Evaluating the Effect of Predication on Instruction Level Parallelism

– Student Project Report –

Sapna Bejai
Department of Computer Science
University of Kaiserslautern, Germany
http://es.cs.uni-kl.de

Abstract*—Instruction level parallelism (ILP) means that some instructions of a sequential program are carried out in parallel on multiple function units within a single microprocessor. It does not require any special consideration by the programmer and is dealt with instead by the compiler and the processor. The amount of ILP existing in programs may vary a lot and is also application-specific: In high performance computing like computer graphics and scientific computing, the amount of ILP may be very large while other applications may exhibit less ILP. However, also compiler transformations may increase the amount of ILP of a program, and in particular, predication (also called if-conversion) is known for this effect. The report investigates the increase of ILP of an algorithm that removes all acyclic control-flow statements by predication and that reiterates innermost loop bodies for a given number of times. Our motivation is the optimization of program execution by data flow processors like SCAD. In these architectures, the early availability of operands and independence of many instructions determines the ILP so that we are measuring the latter by some benchmark programs.

I. Introduction

In a parallel computation, every used processor is assigned a specific task of a part of the parallel algorithm. Depending on the parallel model of computation, each processor may have its own individual task (task-level parallelism) or all processors may be given identical tasks to be performed on their individual data (data-level parallelism). The task may be a simple operation like increment of a counter or it may be a complicated subroutine that involves many operations. The size of these tasks is expressed as the granularity of the parallelism. Granularity [7] (see Table I) is a relative measure of the ratio of the amount of computation to the amount of communication within a parallel algorithm's implementation. The grain size of a parallel task is the number of its atomic sequential instructions[1].

In this report, we are mainly interested in *instruction level parallelism* (ILP) which considers the parallel execution of instructions of a sequential program on multiple function units within a single microprocessor. It does not require any special consideration by the programmer and is dealt with instead by the compiler and the processor. ILP is exploited in processors in many ways: In particular, the most popular one is pipelining which is nowadays used in essentially all processors. A further increase of ILP has been achieved by optimizing the scheduling of the instructions of a sequential program. The two major techniques used for this are *dynamic scheduling* where the processors fetch many instructions, analyze their dependencies and schedule them *out-of-order* on the available function units to exploit ILP. In contrast, *static scheduling* uses special compiler techniques like trace scheduling and modulo scheduling to increase the ILP for processors like VLIW processors that offer many processing units but do not dynamically reschedule the given instructions. The classic static/compiler techniques used to increase ILP have been refined in many ways like superblock and hyperblock formation, techniques used in optimizing the block formation, and also many techniques to unroll or reiterate loops. Among these, predicated execution (also called if-conversion) is one outstanding technique which increases ILP by eliminating control-flow so that basic blocks of the program are merged and more independent instructions can be found for scheduling in a parallel way. While predication is generally considered to increase the amount of ILP, some authors also reported negative effects and therefore introduced *reverse if-conversion (RIC)*.

The use of ILP of a program also depends on a particular processor, and clearly on the kind and number of its processing units. In this report, we do not assume application-specific processing units, and neither do we assume special function units. Instead, all function units are considered to be universal ones, i.e., to be able to execute all kind of basic atomic instructions of the programs (like arithmetic operations etc.). Of course, we assume the processor however to be able to offer predicated execution.

In this student research report, we consider the effect of predication on the ILP of sequential programs. The remainder of this project report is organized as follows: In Section II, we consider the SCAD datapath processor architecture that we envision to make us of ILP. Section

[1]See http://home.wlu.edu/~whaleyt/classes/parallel/topics/granularity.html and [2] for a further discussion.

Granularity of Parallelism	Parallel Blocks	Architectures
Task-level parallelism (coarse grain)	Processes	Multiprocessors
Loop-level parallelism (medium grain)	Loop iterations	Multiprocessors through program Transformations
Instruction-level parallelism (fine grain)	Instructions	Superscalars and VLIWs

TABLE I: Different Types of Parallelism.

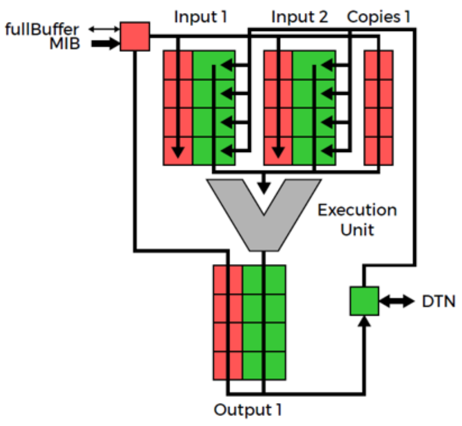
III introduces predication in general, and Section IV defines the predication technique considered in this report for improving ILP. Section V describes the experimental setup and the calculation of ILP with quantitative results. Finally, we summarize conclusions and future work in Section VI, and list in Section A the programs used as benchmark programs for the experiments.

II. Exposed Datapath Architectures

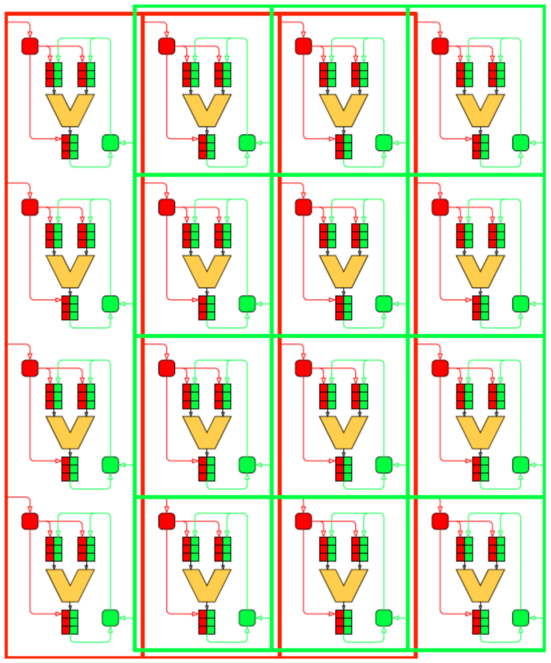
Most modern processor architectures as well as compilers which try to expose ILP in programs [10] depend on the amount of available registers to store intermediate results. Limited numbers of registers and the consequent need of load and store instructions limit the use of ILP [4]. Increasing the number of registers is however difficult because this number is directly encoded in the instruction sets. Increasing the number of registers and PUs quickly leads to a bottleneck in wiring these on the chips. Conventional processor architectures are therefore restricted in exploiting ILP due to the limited number of available registers in their instruction sets.

Exposed datapath architectures therefore expose not only the processing units, but also all datapaths between them [8], hence eliminating the use of central/global registers. Exposed datapath architectures [4] aim at applications with high parallelism and low communication between the parallel elements. These architectures provide many PUs and allow the compiler to move values directly from one PU to another one. They allow the compiler to mitigate communication delays by appropriate instruction placement which minimizes the physical distance that the data from a producer PU must travel to reach the consumer PUs. The compiler not only schedules instructions to functional units, but also takes care of directly moving values between functional units avoiding the need of registers at all. Bypassing register usage this way generally allows the compiler to improve the degree of instruction-level parallelism.

A particular exposed datapath architecture called SCAD has been developed at the university of Kaierslauern (see Figure 1). The PUs of a SCAD architecture are connected with each other by the data transport network (DTN) and by the move instruction bus (MIB) also a special control unit (CU). The inputs of PUs have FIFO buffers for input and output values so that the parallel execution of PUs does not require synchronization and PUs can execute instructions as soon as operands are available following



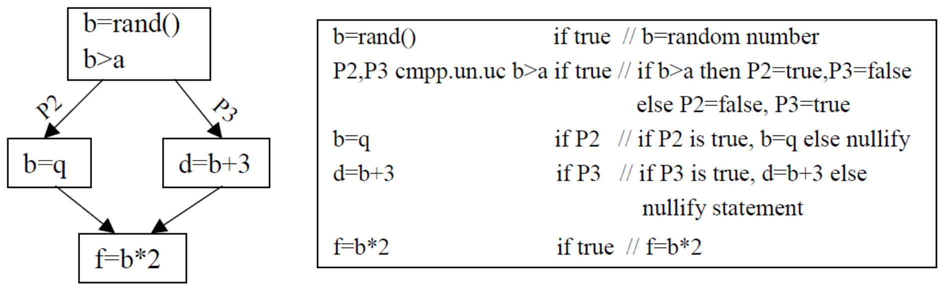
(a) SCAD Machine with a single universal PU.



(b) SCAD Machine with many PUs

Fig. 1: Architecture of a SCAD Machine

the dataflow paradigm. SCAD programs consist of move instructions that simply instruct PUs to send available results to other PUs which consume these as operands for their executions. The CU fetches the move instructions of a SCAD program in program order following the program counter, and broadcasts these on the MIB (given in red



```
b=rand()                   if true  // b=random number
P2,P3 cmpp.un.uc b>a if true // if b>a then P2=true,P3=false
                                    else P2=false, P3=true
b=q                        if P2   // if P2 is true, b=q else nullify
d=b+3                      if P3   // if P3 is true, d=b+3 else
                                    nullify statement
f=b*2                      if true  // f=b*2
```

a) Original Control Flow Graph b) Predicated Hyperblock

Fig. 2: Example showing the transformation into predicated conditional-free dataflow format [6]

color in Figure 1). The producer and consumer PUs then notify these addresses so that the values can be moved later on when these are available. The data moves are done via the DTN (given in green color in Figure 1) which is used by the PUs to asynchronously send values to each other whenever these are available. SCAD machines therefore execute a sequential program consisting of *move* instructions whose effect is to transport a value from the head of an output queue of a processing unit to the tail of an input queue of the same or another processing unit.

The breadth-first traversal on expression trees by compilers when generating code for the SCAD machine ensures that the operands are found in the correct order in the buffers and also ensures that there is no need for an additional memory. The experimental results of [3], [5] not only demonstrate the superiority of this special code generation compared to the traditional register based code generation, but also show that this compilation enables exposed datapath architectures to exploit concurrency in programs to the fullest [5]. The classic depth-first traversal was influenced by the reuse of registers, while the classic queue machines directed how to exploit the maximum ILP via the breadth-first approach *eliminated the use of registers completely*.

SCAD machines resemble queue machines but in contrast to queue machines, they have more than one queue and may have also many PUs. The code generation for SCAD machines is therefore different to queue machines [3], [4], [5] and has been recently proved to be NP-complete [1].

III. Predicated Execution

A major obstacle for using ILP in the SCAD machine are control-flow dependencies of instructions since the current SCAD machines do not make use of branch prediction. This is typical for statically scheduled processors in contrast to dynamically scheduled architectures that break the control-flow barriers by branch prediction.

The essential low-level control-flow statements are conditional and unconditional branches, i.e., if-then-else statements and goto statements. Using these basic control-flow statements other high-level statements can be defined like loops. Conversely, compilers decompose high-level statements to low-level statements of intermediate languages where control-flow statements are also conditional and unconditional branches. In this report, we consider sequential programs given in such a typical intermediate language.

Each conditional branch contains a target instructions to which control flows if the branch condition evaluates to true or a false. In control-flow graphs (CFGs), this leads to different basic blocks that are obstacles for ILP. Predicated execution creates one hyperblock of such typical if-then-else structures as shown in Figure 2.

Predication, i.e., if-conversion, therefore replaces a set of basic blocks containing conditional control flow between the blocks with a single block of predicated instructions, thus transforming all control dependencies into data dependencies [9]. This has the advantages that the 'then' and 'else' parts are scheduled and executed simultaneously by the processor ignoring the instructions that were not supposed to be run [6]. The advantage is that even though some instruction are scheduled that are not executed at all, more instruction can be scheduled in parallel to increase the amount of ILP.

IV. Algorithm

For our experiments, we use the experimental compiler used for teaching purposes of the embedded systems group of the computer science department of the university of Kaiserslautern. Inputs are MiniC programs that consist of the following statements where λ, λ_1, and λ_2 are left-hand side expressions, τ is a right-hand side expression:

- $\lambda = \tau$;
- $\lambda_1, \lambda_2 = \tau$;
- $S_1\ S_2$
- **if**$(\sigma)\ S$
- **if**$(\sigma)\ S_1$ **else** S_2
- **while**$(\sigma)\ S$

- **do** S **while**(σ)
- **for**($i = m..n$) S
- **return** τ
- **sync**

Available data types are **bool**, **nat**, **int** as atomic data types, and `[n]`α for arrays on type α and $\alpha*\beta$ as product, i.e., pairs, of types α and β. MiniC programs are furthermore structured into threads and functions, but we only consider a single thread for our experiments.

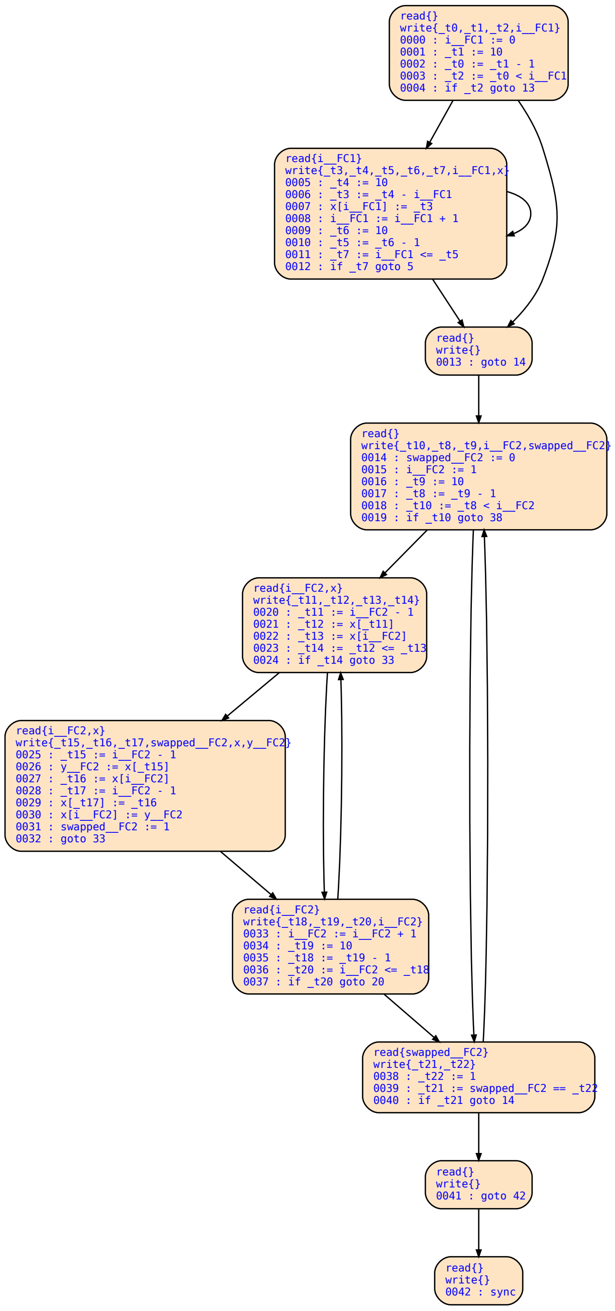
```
procedure BubbleSort([]nat x,nat xlen) {
   nat i,y,swapped;
   do {
      swapped = 0;
      for(i=1..xlen−1) {
         // if pair x[i−1],x[i] is in the wrong order
         if(x[i−1] > x[i]) {
            // then swap it and remember the change
            y = x[i−1];
            x[i−1] = x[i];
            x[i] = y;
            swapped = 1;
         }
      }
   } while(swapped==1)
   return;
}


// create a test array in reverse order
procedure Initialize([]nat x,nat xlen) {
   nat i;
   for(i=0..xlen−1)
      x[i] = xlen−i;
   return;
}


thread t {
   [10]nat x;
   Initialize(x,10);
   BubbleSort(x,10);
}
```

Fig. 3: MiniC Program BubbleSort (Before Predication)

For example, Figure 3 shows the well-known bubblesort algorithm for sorting an array `x` of 10 natural numbers. The program defines two procedures, one for generating an unsorted array `[9,...,0]` and the other one for sorting this sequence with the bubblesort algorithm. Figure 4 shows its control-flow graph (CFG) consisting of the basic blocks of the program. As can be seen, there is one loop that stems from procedure `Initialize` and two nested loops that stem from procedure `BubbleSort`.

As can be observed – and this is the usual observation – the program yields a control-flow graph with many small basic blocks so that compilers cannot make use of much ILP. For this reason, compilers have to merge basic blocks to larger ones using different techniques like trace scheduling, modulo scheduling and hyperblock formation, i.e., predication. As already explained, we want to measure the effect of the increase of ILP using predication on algorithms.

Our algorithm first reads the MiniC file and inlines all function calls. It then reiterates loops according to the



```
read{}
write{_t0,_t1,_t2,i__FC1}
0000 : i__FC1 := 0
0001 : _t1 := 10
0002 : _t0 := _t1 - 1
0003 : _t2 := _t0 < i__FC1
0004 : if _t2 goto 13
```

```
read{i__FC1}
write{_t3,_t4,_t5,_t6,_t7,i__FC1,x}
0005 : _t4 := 10
0006 : _t3 := _t4 - i__FC1
0007 : x[i__FC1] := _t3
0008 : i__FC1 := i__FC1 + 1
0009 : _t6 := 10
0010 : _t5 := _t6 - 1
0011 : _t7 := i__FC1 <= _t5
0012 : if _t7 goto 5
```

```
read{}
write{}
0013 : goto 14
```

```
read{}
write{_t10,_t8,_t9,i__FC2,swapped__FC2}
0014 : swapped__FC2 := 0
0015 : i__FC2 := 1
0016 : _t9 := 10
0017 : _t8 := _t9 - 1
0018 : _t10 := _t8 < i__FC2
0019 : if _t10 goto 38
```

```
read{i__FC2,x}
write{_t11,_t12,_t13,_t14}
0020 : _t11 := i__FC2 - 1
0021 : _t12 := x[_t11]
0022 : _t13 := x[i__FC2]
0023 : _t14 := _t12 <= _t13
0024 : if _t14 goto 33
```

```
read{i__FC2,x}
write{_t15,_t16,_t17,swapped__FC2,x,y__FC2}
0025 : _t15 := i__FC2 - 1
0026 : y__FC2 := x[_t15]
0027 : _t16 := x[i__FC2]
0028 : _t17 := i__FC2 - 1
0029 : x[_t17] := _t16
0030 : x[i__FC2] := y__FC2
0031 : swapped__FC2 := 1
0032 : goto 33
```

```
read{i__FC2}
write{_t18,_t19,_t20,i__FC2}
0033 : i__FC2 := i__FC2 + 1
0034 : _t19 := 10
0035 : _t18 := _t19 - 1
0036 : _t20 := i__FC2 <= _t18
0037 : if _t20 goto 20
```

```
read{swapped__FC2}
write{_t21,_t22}
0038 : _t22 := 1
0039 : _t21 := swapped__FC2 == _t22
0040 : if _t21 goto 14
```

```
read{}
write{}
0041 : goto 42
```

```
read{}
write{}
0042 : sync
```

Fig. 4: Control-flow Graph (CFG) of BubbleSort Before Predication

following code transformations if the loop bodies do not contain other loops:

- **while**(σ) S $\equiv$ **while**(σ) $\{S$; **if**(σ) $S\}$
- **do** S **while**(σ) $\equiv$ **do** $\{S$; **if**(σ) $S\}$ **while**(σ)
- **for**(i=$m..n$) S
  $\equiv$ { **nat** i; i=m; **while**(i<=n)**do** S; i=i+1; }

The above rules may be applied for a given number of times ℓ, thus creating ℓ copies of an innermost loop body.

After this, acyclic code regions are predicated, i.e., all instructions are given a predicate condition that must

```
thread t {
 bool c0;
 [10]nat x;
 // inlined code of Initialize
 nat i0;
 for(i0=0..9)
   x[i0] = 10−i0;
 // inlined code of BubbleSort
 nat i1,y1,swapped;
 do {
   swapped = 0;
   for(i1=1..9) {
     c0 = x[i1] < x[i1−1];
     y1 = (c0 ? x[i1−1] : y1);
     x[i1−1] = (c0 ? x[i1] : x[i1−1]);
     x[i1] = (c0 ? y1 : x[i1]);
     swapped = (c0 ? 1 : swapped);
   }
 } while(swapped)
}
```

Fig. 5: Predicated MiniC Program BubbleSort Without Unrolling of Loops

hold for the execution of the instruction. To this end, a single pass over the program statements are made where a preliminary predicate condition is updated whenever an if-statement is traversed. The procedure is essentially defined as follows using as intial predicate condition $\varphi = \texttt{true}$:

- $\mathsf{Pred}(\varphi, \lambda = \tau) := \lambda \overset{\varphi}{=} \tau$
- $\mathsf{Pred}(\varphi, \lambda_1, \lambda_2 = \tau) := \lambda_1, \lambda_2 \overset{\varphi}{=} \tau$
- $\mathsf{Pred}(\varphi, S_1\ S_2) := \mathsf{Pred}(\varphi, S_1)\ \mathsf{Pred}(\varphi, S_2)$
- $\mathsf{Pred}(\varphi, \texttt{if}(\sigma)\ S) := \mathsf{Pred}(\varphi \wedge \sigma, S)$
- $\mathsf{Pred}(\varphi, \texttt{if}(\sigma)\ S_1\ \texttt{else}\ S_2)$
  $:= \mathsf{Pred}(\varphi \wedge \sigma, S_1)\ \mathsf{Pred}(\varphi \wedge \neg\sigma, S_2)$
- $\mathsf{Pred}(\varphi, \texttt{while}(\sigma)\ S)$
  $:= \texttt{if}(\varphi)\ \texttt{while}(\sigma)\ \mathsf{Pred}(\texttt{true}, S)$
- $\mathsf{Pred}(\varphi, \texttt{do}\ S\ \texttt{while}(\sigma))$
  $:= \texttt{if}(\varphi)\ \texttt{do}\ \mathsf{Pred}(\texttt{true}, S)\ \texttt{while}(\sigma)$

Statement $\lambda \overset{\varphi}{=} \tau$ denotes thereby a predicated assignment, i.e., the assignment $\lambda = \tau$ is executed only if φ holds (and otherwise, nothing is executed).

After applying the predication algorithm, the predicated MiniC code of the bubblesort MiniC program shown in Figure 3 looks as shown in Figure 5 and Figure 6, respectively, if innermost loop bodies are not unrolled at all or unrolled once, respectively.

In both cases, the structure of basic blocks concerning loops is kept, but if-then-else parts are merged into a single basic block that contains predicated assignments. In particular, innermost loop bodies will now only consist of a single basic block so that unrolling this basic block before predication will generate also a single basic block of twice the size where now two of the previous loop bodies can be scheduled together. Hence, after predication, the only basic blocks that remain are due to loops.

V. Experimental Evaluation

To measure the effect on the available ILP by our predication algorithm, we first compute the CFG of the original

```
thread t{
 bool c0,c1,c2,c3,c4,c5,c6,c7,c8;
 [10]nat x;
 // inlined code of Initialize
 nat i;
 i = 0;
 while(i<=9) {
   x[i] = 10−i;
   i = i+1;
   c0 = i<=9;
   c1 = !c0;
   x[i] = (c0 ? (10−i) : x[i]);
   i = (c0 ? (i+1) : i);
 }
 // inlined code of BubbleSort
 nat y1,swapped;
 do {
   swapped = 0;
   nat i1;
   i1 = 1;
   while(i1<=9) {
     c2 = (x[i1] < x[i1−1]);
     c3 = !c2;
     y1 = (c2 ? x[i1−1] : y1);
     x[i1−1] = (c2 ? x[i1] : x[i1−1]);
     x[i1] = (c2 ? y1 : x[i1]);
     swapped = (c2 ? 1 : swapped);
     i1 = i1+1;
     c4 = i1<=9;
     c5 = !c4;
     c6 = (x[i1] < x[i1−1]);
     c7 = c4 & c6;
     c8 = c4 & !c6;
     y1 = (c7 ? x[i1−1] : y1);
     x[i1−1] = (c7 ? x[i1] : x[i1−1]);
     x[i1] = (c7 ? y1 : x[i1]);
     swapped = (c7 ? 1 : swapped);
     i1 = (c4 ? (i1+1) : i1);
     }
   } while(swapped)
}
```

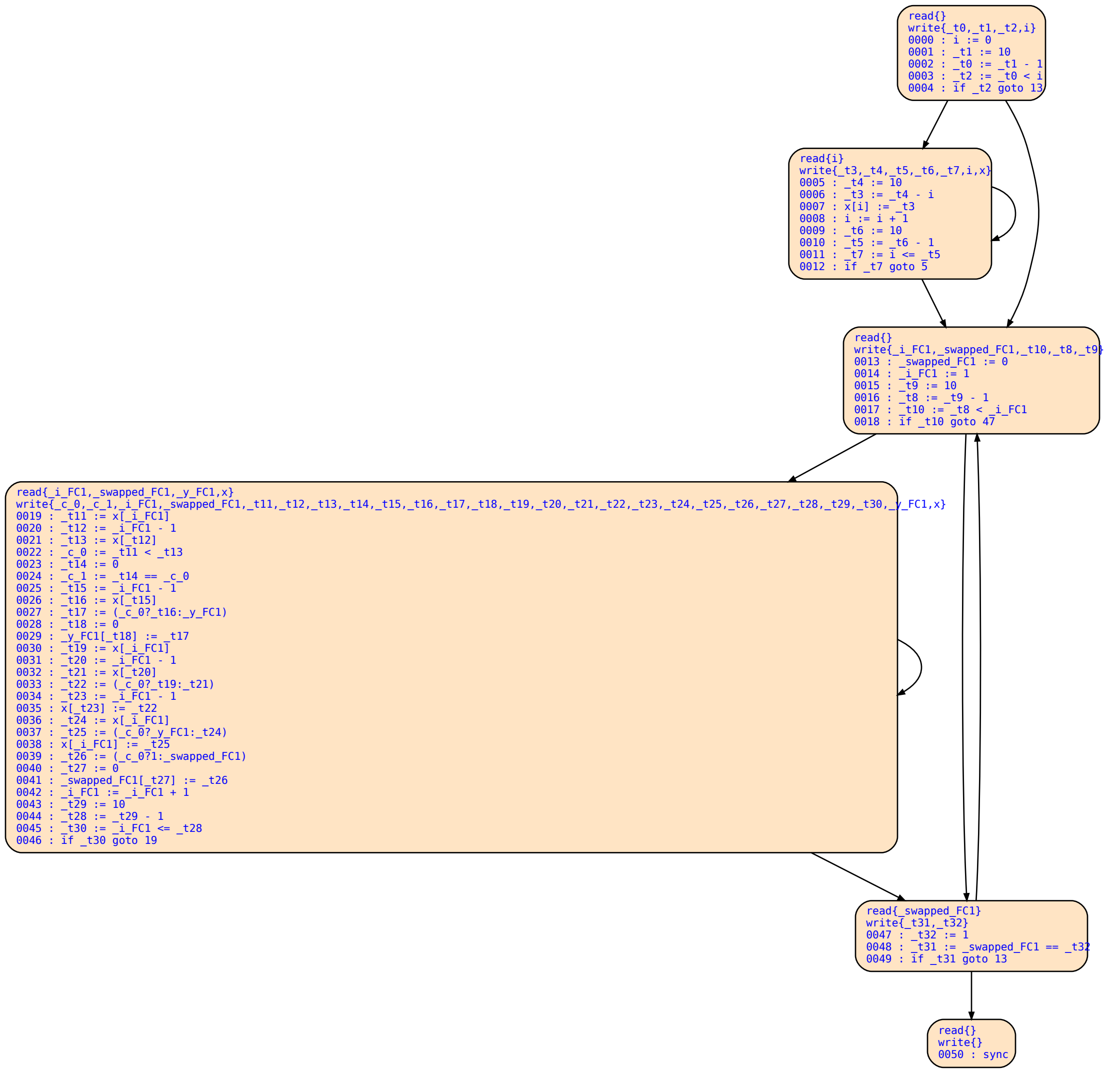
Fig. 6: Predicated MiniC Program BubbleSort With Unrolling Innermost Loop Bodies Once

and the predicated program and then compare their respective ILPs. To measure the ILP, we compute the data dependency graphs for each basic block and schedule the instructions by an ASAP (as-soon-as-possible) schedule. The number of instructions divided by length of the longest path of these data dependency graphs is then used as measure of ILP since it means how many instructions of the basic block can be executed in average in parallel of the basic block. We compare this ILP before and after predication and, in particular, we consider these numbers for the innermost loop bodies that usually contribute most to the runtime of a program.

In particular, we first obtain the CFG from the MiniC compiler[2], and calculate the ILP for the entire program. After this, we generate the predicated MiniC program using our algorithm and calculate its ILP. A reduced number of blocks and maximum ILP within a block indicate that our algorithm has been successful at providing an improvised overall ILP.

Table II shows the analysis for program `BubbleSort`: While the original CFG shown in Figure 4 consists of

[2]See https://es.informatik.uni-kl.de/tools/teaching/MiniC.html



```
read{}
write{_t0,_t1,_t2,i}
0000 : i := 0
0001 : _t1 := 10
0002 : _t0 := _t1 - 1
0003 : _t2 := _t0 < i
0004 : if _t2 goto 13
```

```
read{i}
write{_t3,_t4,_t5,_t6,_t7,i,x}
0005 : _t4 := 10
0006 : _t3 := _t4 - i
0007 : x[i] := _t3
0008 : i := i + 1
0009 : _t6 := 10
0010 : _t5 := _t6 - 1
0011 : _t7 := i <= _t5
0012 : if _t7 goto 5
```

```
read{}
write{_i_FC1,_swapped_FC1,_t10,_t8,_t9}
0013 : _swapped_FC1 := 0
0014 : _i_FC1 := 1
0015 : _t9 := 10
0016 : _t8 := _t9 - 1
0017 : _t10 := _t8 < _i_FC1
0018 : if _t10 goto 47
```

```
read{_i_FC1,_swapped_FC1,_y_FC1,x}
write{_c_0,_c_1,_i_FC1,_swapped_FC1,_t11,_t12,_t13,_t14,_t15,_t16,_t17,_t18,_t19,_t20,_t21,_t22,_t23,_t24,_t25,_t26,_t27,_t28,_t29,_t30,_y_FC1,x}
0019 : _t11 := x[_i_FC1]
0020 : _t12 := _i_FC1 - 1
0021 : _t13 := x[_t12]
0022 : _c_0 := _t11 < _t13
0023 : _t14 := 0
0024 : _c_1 := _t14 == _c_0
0025 : _t15 := _i_FC1 - 1
0026 : _t16 := x[_t15]
0027 : _t17 := (_c_0?_t16:_y_FC1)
0028 : _t18 := 0
0029 : _y_FC1[_t18] := _t17
0030 : _t19 := x[_i_FC1]
0031 : _t20 := _i_FC1 - 1
0032 : _t21 := x[_t20]
0033 : _t22 := (_c_0?_t19:_t21)
0034 : _t23 := _i_FC1 - 1
0035 : x[_t23] := _t22
0036 : _t24 := x[_i_FC1]
0037 : _t25 := (_c_0?_y_FC1:_t24)
0038 : x[_i_FC1] := _t25
0039 : _t26 := (_c_0?1:_swapped_FC1)
0040 : _t27 := 0
0041 : _swapped_FC1[_t27] := _t26
0042 : _i_FC1 := _i_FC1 + 1
0043 : _t29 := 10
0044 : _t28 := _t29 - 1
0045 : _t30 := _i_FC1 <= _t28
0046 : if _t30 goto 19
```

```
read{_swapped_FC1}
write{_t31,_t32}
0047 : _t32 := 1
0048 : _t31 := _swapped_FC1 == _t32
0049 : if _t31 goto 13
```

```
read{}
write{}
0050 : sync
```

Fig. 7: Control-flow Graph of BubbleSort After Predication

Bubble sort Blocks		
Block Number	Instruction count Before Pred (10 Blocks)	Instruction count After Pred (6 blocks)
Block 1	00 to 04	00 to 04
Block 2	05 to 12	05 to 12
Block 3	13	13 to 18
Block 4	14 to 19	19 to 46
Block 5	20 to 24	47 to 49
Block 6	25 to 32	50
Block 7	33 to 37	
Block 8	38 to 40	
Block 9	41	
Block 10	42	

TABLE II: Increase of ILP for BubbleSort.

ten basic blocks, the CFG of the predicated version has only six basic blocks. The innermost loop body of the unpredicated version consists of three basic blocks that are merged into a single one in the predicated version. The maximal length of a basic block is 8 in the unpredicated version, while it is 28 in the predicated version. In particular, the length of the innermost loop bodies basic block has this length and it has therefore now good chances for ILP.

To measure the amount of ILP per basic block, we consider the data dependency graphs of each basic block, and calculate the length of the longest path as the number of cycles required to execute the basic block. Table III shows these calculations for the CFGs of the `BubbleSort` program. For each basic block of the unpredicated and the predicated CFG, we use the ASAP schedules where instructions are scheduled to cycles, and calculate for each basic block the number of instructions divided by the number of cycles to execute them, i.e., the average

ILP calculation of Bubble Sort		
Blocks	**Instructions Before Predicate**	**Instructions After Predicate**
1	Cycle 1=00,01 Cycle 2=02 Cycle 3=03 Cycle 4=04 $\text{ILP} = \frac{5 \text{ instr}}{4 \text{ cycles}} = 1.25$	Cycle 1=00,01 Cycle 2=02 Cycle 3=03 Cycle 4=04 $\text{ILP} = \frac{5 \text{ instr}}{4 \text{ cycles}} = 1.25$
2	Cycle 1=05, 08,09 Cycle 2=06,10 Cycle 3=07,11 Cycle 4=12 $\text{ILP} = \frac{8 \text{ instr}}{4 \text{ cycles}} = 2$	Cycle 1=05,09 Cycle 2=06,10 Cycle 3=08,07,11 Cycle 4=12 $\text{ILP} = \frac{8 \text{ instr}}{4 \text{ cycles}} = 2$
3	Cycle 1=13 $\text{ILP} = \frac{1 \text{ instr}}{1 \text{ cycles}} = 1$	Cycle 1=13,14,15 Cycle 2=16 Cycle 3=17 Cycle 4=18 $\text{ILP} = \frac{6 \text{ instr}}{4 \text{ cycles}} = 1.5$
4	Cycle 1=14,15,16 Cycle 2=17 Cycle 3=18 Cycle 4=19 $\text{ILP} = \frac{6 \text{ instr}}{4 \text{ cycles}} = 1.5$	Cycle 1=19,20,23,25,28,31,30,40,43,34,36 Cycle 2=32,21,26,24,44 Cycle 3=22,27,33 Cycle 4=39,29,35 Cycle 5=41,37 Cycle 6=38 Cycle 7=42 Cycle 8=45 Cycle 9=46 $\text{ILP} = \frac{28 \text{ instr}}{9 \text{ cycles}} = 3.11$
5	Cycle 1= 20,22 Cycle 2=21 Cycle 3=23 Cycle 4=24 $\text{ILP} = \frac{5 \text{ instr}}{4 \text{ cycles}} = 1.25$	Cycle 1= 47 Cycle 2=48 Cycle 3=49 $\text{ILP} = \frac{3 \text{ instr}}{3 \text{ cycles}} = 1$
6	Cycle 1= 25,27,28,31,32 Cycle 2=26,29 Cycle 3=30 $\text{ILP} = \frac{8 \text{ instr}}{3 \text{ cycles}} = 2.66$	Cycle 1=50 $\text{ILP} = \frac{1 \text{ instr}}{1 \text{ cycles}} = 1$
7	Cycle 1= 33,34 Cycle 2=35 Cycle 3=36 Cycle 4=37 $\text{ILP} = \frac{5 \text{ instr}}{4 \text{ cycles}} = 1.25$	
8	Cycle 1= 38 Cycle 2=39 Cycle 3=40 $\text{ILP} = \frac{3 \text{ instr}}{3 \text{ cycles}} = 1$	
9	Cycle 1= 41 $\text{ILP} = \frac{1 \text{ instr}}{1 \text{ cycles}} = 1$	
10	Cycle 1= 42 $\text{ILP} = \frac{1 \text{ instr}}{1 \text{ cycles}} = 1$	

TABLE III: Calculation of ILP

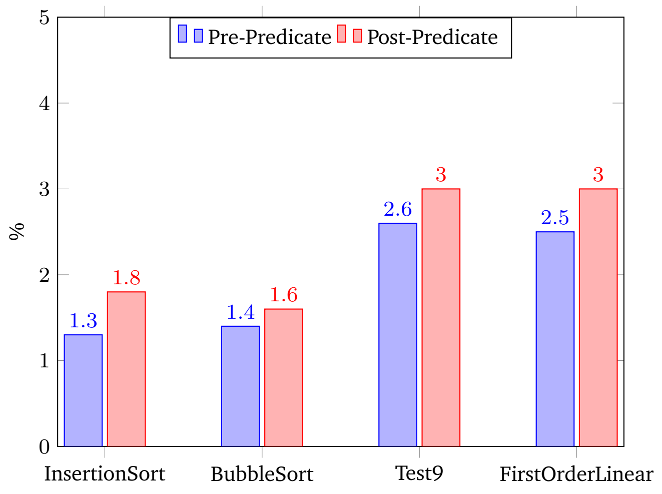


Fig. 8: Relative Increase of ILP of Benchmark Programs.

number of instructions executed per cycle of the basic block. Hence, we define for each basic block

$$\text{ILP}_{block} = \frac{\text{number of Instructions}}{\text{number of cycles}}$$

The arithmetic means of all the ILPs per block in the entire program gives the ILP of the execution of the program (we assume here that all blocks are executed equally often which is however not realistic).

$$\text{ILP}_{prog} = \frac{1}{n}\sum_{i=1}^{n} \text{ILP}_i,$$

where n is the number of blocks of the program, and ILP_i is the ILP of basic block i. For program `BubbleSort`, we obtain before predication $\text{ILP}_{prog} = \frac{13.91}{10} = 1.39$ and after predication $\text{ILP}_{prog} = \frac{9.86}{6} = 1.64$.

We performed the same calculations on programs catering to various types of branching constructs (see appendix). Figure 8 presents the results in terms of percentage increase in the ILP. These programs contained if-then-else, function calls constructs, looping constructs. We can summarize that there has been significant increase in the performance from the visualization obtained by plotting the results.

VI. Conclusion and Future Work

Based on the results obtained from the experiments, we observe a significant improvement of the amount of ILP by predicating the programs. Further investigation is going on about whether a programs' executions is dominated by non-loop branches or loop branches. Loop unrolling dynamically also leading to increased code size due to code expansion is also to be considered as major game player. If the loop unrolling takes place by a bigger number, then the other blocks in the program execution can be discarded, but if the loop is unrolled for a very small number with respect to the larger number of blocks, then we need to discuss about the further steps.

The predicate manipulation phase termed as 'partial reverse if-conversion' i.e; adjustment of hyperblocks during scheduling, is proposed by [9] is also the future discussion activity of this project. To address the problem of false dependencies, [6] has proposed a predicate-sensitive implementation of SSA called Predicated Static Single Assignment (PSSA) which we would also discuss this in the next phase. PSSA seeks to accomplish the same objectives as SSA for a predicated hyperblock. As a limitation, we can state that single static assignment depends on the type of the program. If the program instructions involve various variables, then the SSA can be explored to the most.

References

[1] M. Anders. Complexity analysis of code generation for the SCAD machine. Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany, October 2017. Bachelor.

[2] S. Bejai. Compilation techniques for increasing instruction-level parallelism for the scad machine, 2018. Seminar on Embedded Systems, Embedded Systems Group, Department of Computer Science, University of Kaiserslautern, Germany.

[3] A. Bhagyanath, T. Jain, and K. Schneider. Towards code generation for the synchronous control asynchronous dataflow (SCAD) architectures. In R. Wimmer, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 77–88, Freiburg, Germany, 2016. University of Freiburg.

[4] A. Bhagyanath and K. Schneider. Optimal compilation for exposed datapath architectures with buffered processing units by SAT solvers. In E. Leonard and K. Schneider, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 143–152, Kanpur, India, 2016. IEEE Computer Society.

[5] A. Bhagyanath and K. Schneider. Exploring different execution paradigms in exposed datapath architectures with buffered processing units. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2017.

[6] L. C. .et al. Path analysis and renaming for predicated instruction scheduling. International Journal of Parallel Programming, 2000. University of California, San Diego.

[7] N. Hottle. Granularity in parallel algorithms, 1992. http://home.wlu.edu/~whaleyt/classes/parallel/topics/granularity.html.

[8] E. Jedermann. Exposed datapath processor architecture implementation survey, 2015. Seminar on Embedded Systems, Embedded Systems Group, Department of Computer Science, University of Kaiserslautern, Germany.

[9] S. A. Mahlke, D. I. August, and W. mei W. Hwu. The partial reverse if-conversion framework for balancing control flow and predication. Center for Reliable and High-Performance Computing, Hewlett-Packard Laboratories, May 1999.

[10] M. Schlansker, T. Conte, J. Dehnert, K. Ebcioglu, J. Fang, and C. Thompson. Compilers for instruction-level parallelism. *IEEE Computer*, 30(12):63–69, December 1997.

Appendix

```
function vectorLength (nat x1,x2,x3,x4) : nat {
   nat y1,y2,y3,y4;
   y1 = x1 * x1;
   y2 = x2 * x2;
   y3 = x3 * x3;
   y4 = x4 * x4;
   y1 = y1 + y2;
   y3 = y3 + y4;
   return(y1 + y3);
}
thread test {
   nat x1,x2,x3,x4,x5,x6,x7,x8,y;
   x1 = 1;
   x2 = 2;
   x3 = 3;
   x4 = 4;
   x5 = 5;
   x6 = 6;
   x7 = 7;
   x8 = 8;
   y = vectorLength(x1,x2,x3,x4) + →
        vectorLength(x5,x6,x7,x8);
}
```

Fig. 9: MiniC Program for Computing Length of Vectors (Before Predication)

```
function fibonacci(nat n) : nat {
   nat i,f1,f2,fn;
   if(n==1 | n==2) {
     fn = 1;
   } else {
     f1 = 1;
     f2 = 1;
     i = 2;
     while(i<n) {
       i = i+1;
       fn = f1+f2;
       f1 = f2;
       f2 = fn;
     }
   }
   return fn;
}

thread Fibonacci {
   nat n;
   n = fibonacci(10);
}
```

Fig. 10: MiniC Program Fibonacci (Before Predication)

```
[8]int y;

function exp(nat x0, nat y0):nat {
   nat h,i;
   h = 1;

   if(x0!=0) {
      for(i=1..y0) {
         h = h∗x0;
      }
   }

   return h;
}

function FirstOrderLinearRec(int y0, []int a, []int b) : →
      nat {
   nat n,i,j,k;
   [8][2]int m;
   n = 10;
   for(i=0..n−1){
      m[i][0] = a[i];
      m[i][1] = b[i];
   }
   for(i=0..2){ //log 2 8 = 3
      k = exp(2,i);
      for(j=k..n−1){
         m[j][0] = m[j][0] ∗ m[j−k][0];
         m[j][1] = m[j][0] ∗ m[j−k][1] + m[j][1];
      }
   }
   //compute y[i]
   for(i=0..n−1) {
      y[i]= m[i][0] ∗ y0 + m[i][1];
   }
   return 1;
}

thread test{
   [8]int a;
   [8]int b;
   nat dummy;
   a[0] = +1;
   a[1] = +2;
   a[2] = −2;
   a[3] = −1;
   a[4] = +3;
   a[5] = −3;
   a[6] = +0;
   a[7] = +1;

   b[0] = +3 ;
   b[1] = +1;
   b[2] = +0;
   b[3] = +1;
   b[4] = −2;
   b[5] = −1;
   b[6] = +3;
   b[7] = −4;

   dummy = FirstOrderLinearRec(+1,a,b);
}
```

Fig. 11: MiniC Program First Order Linear Recursive (Before Predication)

```
thread InsertionSort {
   [10]nat x;
   nat i,j,y;
   // first create a test array in reverse order
   for(i=0..9) {
      x[i] = 10−i;
   }
   // now apply insertion sort
   for(i=1..9) {
      y = x[i]; // for element x[i], find its place in
      j = i−1; // already sorted list x[0..i−1]
      while(j>0 & x[j]>y) {
         x[j+1] = x[j];
         j = j−1;
      }
      // now j==0 or x[j]<=y
      if(x[j]>y) {
         // note that this implies j==0
         x[j+1] = x[j];
         x[j] = y;
      } else {
         // here we have x[j]<=y, so that y must be →
              placed at j+1
         x[j+1] = y;
      }
   }
}
```

Fig. 12: MiniC Program InsertionSort (Before Predication)

```
procedure Initialize ([]nat a,b,nat n) {
   nat i;
   for(i=0..n−1) {
         a[i] = i+1;
         b[i] = i+1;
   }
   return;
}

function InnerProduct([]nat a,b,nat n) : nat {
   nat i,y;
   for(i=0..n−1)
      y = y + a[i]∗b[i];
   return y;
}

thread main {
   [8]nat a,b;
   nat y;
   // first create some test matrices a and b
   Initialize(a,b,8);
   // now call one of the above procedures
   y = InnerProduct(a,b,8);
}
```

Fig. 13: MiniC Program InnerProduct (Before Predication)

```
function sumUp(nat n) : nat {
   nat i,sum;
   i = 1;
   sum = 0;
   while(i <= n) {
     sum = sum + i;
     i = i + 1;
   }
   return sum;
}

thread SumUp {
   nat sum;
   sum = sumUp(10);
}
```

Fig. 14: MiniC Program SumUp (Before Predication)