Issue-Slot Based Predication Encoding Technique for VLIW Processors

Lukas Gerlach, Fabian Stuckmann, Holger Blume, and Guillermo Payá-Vayá
Institute of Microelectronic Systems
Leibniz Universität Hannover
Cluster of Excellence Hearing4all
Hannover, Germany
Email: {gerlach, stuckmann, blume, guipava}@ims.uni-hannover.de

***Abstract*—Predication is a well-known alternative to conditional branching. However, the implementation of predication is costly in terms of extending the instruction set of the processor architecture. In this paper, a predication encoding technique for VLIW processors is proposed. Instead of using additional bits in the instruction encoding, the assigned issue-slot of a conditionally executed instruction encodes the associated predicate register. The number of addressable predicate registers scales with the number of issue-slots. All predicate registers have only one read and write port and can be accessed in parallel. Compared to the related work, no additional instruction encoding bits for selecting a predicate register are required and the processor core area increases only by about $1\,\%$ per predicate register set. With the proposed predication technique, the processing performance increases by up to $4.5\,\%$ when using two instead of one predicate register for a digital filter case study with floating-point emulation operations. A second case study shows, that conditional execution with two predicate register in combination with loop unrolling and operation merging almost doubles the achieved parallel instructions per cycle for a bit-reversal permutation algorithm.**

***Index Terms*—predication, conditional execution, VLIW**

I. Introduction

Very long instruction word (VLIW) processors are commonly used for embedded high performance and low-power multimedia applications [1]–[8]. VLIW processors are designed for instruction level parallelism (ILP), executing multiple instructions with a fixed order in parallel. Since the order of the instructions is determined by the compiler, the hardware is less complex compared to superscalar architectures. Therefore, to fully utilize hardware resources, ILP compiler optimizations are required [1], [8].

One limitation for further compiler optimizations are branch instructions, which cause smaller basic blocks (straight line microcodes (SLMs)) in the code, restricting the scope of ILP optimizations [8]. These branch instructions, which are required for the control flow of an application, are expensive in terms of processing performance [7]. The condition of a branch cannot be evaluated at the beginning of the pipeline. Hence, the successive instructions already fetched and decoded must be discarded, while the pipeline is flushed, or are executed whether or not the conditional branch is taken. One alternative to conditional branches is predication [2], [8]. Predication is the conditional execution or guarded execution of instructions. The instructions of both instruction-sequences of the conditional code are executed, but only the instructions of one of the sequences change the state of the processor and memories, depending on the value (condition flag) stored in a predication or guard register [8]. The remaining instructions act as no operation (NOP) instructions. Branches may be replaced by conditionally executed instructions during the *if*-conversion [2], [3], [9]. The control dependencies are converted to data dependencies by converting multiple regions of a control flow graph into a basic block, composed of predicated (conditional) code [1].

However, the application of a predication technique requires an extension of the instruction set architecture (ISA) of the processor [5], [8]. Additional instruction encoding bits are needed in order to address one of the predicate register. But adding for example 6 instruction encoding bits to address 64 predicate register is prohibitively too expensive for embedded processors [8]. Studies for related embedded processors [10], [11] show the importance of the instruction memory size on power consumption. As shown in [10], the memory subsystem of *ARM* processor [12] accounts for $65.2\,\%$ of the total energy. For multi-processor *ARM* systems, this proportion is $45.9\,\%$. The instruction memory cache alone accounts for $20.6\,\%$ in this system. The *ARM* processors with the reduced $16\,\mathrm{bit}$ *Thumb* instruction set are more energy efficient compared to the $32\,\mathrm{bit}$ *ARM* processors, however the processing performance is lower [10]. The instruction caches of the *TMS320C6000* VLIW processor family [13] account for up to $30\,\%$ of the total processor power [14], [15]. Besides the power consumption, the area requirements of the instruction memory are crucial for embedded processors. Each additional instruction bit increases the required SRAM area linearly, as depicted in Fig. 1 for a $40\,\mathrm{nm}$ ASIC technology.

Additionally, the number of read and write ports of the predicate register file and the instruction fetch and decode logic are important for the resulting hardware complexity [5], [11]. The instruction fetch and decode logic of a DSP can consume up to $40\,\%$ of the total processor logic [11], [16]. Therefore, the

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy — EXC 2177/1 — Project ID 390895286.

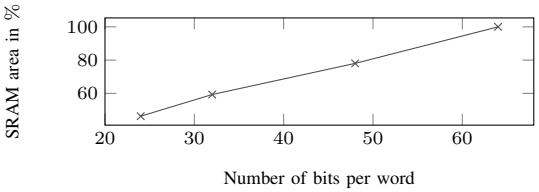


Fig. 1. Area of SRAM macro blocks of a 40 nm ASIC technology with different number of bits per word. The area of the SRAM macro block with 64 bits per word is used as a reference.

goal is to reduce the overhead caused by implementing the predication technique in these processors [5].

In this paper, a new scalable and low overhead predication technique for VLIW processors is proposed. Instead of encoding the address of the predicate registers with additional instruction encoding bits, each issue-slot of the VLIW architecture includes a dedicated predicate register. With the help of compiler optimizations, which are also presented in this paper, instructions are scheduled on the issue-slot with the corresponding predicate register. This technique scales with the number of issue-slots, requires no additional instruction encoding bits, and therefore decreases the hardware overhead for predication.

This paper is organized as follows. The related predication techniques are presented in Section II. The VLIW architecture is described in Section III and the details of the proposed predication technique are given in Section IV. The case studies in Section V and Section VI show the application and evaluation of the technique. The paper is concluded in Section VII.

II. Related Work

An overview of the related predication techniques is given in Table I. The number of predicate registers and the required instruction encoding bits to address these registers are compared. The number of predicate registers determines the maximum number of conditional statements, which can be processed in parallel. The cause for these conditional statements are parallel or nested *if-else* constructs. A comparatively high number of up to 64 predicate registers is used by [2], [17]–[19]. Consequently, these architectures require most instruction encoding bits to address one of the predicate registers. The number of encoding bits is reduced in [19], by splitting the registers into two sets. A special instruction is used to switch between the two register sets. The *ARMv7-A* [12] architecture includes one *application program status register* (APSR), which holds four different conditions: negative, zero, carry and overflow. In order to select these conditions 4 bits are required for every instruction [8]. In [5], [13], [20] the predication technique of the *TMS320C6X* processor family is described. The condition flags are stored in a restricted number of registers of the general purpose register file. The limited number of six predicates for the *TMS320C6X* processor family results in a limited control-flow nesting [8]. As every *if-then-else* statement requires two predicates, only two levels of control-flow nesting are possible. The number of read and write ports of the general purpose processor register file needs to be increased and the access and usage patterns might influence overall performance [21]. However, no dedicated predicate register file is required and the predicate registers can be used otherwise if they are not needed. The authors of [5] propose to reduce the overhead of predication by restricting the number of predicate registers to one. Consequently, only one additional read port for the predicates is required. Four ports were previously required by the four issue-slots of the VLIW architecture to read four predicate registers per cycle. Instead of 3 bits for the predicate operand, only 1 bit is required. Nested and parallel *if-else* statements can not be fully predicated with only one predicate register [5].

TABLE I
Comparison of related predication techniques. The required instruction encoding bits are those that address one predicate register.

Architecture	Number of predicate registers	Required instruction encoding bits
Itanium IA-64 processor [17], [18]	64	6 bits
Generic ILP processor [2]	32	5 bits
ARMv7-A [12]	1	4 bits
TMS320C64x/C67x VLIW [13], [20]	5	3 bits
PLX [19]	128*	3 bits
HP VLIW ST231 ISA [5]	1	1 bit
KAVUAKA [this work]	16**	none

* (16 sets of 8) ** (8 per issue-slot, one for each SIMD subword)

In this paper, a predication technique is presented, which does not require any instruction bits to encode the predicate registers. The predicate registers are encoded by scheduling the conditional instructions on different issue-slots. The instruction bits saved compared to the related architectures are summarized in Table I. Due to the issue-slot based predicate encoding, the proposed technique decreases the required instruction memory size, power consumption and the complexity of the instruction decoding stage compared to the related work. A compiler extension for this encoding technique is presented, which handles the predicate register allocation automatically, without the need to manually encode the predicate register within the instructions.

III. Generic VLIW Processor Architecture

The proposed predication technique is evaluated with a VLIW processor called *KAVUAKA* [22], which is shown in Fig. 2. *KAVUAKA* is an application-specific instruction-set processor (ASIP). The architecture is based on the *Moai4k2* architecture, a scalable and configurable ASIP architecture for multimedia applications [23]. *KAVUAKA* supports instruction and data level parallelism with very long instruction words (VLIWs) and single instruction, multiple data (SIMD). The architecture was implemented with the hardware description language VHDL, so that the processor configuration can be

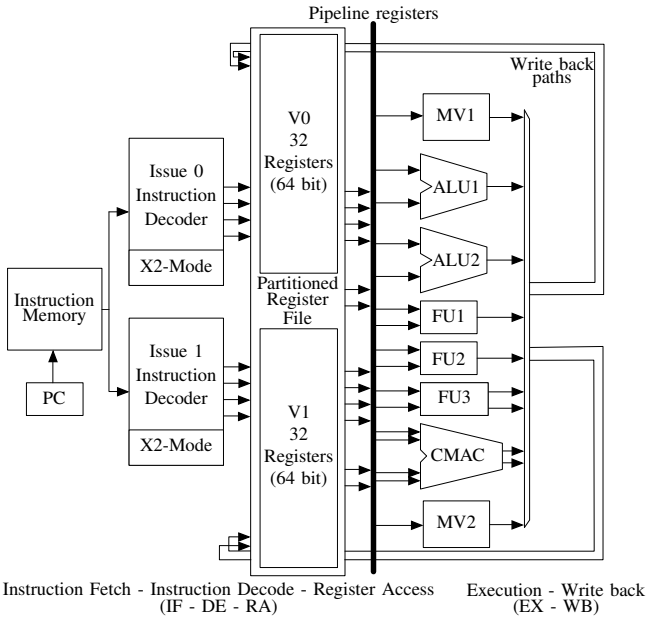
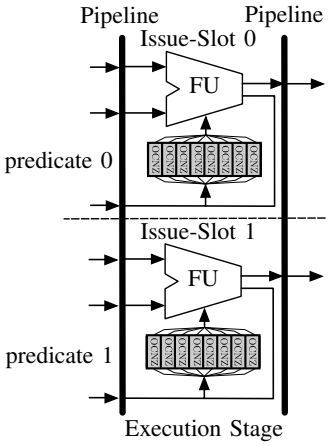


Fig. 2. *KAVUAKA* application-specific instruction-set processor (ASIP) architecture. A vector unit with two pipeline stages, two instruction decoders, a partitioned register file and several execution units is shown.



Predicate register

OCNZ	OCNZ	...	OCNZ	OCNZ
SW #7	SW #6	...	SW #1	SW #0

Predicate register content: flags (32 bits) with 4 conditions per SIMD subword (SW): overflow (O), carry (C), negative (N) and zero (Z)

Fig. 3. One dedicated predicate register is part of each issue-slot. Each functional unit (FU) reads the condition flags for each subword from the predicate register of the same issue-slot.

customized by a set of parameters, such as the number of predicate registers.

Fig. 2 shows a configuration consisting of one vector unit with two issue-slots, labeled *Issue 0* and *Issue 1*. Two 32-bit wide micro-operations (MOs) are executed in parallel, which, according to the VLIW philosophy, are combined into a 64-bit wide instruction word called a micro-instruction (MI). Two micro-operations (MOs) of the same type, i.e. with the same instruction encoding, may be merged into one. These merged operations, called X2-operations, use two identical functional units of the execution (EX) stage to process twice the number of input values stored in consecutive register pairs, thereby doubling the number of parallel executed operations. This kind operation merging technique is introduced in [24] as the X2-mode. The structure of the vector unit is based on reduced instruction-set computer (RISC) principles, where every instruction takes only one cycle to execute. The number of pipeline stages is two.

The processor is programmed with assembly language that maps human-readable instructions directly to micro-operations defined in the instruction set architecture (ISA). A scheduler translates the assembly code into micro-instructions, taking into account control and data dependencies as well as the parameterized processor configuration. It uses evolutionary algorithms for scheduling and register allocation for optimizations [25].

IV. Proposed Issue-Slot Based Predication Technique

In this section, the hardware architecture (Section IV-A) and the predicate register allocation (Section IV-B) are described.

A. Hardware Architecture

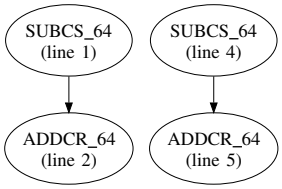
A new predication technique is presented, which exploits the issue-slot based processing of VLIW processors. The architecture is depicted in Fig. 3. A conditional instruction, which is defined by setting one instruction encoding bit to '1', receives or sets the condition flags of one predicate register. Instead of selecting a predicate register using additional instruction encoding bits, the predicate register is selected by the issue-slot, on which the conditional instruction is scheduled. Each issue-slot of the VLIW processor contains one dedicated predicate register. The predicate register is selected with the issue-slot, on which the instruction has been scheduled. No additional bits for addressing the predicate register are required in the instruction encoding. Every conditional instruction, which reads or writes to same predicate register, is scheduled on the same issue-slot. This technique requires that the issue-slots of the VLIW architecture are identical/symmetric in terms of functional and data movement units (FUs), so that the instruction scheduler can switch the instructions between the slots without constraints.

These predicate registers contain four condition flags (overflow (O), carry (C), negative (N) and zero (Z)) for each subword of one SIMD data word, as shown in Fig. 3 and proposed by [8], [21]. On the one hand, this format can be used to process conditional SIMD instructions on subword-level, with up to 8 subwords in parallel depending on the condition flags stored in the corresponding position in the predicate registers. On the other hand, each single subword with four condition flags can be used for nested *if-else* statements. This use case assumes that each subword holding the conditions can be selected and used for one of the conditional *if-else*

Sequential assembler code with two conditions

```
1 SUBCS_64 V0R0, V0R1, V1R2
2 ADDCR_64 V0R3, V0R4, V1R5
3
4 SUBCS_64 V0R6, V0R7, V1R8
5 ADDCR_64 V0R9, V0R10, V1R1
```

Corresponding data dependency graph (DDG)



Scheduled code on two issue-slots

Issue-Slot 0	Issue-Slot 1
1 **SUBCS_64** V0R6 V0R7 V1R8;	**SUBCS_64** V0R0 V0R1 V1R2
2 **ADDCR_64** V0R9 V0R10 V1R1;	**ADDCR_64** V0R3 V0R4 V1R5

Fig. 4. Exemplary assembler code with two independent conditions. The corresponding DDG representing the condition flag data dependency is depicted. The condition set (CS) and condition read (CR) instructions are scheduled on two issue-slots in parallel, using a separate predicate register.

statements. The total number of predicates is therefore 16, stored in 2 predicate registers with 8 subwords holding 4 condition flags each.

The number of predicates scales with the number of issue-slots and available SIMD subwords of the processor. The number of read and write ports of the predicate register file is one. The area overhead for implementing one additional predicate register per issue-slot with one read and write port is around $1\,\%$ of the total core cell area of the *KAVUAKA* processor for an application-specific integrated circuit (ASIC) synthesis with a $40\,\mathrm{nm}$ low-power technology [22].

B. Predicate Register Allocation

The issue-slot of conditional instructions is bound to the issue-slot of the associated predicate register. Therefore, the predicate register is allocated during instruction scheduling. The instruction scheduler determines the issue-slot for every instruction for a given application. The predicate register resource dependencies are checked during instruction scheduling using data dependency graphs (DDGs). The DDGs indicate, which conditional instructions are interdependent and must therefore be scheduled before other conditional instructions can be scheduled. A DDG representation for a code with two conditionally executed *addition (ADD_CR)* (condition read (CR)) instructions is shown in Fig. 4. There is no dependency between these two instructions, because there is no connection between the subtrees in the DDG. These can be scheduled in parallel on two issue-slots, using two different predicate registers set by the *subtraction (SUB_CS)* (condition set (CS)) instructions. The instruction scheduler can optimize the issue-slot and predicate register allocation for an overall efficient scheduling [25].

TABLE II
FLOATING-POINT EMULATION MACROS

Floating-Point (FP) macro	Number of conditional operations	Number of macro calls
FP_ADD	5 (26 %)	8
FP_SUB	5 (26 %)	8
FP_MUL	2 (14 %)	16

V. CASE STUDY: FLOATING-POINT EMULATION

In this case study, floating-point emulation code with a high number of conditionally and independently executed operations, like overflow and sign checks, normalization operations and bitwise comparisons, is used. The underlying floating-point emulation library is described in detail in [26]. The computation of the addition and subtraction operations of this library is shown in Fig. 5. For a floating-point addition or subtraction, the exponents of both numbers have to be adjusted to the same value. Therefore, the absolute difference of the exponents is computed. Based on the difference, the significand of the smaller number is shifted right by the absolute difference of the exponent and the larger exponent is selected. The significands are then added or subtracted from each other. After normalization of the significand, the exponent is updated and the floating-point addition or subtraction is computed.

The floating-point emulation addition assembler code macro is shown in Fig. 6. Here the exponents are subtracted from each other. Depending on the result of this subtraction, one of the significands is chosen for shifting before adding the significands. The normalization is performed by counting leading zeros or ones. The resulting register is then rebuilt from the updated exponents and the normalized significands.

In the related work presented in [27] and [28], processor architectures with multiple conditional registers and predicated execution are selected for floating-point emulation. In this case study, the processing performance is evaluated based on the number of available predicate registers.

In this case study, a finite impulse response filter (FIR) and infinite impulse response filter (IIR) filter with a order of 17 and 7 are computed using floating-point emulation macros. Table II lists the floating-point point emulation macros and how often these macros are called during the computation. These macros contain up to $26\,\%$ conditional operations. The number of predicate registers determines how many operations of these floating-point emulation macros can be scheduled in parallel.

The required number of processing cycles for computing one sample with one or two predicate registers is shown in Table III. When two predicate registers can be accessed in parallel, the number of required processing cycles decreases by $4.4\,\%$ and the instructions per cycle (IPC) increases from 1.82 to 1.91 for the *KAVUAKA* processor with a maximum IPC of 2, when no operation merging is used.

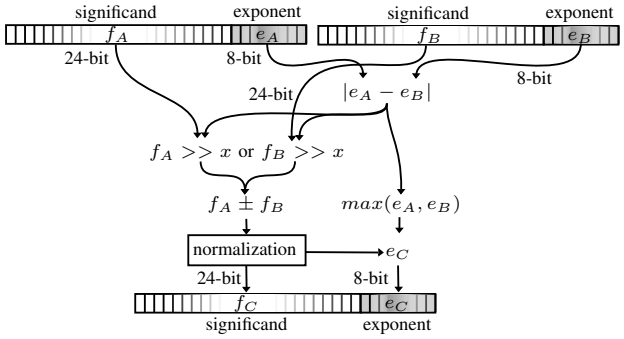


Fig. 5. Optimized floating-point addition for SIMD processors.

```
// ***************************************************
// emulated floating-point addition macro (Q 24.8 format)
// ***************************************************
MACRO FP_ADD_32 DST, OP1, OP2
// compare exponents
PERMREG0_8  xOP1, OP1,  OP2  // E0|E0|E2|E2|E1|E1|E3|E3
PERMREG0_8  xOP2, OP2,  OP1  // E2|E2|E0|E0|E3|E3|E1|E1
SUBCS_8s    TEMP, xOP1, xOP2 // exponent difference
// swap mantissas according to magnitude of exponents
// (choose number with smaller exponent)
MV          TEMP1, OP1
MV          TEMP4, OP2
MVCR_32     TEMP1, OP2
MVCR_32     TEMP4, OP1
// remove exponent (for significand computation)
SRI_32      TEMP5, TEMP1, #8
// right shift of secondary significand operand
ABSADD_8    TEMP, R_8, TEMP
CLIPI_U8    TEMP, TEMP, #0b11111
SR_32       TEMP, TEMP4, TEMP
// add mantissas
ADD_32      TEMP5, TEMP5, TEMP
// normalize resulting significand
CLX_32      TEMP, TEMP5, TEMP5
SMVI        V0CONDSEL, #0b0010
SLCS_32     TEMP5, TEMP5, TEMP
// compute new (resulting) exponent
SUB_8s      TEMP, R_8, TEMP
ADD_8s      TEMP, TEMP1, TEMP
// recreate original format
PERMREG1_8  DST, TEMP5, TEMP
MVCR_32 DST, REG_ZERO
ENDMACRO
```

Fig. 6. Assembler code macro for emulating the floating-point addition using SIMD instructions.

TABLE III
PROCESSING PERFORMANCE IN NUMBER OF CYCLES

	one predicate register	two predicate register
17-tap FIR	740	707 ($-4.4\,\%$)
7-tap IIR	300	287 ($-4.3\,\%$)

VI. CASE STUDY: LOOP UNROLLING AND INSTRUCTION MERGING

An example for an *if*-conversion based on partial predication is shown in Fig. 7. A reference code for bit-reversal permutation is given, which is required for the radix-2 Cooley-Tukey FFT algorithms [29]. The *if* construct is evaluated within the for loop for every index of the input vector.

Taking the bit-reversal permutation code in Fig. 7 as an example, the inner loop can be unrolled to increase the size of the basic block. This loop including the *if* condition is implemented on the *KAVUAKA* processor with conditional branches or predication with one and two predicate registers. Different levels of loop unrolling are evaluated, measuring the dynamic IPC. All ILP compiler optimizations are turned on, which includes automatic operation merging [25]. The

Reference *Matlab* code for bit-reversal permutation of the vector x. The *if* condition is needed for swapping elements based on their position (index) within the vector.

```
function x = bit_reversal(x)
    number_of_elements = length(x);
  for i = 1:length(x)
      bit_reversed_index = br_index(i-1, number_of_elements);
      if(0<(bit_reversed_index-(i-1)))
          temp = x(bit_reversed_index+1);
          x(bit_reversed_index+1) = x(i);
          x(i)=temp;
      end
  end
end
```

Conditional branch implementation

```
SUBCS_64 REG_Z,REVERSED_INDEX,INDEX
//Conditional branch (greater than)
BSR_AND NO_SWAP, #0b1001, #0b10000000
//Load two elements
MV_64 REG_INDEX_plus_1, (FIR_INPUT_ELEMENT_ADDRESS)
MV_64 REG_i, (FIR_OUTPUT_ELEMENT_ADDRESS)
//Store elements in memory
MV_64 (FIR_INPUT_ELEMENT_ADDRESS), REG_i
MV_64 (FIR_OUTPUT_ELEMENT_ADDRESS), REG_INDEX_plus_1
:L_NO_SWAP
```

Scheduled code: conditional branch implementation

```
SUBCS_64    V1R30   V0R2    V0R0;   NOP
BSR_AND     NO_SWAP 0x9     0x80;   NOP
NOP                         ;   NOP
MV_64       V1R30   FIR_IND1    ;   NOP
MV_64       FIR_IND0    V1R30   ;   MV_64   V1R30   FIR_IND0
MV_64       FIR_IND1    V1R30   ;   NOP
:L_NO_SWAP
```

Conditional execution implementation

```
//Set condition (greater than)
SMVI_64 V0CONDSEL, #0b0101
SUBCS_64    REG_Z,REVERSED_INDEX,INDEX
//Load two elements
MV_64 REG_INDEX_plus_1, (FIR_INPUT_ELEMENT_ADDRESS)
MV_64 REG_i, (FIR_OUTPUT_ELEMENT_ADDRESS)
//Conditionally swap to elements
MV_64   REG_TEMP, REG_INDEX_plus_1
MVCR_64 REG_INDEX_plus_1, REG_i
MVCR_64 REG_i, REG_TEMP
//Store elements in memory
MV_64 (FIR_INPUT_ELEMENT_ADDRESS), REG_i
MV_64 (FIR_OUTPUT_ELEMENT_ADDRESS), REG_INDEX_plus_1
```

Scheduled code: conditional execution implementation

```
SMVI_64 V0CONDSEL, #0b0101      ;   NOP
MV_64   V1R0    FIR_IND1        ;   SUBCS_64    V1R30   V1R1    V1R30
MV_64   V1R1    FIR_IND0        ;   NOP
MV_64   V1R30   V1R1            ;   MVCR_64     V1R1    V1R0
MV_64   FIR_IND1    V1R1        ;   MVCR_64     V1R0    V1R30
MV_64   FIR_IND0    V1R0        ;   NOP
```

Fig. 7. Code example with a conditional *if* construct. The example is a bit-reversal permutation of an input vector, which is required for the Cooley–Tukey FFT algorithm. The two assembler implementations use conditional branches or execution. The scheduled code of the two assembler implementation is also given.

results are depicted in Fig. 8. The conditional branch (BR) implementation requires fewer cycles than the predicated version (CE1_PR), although the branch requires a branch delay slot with a NOP instruction. The reason for this is that *if-else* statement is unbalanced [30]. Increasing the predicate registers to two (CE_2PR) does not improve performance, since only one condition is used per loop iteration. When the loop unrolling is applied, the size of the of the basic block and the number conditions, that can be processed in parallel, is increased. Additionally, more operations can be merged. The achieved IPC increases from 1.53 (CE_1PR) to 2.88 (CE_2PR_2LU), when using 2 predicate registers, loop unrolling and operation merging. Loop unrolling and operation merging are not as effective when using conditional branches

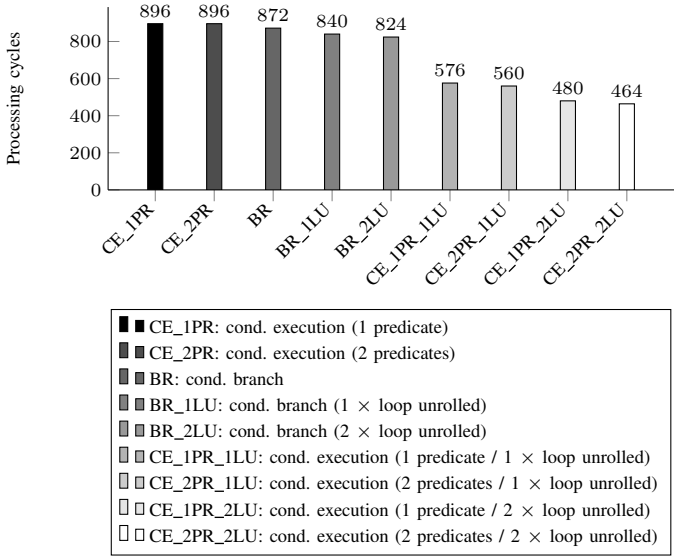


Fig. 8. Processing performance in number of cycles for the bit-reversal permutation (see Fig. 7) of 32 elements with conditional branches, predication with one and two predicate registers with different levels of loop unrolling. Automatic operation merging is activated.

(BR_1LU, BR_2LU) because the basic blocks are smaller and prevent further compiler optimizations.

VII. Conclusion

This paper presents an issue-slot based predication technique for VLIW processors, that does not require additional instruction encoding bits to address a predicate register. The predicate registers are addressed based on the issue-slot, on which the conditional instructions are scheduled. Multiple predicate registers can be accessed simultaneously. The core area overhead for multiple predicate registers is about $1\,\%$. Only one read and write port is required per register. A compiler optimization approach is presented, to optimize the predicate register allocation. Two case studies show a performance increase by about $4\,\%$ when using 2 instead of 1 predicate register in parallel, reaching almost the maximum possible IPC.

References

[1] P. Faraboschi, J. A. Fisher, and C. Young, "Instruction scheduling for instruction level parallel processors," *Proceedings of the IEEE*, vol. 89, no. 11, pp. 1638–1659, 2001.

[2] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2, pp. 138–150, 1995.

[3] B. R. Rau and J. A. Fisher, "Instruction-level parallelism," 2003.

[4] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley, "Dataflow predication," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 89–102.

[5] R. A. Starke, A. Carminati, and R. S. de Oliveira, "Evaluation of a low overhead predication system for a deterministic VLIW architecture targeting real-time applications," *Microprocessors and Microsystems*, vol. 49, pp. 1–8, 2017.

[6] D. N. Pnevmatikatos and G. S. Sohi, *Guarded execution and branch prediction in dynamic ILP processors*. IEEE Computer Society Press, 1994, vol. 22, no. 2.

[7] C. W. Kesseler, "Compiling for VLIW DSPs," in *Handbook of Signal Processing Systems*. Springer, 2018, ch. 3.

[8] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[9] J. Crawford and F. J. Huck, "Next Generation Instruction Set Architecture," in *Microprocessor Forum*, 1997.

[10] M. Verma and P. Marwedel, "Memory wall problem," in *Advanced memory optimization techniques for low-power embedded processors*. Springer, 2007, vol. 1, ch. 1.1.1.

[11] A. Artes, J. L. Ayala, J. Huisken, and F. Catthoor, "Survey of low-energy techniques for instruction memory organisations in embedded systems," *Journal of Signal Processing Systems*, vol. 70, no. 1, pp. 1–19, 2013.

[12] D. Jaggar and D. Seal, *ARM architecture reference manual*. Prentice Hall, 2018.

[13] T. Instruments, "TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide, November 2006," *Literature Number: SPRU733A*.

[14] M. Jayapala, F. Barat, P. O. De Beeck, F. Catthoor, G. Deconinck, and H. Corporaal, "A low energy clustered instruction memory hierarchy for long instruction word processors," in *International Workshop on Power and Timing Modeling, Optimization and Simulation*. Springer, 2002, pp. 258–267.

[15] T. Instruments, "TMS320C62x/C67x Power Consumption Summary," 2004.

[16] R. S. Bajwa, M. Hiraki, H. Kojima, D. J. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, "Instruction buffering to reduce power in processors for signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, no. 4, pp. 417–424, Dec 1997.

[17] R. Chen, "The Itanium Processor," Microsoft, Tech. Rep., 2015.

[18] Intel, *Intel Itanium Architecture: Software Developer's Manual Volume 3: Intel Itanium Instruction Set Reference*, Intel.

[19] R. B. Lee and A. M. Fiskiran, "PLX: A fully subword-parallel instruction set architecture for fast scalable multimedia processing," in *Proceedings. IEEE International Conference on Multimedia and Expo*, vol. 2. IEEE, 2002, pp. 117–120.

[20] B. Valentine and O. Sohm, "Optimizing the JPEG2000 binary arithmetic encoder for VLIW architectures," in *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5. IEEE, 2004, pp. V–117.

[21] C. J. Hughes, "Single-instruction multiple-data execution," *Synthesis Lectures on Computer Architecture*, vol. 10, no. 1, pp. 1–121, 2015.

[22] L. Gerlach, G. Payá-Vayá, and H. Blume, "KAVUAKA: A Low Power Application Specific Hearing Aid Processor," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct 2019, pp. 99–104.

[23] G. Payá-Vayá, "Design and Analysis of a Generic VLIW Processor for Multimedia Applications," Ph.D. dissertation, Leibniz Universität Hannover, 2011.

[24] G. Payá-Vayá, J. Martín-Langerwerf, F. Giesemann, H. Blume, and P. Pirsch, "Instruction Merging to Increase Parallelism in VLIW Architectures," in *System-on-Chip, 2009. SOC 2009. International Symposium on*. IEEE, 2009, pp. 143–146.

[25] F. Giesemann, L. Gerlach, and G. Payá-Vayá, "Evolutionary Algorithms for Instruction Scheduling, Operation Merging, and Register Allocation in VLIW Compilers," *Journal of Signal Processing Systems*, 2020.

[26] L. Gerlach, G. Payá-Vayá, and H. Blume, "Efficient Emulation of Floating-Point Arithmetic on Fixed-Point SIMD Processors," in *Signal Processing Systems (SiPS), 2016 IEEE International Workshop on*. IEEE, 2016, pp. 254–259.

[27] S. K. Raina, "FLIP: a floating-point library for integer processors," Ph.D. dissertation, École Normale Supérieure de Lyon, 2006.

[28] C. Iordache and P. T. P. Tang, "An Overview of Floating-point Support and Math Library on the Intel/spl reg/XScale/spl trade/architecture," in *Proceedings 2003 16th IEEE Symposium on Computer Arithmetic*. IEEE, 2003, pp. 122–128.

[29] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[30] K. Han, J. Ahn, and K. Choi, "Power-Efficient Predication Techniques for Acceleration of Control Flow Execution on CGRA," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 2, p. 8, 2013.