markup pass in Machine-SUIF [27] working at the CFG level which is used to pass macro-instructions and additional information to a SALTO [28] transformation pass for finalizing the ZOLC initialization sequence. The targeted architecture is named 'SUIF real machine' or *SUIFrm* for which we have a working experimental backend written for Machine-SUIF and a set of ArchC [29] instruction- and cycle-accurate simulators. *SUIFrm* has been designed as a 'real' ISA, closely matching the *SUIFvm* IR (which is the actual Machine-SUIF IR). The main additions for the *SUIFrm* backend were:

- Homogeneous register architecture (12, 32, 64 register versions)
- Proper interpretation of the general convert (*cvt*) operator to forms of zero-extension (*zxt*) and sign-extension (*sxt*)

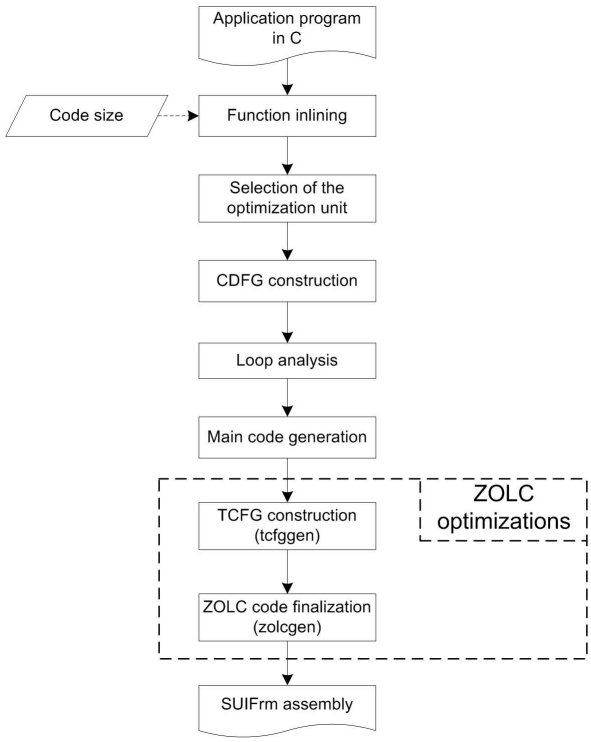


Fig. 8: Part of a compilation flow supporting the automation of ZOLC code generation.

– Procedure argument passing via specialized argument registers

There are still certain features missing from the *SUIFrm* target affecting its capability of servicing general-purpose applications: a) lack of hardware floating-point support, b) C structs, unions and recursive procedures are not supported. However, *SUIFrm* has been successfully used on the kernel benchmarks of Subsection 7.2. Also, a set of *SUIFrm* backends (with/without ZOLC support) has been developed for SALTO.

Fig. 8 illustrates the code generation procedure integrated within our compilation flow, which comprises of SUIF frontend and Machine-SUIF backend functionality, complemented with passes for ZOLC optimization written according to the Machine-SUIF or SALTO application programming interface. Its operation can be subdivided into the following stages:

– Function inlining

In this stage, selective function inlining is applied to expose as many basic blocks as possible in the top-level function of the application program for a given code size constraint.

– Selection of the optimization unit

As a consequence of operating on CFGs and not on whole-program IR units (e.g. program dependence graphs), it is mandatory to select a single procedure for applying the ZOLC optimizations. The optimization unit can be selected on the basis of execution frequency profile of the application program under consideration.

– CDFG construction

The collection of connected CDFGs for the entire algorithm is constructed. It is assumed that a form of operation scheduling has been performed at least locally (on the basic block DAGs).

– Loop analysis

This stage has the purpose of identifying the looping statements in the algorithm, either by natural loop analysis [17] which is the currently supported method in Machine-SUIF or preferably by more sophisticated multiple-entry loop detection methods. The natural loop analysis report contains the loop nesting depth and three additional boolean flags for determining: a) if a loop begins at the specified node (*begin_node*), b) if a loop ends at the specified node (*end_node*), c) if an exit from the loop is possible from that node (*exit_node*). General loop analysis identifying loops with multiple entries and multiple exits to non-sequential basic blocks, would additionally require knowing to which basic block, control is transferred from its predecessor loop exit basic block. Only a few experimental compiler infrastructures seem to support direct multiple-entry loop detection [30].

– Main code generation

In this stage, the instruction selector ('do_gen' pass) and register allocator ('do_raga') are invoked. The instruction selector of Machine-SUIF undertakes the task of generating SUIFrm code from the SUIFvm IR. The core of 'do_gen' implements a simple pattern-matching translator from SUIFvm to SUIFrm and it only allows room for peephole-like optimizations in this process. The register allocator is more sophisticated and implements the well-known method of iterated register coalescing [31]. At the end of the stage, valid SUIFrm assembly can be emitted.

– TCFG construction (*tcfggen* pass)

Based on the loop analysis results, the control flow of the algorithm is mapped to its TCFG. Further, this stage passes the static information for the loops in the algorithm to the subsequent stage in the form of four different types of pseudo-instructions. An LDST instruction incorporates all the information needed for creating a task selection LUT entry: for a given task encoding (*current-task_d*), the succeeding task (*next-task_d*), its type (*next-ttsel*) and loop address (*next-loop_a*) are given. DPTI instructions denote the first and last basic block of a task, while a LOOP provides a given loop address, the actual loop index register and the loop parameters. An OVERHEAD marks its following non-pseudo instruction whether it must be kept (default), replaced by a no-operation, or entirely removed.

– ZOLC code finalization (*zolcgen* pass)

The *zolcgen* SALTO pass produces the actual ZOLC initialization code that has to be inserted in a preceding basic block to the loop nest to update the ZOLC storage resources and is typically the first basic block of the targeted procedure. This process also involves the following tasks:

1) conversion of LOOP pseudos to an LDSA-LDSI-LDSS-LDSF sequence of instructions (and their repositioning). The LDS[I|S|F] instructions are explained in Table 3 while an LDSA (Set index register alias) associates the general-purpose register used for indexing of a certain loop with its *loop_a*. LDSA instructions can be used for static renaming in case of an architecture with dedicated loop index registers. For a homogeneous register architecture this information would be mapped on a small LUT providing the alias relation of registers used for indexing with their correspondent loop addresses.
2) the overhead instructions are properly handled and the respective overhead markers are removed
3) the task entry and exit PC addresses are calculated and the corresponding LDS[N|X] instructions (Set a PC entry/exit address) are created.

TABLE 3: Instruction-set extensions for ZOLC support on the XiRisc processor.

Mnemonic	Description
LDSL	Set task transition LUT entry
LDSNi	Set PC entry address for $k_{ent} = i$
LDSXi	Set PC exit address for $k_{ext} = i$
LDSM	Set mask register for $muxsel\_m$ and index calculation ALU initialization
LDS[I|S|F]	Set initial/step/final loop parameter
MOVG2[I|S|F]	Move a general-purpose register to an initial/step/final loop parameter register
MOV[I|S|F]2G	Move an initial/step/final loop parameter register to a general-purpose register

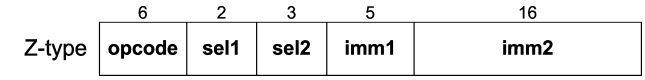


Fig. 9: Suggested instruction format for ZOLC initialization. It is assumed that $k_{ent}, k_{ext} \leq 4$.

6.2 Elaborating on hardware and software implications of ZOLC on embedded processors

The ZOLC technique is a control path optimization generally applicable to the rather wide context of in-order completion embedded processors. Potential issues regarding the use of ZOLC are highlighted in the following few paragraphs.

– Incorporating ZOLC to VLIW processors

Theoretically, the ZOLC technique could be applied on VLIW processors as well, with much increased overhead in keeping track of redundant state due to the effect of software pipelining inferring a multiple-index context (multiple iterations alive at the same control step). However, as stated in [32], contemporary VLIW DSP processors tend to omit the feature of ZOL. For example, the TMS320C62xx (with 8 issue slots) does not support ZOL instructions; it requires the processor to explicitly perform the looping operations. This is reasonable for the VLIW case since VLIW-based processors are able to execute many instructions in parallel. The operations needed to maintain a loop, can be executed in parallel with other arithmetic computations, assuming that the optimizing compiler for the processor is able to find such schedule. For an optimistic schedule, this would achieve the same effect as if the processor had dedicated looping.

– Integration of TCFG extraction for ZOLC code generation in current compilers

An inherent characteristic of CDFG-like compiler IRs (for example in SSA form) is to represent each procedure by a single CFG which complicates TCFG extraction and therefore directly affects the quality of ZOLC code generation. The extent of applying procedure expansion (inlining) and recursion are the two dominant problems in this approach. On the other hand, the program dependence graph (PDG) [33] representation allows multi-procedure control dependence analysis algorithms that are much more compatible to the requirements for generating a program-level TCFG. A composed control-dependence graph (CCDG) generator written for Machine-SUIF has been reported in [34] and is under consideration in order to be used for further extension of our ZOLC code generation tool.

– Instruction-set extension for ZOLC support

For the software initialization of ZOLC, a small set of instruction extensions are needed for executing specialized load operations on the added hardware. We have implemented this particular approach for incorporating ZOLC in XiRisc [9] where three new opcodes (*LDS, MOVG2X, MOVX2G* instructions) were committed with the corresponding format shown in Fig. 9. The added instuctions are summarized in Table 3.

– Choice of a separate index register bank or dedicating a new register class for indexing

Traditionally, DSP processor architectures have used index register banks enabling more efficient use of complex addressing modes. The easiness of allocation on homogeneous register architectures on the early RISCs prohibited the use of distributing fast storage resources as registers. For these reasons we believe that the index register bank approach can be viable on new ASIP architectures without source or binary compatibility issues to prior ISA generations. Still, using an index register class can be applied in embedded processor families, providing a form of source compatibility is maintained via assembly-level transformations. Since index values usually demand reduced bitwidth, estimated area and power consumption can be significantly reduced. In such scheme, the introduction of smaller bitwidth registers in a homogeneous architecture has been used in asymmetrically-ported register files [35] with significant benefits in power consumption.

– Task switching on multi-cycle or data-dependent timed instructions

The process of selecting the proper subsequent task is performed dynamically during instruction fetches. Parallel comparisons to all PC exit addresses of the current task, reveal if the currently fetched instruction is the last

in the task. Also, the decision process accounts at runtime the loop end condition (for *bwd* tasks) or a dynamic forward branch condition (for *fwd* tasks). These micro-operations suffice only for the simplest cases, that is the last DPT instruction is single-cycle and the processor supports in-order completion. For multi-cycle instructions with known latency, the decision process should be guarded until a proper number of cycles prior the executed instruction is retired. For a 4-cycle multiply operation, the dynamic decision should be enabled 3 cycles after its fetch from program memory. Instructions with latency unknown at compile-time (e.g. the data-dependent multiplication on the ARM7TDMI requires 2 to 5 cycles in the execution stage) decrease ZOLC performance, and if possible should be reordered by the compiler.

7 PERFORMANCE EVALUATION

7.1 Hardware requirements for implementing ZOLC mechanisms

For the purpose of performance evaluation, we have implemented three different manifestations of a ZOLC engine. As *ZOLCfull* we refer to a full-featured version of ZOLC supporting 32 task switching entries (which corresponds e.g. to 16 single-entry DPTs) describing an arbitrary complex 8-loop structure with a maximum of 4 entries/exits per loop. *ZOLClite* differentiates from *ZOLCfull* in the lack of support for multiple-exit loops. In this case, a loop can only be exited from the tail instruction of the stripped loops, being the last useful computation instruction of a loop body. Control-flow is permitted inside loop bodies as long as it does not refer to locations outside the loop. *uZOLC* is a minimalistic version of a ZOLC engine pronounced "micro-ZOLC", usable for single loops. The different ZOLC versions were incorporated in the VHDL description of XiRisc (noted as *XRdefault*). *XRdefault* and the three ZOLC versions of the XiRisc processor were synthesized for an STM 0.13μ standard cell library. Given that the corresponding results have been obtained prior place-and-route, the processor cycle time has not been affected (maximum clock frequency around 170MHz for all versions). This is due to the fact that a path through the 1/1 throughput/latency cycle multiplier was the critical path of the processor for the ZOLC configurations as well. In fact, the estimated maximum clock frequency by the synthesis tool for *ZOLCfull* was even higher than the default version, however this is probably due to the evaluation of wire load delays.

The combinational area overhead for ZOLC was measured from a few hundred gates (298 for *uZOLC*) to a few thousand gates: 4428 for *ZOLCfull* and 4056 for *ZOLClite* given that XiRisc synthesizes to an area of 21309 NAND equivalent gates (combinational logic and storage). Assuming the default XiRisc options (PCW=16, DW=32) the storage demands are easily extracted for the three ZOLC configurations. As can be derived from the analytical expression of Section 5.2, 171, 1552 and 3088 storage bits are needed for the $uZOLC$, $ZOLClite$ and $ZOLCfull$, respectively. Even though not implemented, the storage bit counts can be significantly diminished if size-extension circuitry is used in order to sign-extend 8- and 12-bit stored branch distances in the PC entry and exit tables to the PCW width. If applied, an average of 22.8% is saved regarding storage elements.

7.2 Cycle performance of a MIPS-I processor with ZOLC

Since the XiRisc processor is ISA compatible to MIPS-I, we have made slight modifications on an existing MIPS R3000 model written in ArchC and added ZOLC to it in order to obtain a cycle-accurate XiRisc simulation model.

We have selected two sets of benchmarks for comparing the efficiency of a XiRisc configuration supporting ZOLC which are detailed in Table 4 alongside with the dynamic instruction counts. The first set consists of 10 data-intensive kernels, common to many applications, each consisting of a single C file with reduced data inputs in the source code. Apart from the kernel benchmarks, 10 application instances have been collected from MiBench, for which the "small" data inputs were applied. All programs were compiled with *gcc* and *-O3 -mips1* options forced, with the exception of the *susan* benchmarks, where enabling floating-point emulation (*-msoft-float*) was necessary. Both a *ZOLClite* and an *uZOLC* configuration of ZOLC hardware have been evaluated for these measurements. For the former case, loop distribution was applied by hand for generating maximal feasible loop structures for the *fsme_dr* and *rcdct* kernel benchmarks.

The relative cycle measurements for the examined benchmarks are given in Fig. 10. It can be seen that the use of ZOLC in its 'lite' configuration is responsible for an average reduction in execution cycles of 25.5% and 10% on the selected kernels and applications, respectively. For some algorithms (*rcdct, matmult*), performance improvement is well over 40% when applying the ZOLC principle. Also, it can be concluded that an *uZOLC* configuration is a good compromise for most kernel benchmarks, especially those featuring fully-nested loops where a single inner loop is of interest for optimization. Overall, an average 17.5% performance improvement is obtained with *uZOLC*.

To support our argument of ZOLC efficiency, we have performed measurements on a representative processor with native ZOL optimizations for single loops, namely the LX 1.0 variant of the Xtensa family [7]. The Xtensa compiler, *xt-xcc*, is based on *gcc* enhanced with a number of extensions. The data-intensive kernels as well as 7 out of 10 application benchmarks were successfully compiled and ran with (*-O3*) and without ZOL support (*-O3 -mno-zero-cost-loop*) with both the automatic generation of custom instructions turned on (*-xpres*) and off. The

TABLE 4: Summary of examined benchmark kernels and applications.

Benchmark	Description	Dynamic instructions
Data-intensive kernels		
fsme	Full-search motion estimation	22,030,675
tssme	Three-step logarithmic search motion estimation	2,101,005
fsme_dr	Full-search motion estimation with data-reuse transformations	13,052,691
rcdct	Row-column decomposition forward DCT	836,568
matmult	Blocked matrix multiplication	374,355
edgedet	Gradient-based edge detection	91,091
mc	Motion compensation	113,382
kmpskip	Knuth-Morris-Pratt string matching algorithm	30,437
lcs	Longest common substring search algorithm	1,313,236
wsqc	Wavelet-Scalar Quantization image compression	479,596
Embedded applications		
adpcm-encode	Adaptive Differential Pulse Code Modulation (ADPCM) encoder	34,628,152
adpcm-decode	Adaptive Differential Pulse Code Modulation (ADPCM) decoder	27,256,674
dijkstra	Shortest path calculation between node pairs with Dijkstra's algorithm	59,354,952
rijndael-encode	Advanced Encryption Standard encoder	33,697,370
rijndael-decode	Advanced Encryption Standard decoder	34,666,810
sha	Secure Hash Algorithm producing an 160-bit message digest for a given input	13,036,406
stringsearch	Case-insensitive string matching	279,725
susan-corners	Corner detection with the SUSAN image processing package	3,469,990
susan-edges	Edge detection with the SUSAN image processing package	6,898,760
susan-smoothing	Structure-preserving image noise reduction with the SUSAN image processing package	35,331,316

performance results are illustrated in Fig. 10 alongside the measurements for the ZOLC-enhanced XiRisc. It can be seen that the performance speedups due to ZOL on Xtensa are rather limited ranging from 3% to a maximum of 7.5%. The main reason behind this is the inability of exploiting an opportunity for ZOL when the relevant code region incorporates conditional control flow. In contrast to that, for the case of ZOLC, conditional control flow in *bwd* tasks is directly supported. In some cases (e.g. *sha*) the Xtensa ZOL may perform better than ZOLC, however this is due to the more aggressive optimizations applied by *xt-xcc* performs well, resulting in reduced machine cycles for the DPTs of interest. Thus, a better cycle benefit is obtained, assuming that the number of cycles removed due to ZOLC is not affected.

Moreover, we have evaluated a few benchmarks embodying regions of irreducible control flow: an implementation of Duff's device (*duff*) taken from the WCETbench suite [36] and three variations of the basic irreducible control flow graph discussed in [14] with different DPT sizes, the latter coded in MIPS-I assembly. Since current *gcc*-based compilers do not optimize over such regions, no loop optimization is exploited for either processor. For small DPT sizes, the performance benefits from using 'ZOLCfull' are significant; if the DPT size is increased to 10 instructions, the cycle advantage is reduced, however it remains non-negligible as can be seen by studying Table 5.

7.3 Experiments on a DSP ASIP

The portable RTL specification of ZOLC has been reused in a cycle-accurate model for the hDSP in-house ASIP written in ArchC [37]. Instruction-set extensions as the 'absolute value difference and accumulation' $(dabsa)$ are available on the hDSP in order to accelerate the execution of performance-critical kernels, for example the SAD criterion in motion estimation. The average basic block and TCFG sizes were significantly reduced due to the complex instructions.

A small set of 2 synthetic and 3 real kernels have been examined for the hDSP, and the corresponding results are shown in Table 6. The average task size and ZOLC initialization cycles are also given. Performance speedup was found at 44.15% in average for the real benchmarks, while speedups of about 75% can be obtained for extreme cases (*loop4*, *loop8*) of one useful computational operation per task for the considered ASIP (looping overhead instruction pattern executes in 4 cycles).

8 Conclusion

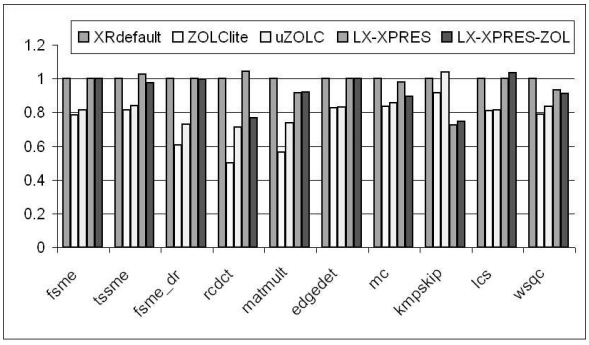
In this paper, a zero-overhead loop controller suited to embedded RISC microprocessors is introduced. The presented architecture is able to execute complex loop

TABLE 5: Performance results for instances of irreducible control flow.

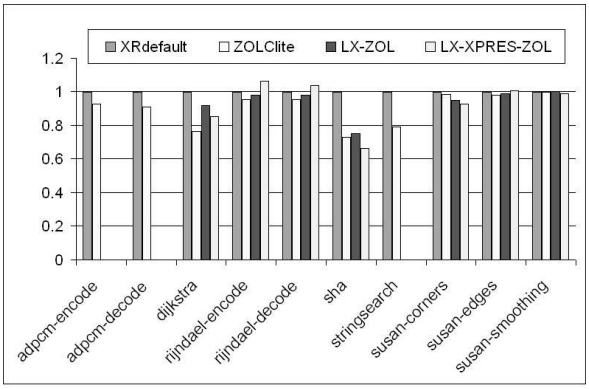
Benchmark	Description	Cyc. with ZOL (ZOLC if MIPS-I)	Cyc. without ZOL (ZOLC if MIPS-I)	%diff
duff	Duff's device [36] on MIPS-I	1241	1750	29.1
duff	Duff's device [36] on Xtensa LX	1300	1300	0.0
bifg1	Irreducible CFG with DPT size = 1	12	64	81.25
bifg5	Irreducible CFG with DPT size = 5	60	119	49.6
bifg10	Irreducible CFG with DPT size = 10	120	175	31.4

TABLE 6: 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
fsme_dr	2.23	164	20,296,016	48,470,213	58.13



(a)



(b)

Fig. 10: Cycle performance results for the examined benchmarks. (a) Kernel benchmarks. (b) Application benchmarks selected from the MiBench suite.

structures, even incorporating irreducible control-flow regions, with no cycle overheads incurred for task switching. A novel graph representation, namely the task control-flow graph has been described, that can be used for ZOLC code generation. An extensive set of loop-level as well as hardware-dependent transformations can be applied on the TCFG of a given application, making TCFG construction and restructuring, an interesting compilation pass for embedded ASIPs. Special cases of nesting structures as multiple-entry loops can be exploited at the TCFG level, despite the fact that they cannot be optimized by conventional compiler techniques. 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 25.5% and 44.1% have been observed for the XiRisc processor and an in-house ASIP, respectively. Finally, an automated tool has been developed for constructing the TCFG of a loop structure and producing ZOLC-aware code.

REFERENCES

[1] ARM ltd. [Online]. Available: http://www.arm.com
[2] MIPS technologies inc. [Online]. Available: http://www.mips.com
[3] Motorola, *DSP56300 24-Bit Digital Signal Processor Family Manual*, 3rd ed., December 2000.
[4] STMicroelectronics, *ST120 DSP-MCU Core Reference Guide*, 1st ed.
[5] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, December 1994.
[6] ARC cores. [Online]. Available: http://www.arccores.com
[7] R. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, March-April 2000.
[8] D. Talla, L. K. John, and D. Burger, "Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements," *IEEE Transactions on Computers*, vol. 52, no. 8, pp. 1015–1031, August 2003.
[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*, Villach, Austria, September 2001, pp. 456–459.
[10] J. Kang, J. Lee, and W. Sung, "A compiler-friendly RISC-based digital processor synthesis and performance evaluation," *Journal of VLSI Signal Processing*, vol. 27, pp. 297–312, 2001.
[11] J.-Y. Lee and I.-C. Park, "Loop and address code optimization for digital signal processors," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science*, vol. E85-A, no. 6, pp. 1408–1415, June 2002.
[12] M. Kuulusa, J. Nurmi, J. Takala, P. Ojala, and H. Herranen, "A flexible DSP core for embedded systems," *IEEE Design and Test of Computers*, vol. 3, no. 4, pp. 60–68, October 1997.
[13] L. Carter, J. Ferrante, and C. Thomborson, "Folklore confirmed: Reducible flow graphs are exponentially larger," in *ACM Symposium on the Principles of Programming Languages (POPL)*, New Orleans, Louisiana, USA, January 2003, pp. 106–114.
[14] J. Janssen and H. Corporaal, "Making graphs reducible with controlled node splitting," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 6, pp. 1031–1052, November 1997.

[15] S. Unger and F. Mueller, "Handling irreducible loops: Optimized node splitting vs. DJ-graphs," *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 4, pp. 299–333, 2002.
[16] G.-R. Uh, Y. Wang, D. Whalley, S. Jinturkar, Y. Paek, V. Cao, and C. Burns, "Compiler transformations for effectively exploiting a zero overhead loop buffer," *Software: Practice and Experience*, vol. 35, no. 4, pp. 393–412, April 2005.
[17] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison-Wesley, 1986.
[18] *MPEG-4 Video Verification Model Version 18.0*, International Organization of Standardization, Working Group on Coding of Moving Pictures and Audio, Pisa, Italy, January 2001.
[19] Graphviz. [Online]. Available: http://www.research.att.com/sw/tools/graphviz/
[20] N. Kavvadias and S. Nikolaidis, "Zero-overhead loop controller that implements multimedia algorithms," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 4, pp. 517–526, July 2005.
[21] S. Kumar, L. Pires, S. Ponnuswamy, C. Nanavati, J. Golusky, M. Vojta, S. Wadi, D. Pandalai, and H. Spaanenburg, "A benchmark suite for evaluating configurable computing systems - status, reflections, and future directions," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, USA, February 2000.
[22] C. Pan, N. Bagherzadeh, A. H. Kamalizad, and A. Koohi, "Design and analysis of a programmable single-chip architecture for DVB-T base-band receiver," in *Proceedings of the Design, Automation and Test in Europe Conference*, Munich, Germany, March 2003, pp. 468–473.
[23] M. E. Benitez and J. W. Davidson, "A portable global optimizer and linker," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, USA, 1988, pp. 329–338.
[24] M. D. Smith and G. Holloway, "An introduction to machine SUIF and its portable libraries for analysis and optimization," Division of Engineering and Applied Sciences, Harvard University, Cambridge, Mass., MachSUIF documentation 2.02.07.15 edition, 2002.
[25] R. M. Stallman, *GNU Compiler Collection Internals for GCC 3.3.2*, Free Software Foundation, Inc., Cambridge, Massachusetts, USA, December 2002. [Online]. Available: http://gcc.gnu.org/onlinedocs/gccint/
[26] N. Kavvadias. Hardware looping unit. [Online]. Available: http://www.opencores.org/projects.cgi/web/hwlu/overview/
[27] Machine-SUIF research compiler. [Online]. Available: http://www.eecs.harvard.edu/hube/research/machsuif.html
[28] E. Rohou, F. Bodin, A. Seznec, G. L. Fol, F. Charot, and F. Raimbault, "SALTO: System for assembly-language transformation and optimization," Institut National de Recherche en Informatique et en Automatique, Technical report 2980, September 1996.
[29] The ArchC resource center. [Online]. Available: http://www.archc.org
[30] 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 2004.
[31] L. George and A. W. Appel, "Iterated register coalescing," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 3, pp. 300–324, May 1996. [Online]. Available: http://citeseer.nj.nec.com/george96iterated.html
[32] J. Eyre and J. Bier, "Evolution of DSP processors," *IEEE Signal Processing Magazine*, vol. 17, no. 2, pp. 43–51, March 2000.
[33] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, July 1987.
[34] T. P. Shevlin, "Composed control dependence graph generator," Master's thesis, Dept. of Computer Science, Tufts University, Medford, Massachusetts, USA, September 2004.
[35] A. Aggarwal and M. Franklin, "Energy efficient asymmetrically ported register files," in *Proceedings of the 21st International Conference on Computer Design*, San Jose, CA, USA, October 2003, pp. 2–7.
[36] WCET project benchmarks. [Online]. Available: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[37] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC architecture description language and tools," *International Journal of Parallel Programming*, vol. 33, no. 5, pp. 453–484, October 2005.

Nikolaos Kavvadias received the B.Sc. degree in physics and M.Sc. in electronics engineering from the Aristotle University of Thessaloniki, Greece in 1999 and 2002, respectively. He is currently in pursuit of his Ph.D. degree in computer engineering from the same university. His current research interests include hardware description languages, application-specific processor design methodologies, and energy consumption modeling for embedded processors.

Spiridon Nikolaidis received the Diploma and PhD degrees in electrical engineering from Patras University, Greece, in 1988 and 1994 respectively. He is an Assistant Professor at the Electronics Laboratory of the Department of Physics of the Aristotle University of Thessaloniki, Greece. His research interests include design of low power high speed processor architectures, CMOS gate propagation delay and power consumption modeling, high speed and low power CMOS circuit techniques. He has published as author and coauthor more than 80 papers in international scientific journals and conferences. He is also involved in research projects funded by European and National resources.