Improving Instruction Level Parallelism Using Predication and Loop Optimization

– 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. Parallelization refers to issuing, scheduling and executing multiple instructions in the same clock cycle to exploit ILP. To reach this goal, machines must be equipped with parallel datapaths for concurrent execution of instructions, alongside the compilers that very necessarily must expose parallelism to such hardware. 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. Compiler transformations may increase the amount of ILP of a program, and in particular, predication (also called if-Conversion) is known for this effect. With the advent of modern processor architectures, loops are considered as good candidates for compiler optimization to extract higher performance. Loop unrolling as the name suggests, is a technique of duplicating iterations a specific number of times to avoid branch and jump overhead; thus resulting in higher ILP and better performance of programs with the side effect of increasing the size of the basic blocks. There has not been a satisfactory answer so far on how often and when to unroll efficiently. Loop flattening is a type of software pipelining which transforms any nested and sequenced loops into a single loop. The report investigates the increase of ILP of an algorithm that removes all acyclic control-flow statements by predication and experiments on loop optimization methods. Our motivation is the optimization of program execution by data flow processors like SCAD.

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). Machine level parallelism and instruction level parallelism (ILP) are relatively similar. The former refers to the number of simultaneous datapaths the architecture consists of whereas the latter refers to the amount of parallelism the compiler exposes to the architecture by means of various techniques [7].

From my earlier student project [2], I have already learned that ILP is better handled by the compiler and the processor with a detailed explanation of how an exposed datapath architecture aims at high parallelism and low communication between the parallel elements supported by various dynamic and static scheduling techniques. Among these techniques, predicated execution 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. With this predicated approach, we clearly saw an increase in the ILP as compared to the execution of actual MiniC benchmark programs. In addition to this, we also proposed the technique for unrolling the loop.

In this student research project report, my study considers now loop unrolling, in particular, the impact of the number of unrollings, called the unroll factor in the following, on the amount of ILP of the program. We implemented an approach to perform loop unrolling irrespective of the number of loop iterations in the program. We would be performing new empirical experiments with loop unrolling and also theoretically study about the benefits of flattening sequences of nested loops along with predication.

The remainder of this project report is organized as follows: Section II is divided into 2 parts: The first part highlights our earlier project with the implementation of predication, and the second part gives a quick tour on the literature introducing loop unrolling and loop flattening. Section III explains loop unrolling implemented as an enhancement over the earlier predication algorithm defined in my earlier student project [2]. Section IV describes the experimental setup and the calculation of ILP with quantitative results. Here, we evaluate the effect of loop unrolling with various unroll factors, and we also perform an empirical study on how the ILP varies for different categories of loop combinations with different unroll factors applied on the MiniC programs. Finally, we summarize conclusions and future work in Section V, and

list the benchmark programs in Section A that are used for our experiments.

II. Background and Related Work

A. Predication algorithm

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. SCAD machines do not make use of branch prediction to overcome control-flow dependencies of instructions. This is typical for statically scheduled processors in contrast to dynamically scheduled architectures that break the control-flow barriers by branch prediction. To eliminate control dependencies, we followed the approach of predicated execution by merging many basic blocks into one large hyperblock. Our algorithm first reads the MiniC file and inlines all function calls. It then reiterates loops according to the code transformations if the loop bodies do not contain other loops. After predication, the only basic blocks that remain are due to loops. Acyclic code regions are predicated, i.e., all instructions are given a predicate condition that must hold for the execution of the instruction [2]. To this end, a single pass over the program statement is made where a preliminary predicate condition is updated whenever an if-statement is traversed. If-then-else parts are merged into a single basic block that contains predicated assignments, thus transforming all control dependencies into data dependencies [1]. This has the advantage that the 'then' and 'else' parts are scheduled and executed simultaneously by the processor ignoring the instructions that were not supposed to be run [4]. The advantage is that even though some instructions are scheduled that are not executed at all, more instructions can be scheduled in parallel to increase the amount of ILP.

B. Related Work

1) Loop unrolling: is expected to be a relatively inexpensive compiler optimization technique. The number of times by which the loop body will be duplicated plays a major role in deciding the effectiveness of the unrolling technique. It can lead to better performance by creating many copies of the loop body, thus eliminating the occurrence of loop indexing overhead and conditional branching. As a consequence, the increase of the block size results in a higher probability of parallel execution of instructions. In contrast, if unrolled inappropriately, it may contribute to drastic increase in code size sometimes leading to bad ILP.

From the literature study [5], outer loop unrolling is often questionable in that if often has no improvement. All innermost loops are actual candidates for unrolling. As we see from Figure 1 and Figure 2, an extra loop body is added, either at the end (epilogue) or at the beginning (prologue) of the unrolled body. The unroll factor is related to the loop iteration count. If the loop is unrolled n times, and the iteration count is not a multiple of the loop unroll factor, then additionally to the unrolled loop, a prologue or epilogue to the loop is added which executes the first few or the last few iterations of the original loop. For example, when the loop iteration count is 10 and the unroll factor is 4, then two bodies of the rolled loop are inserted as prologue or epilogue.

The disadvantage of this approach is the additional loop overhead which is observed when:

- an additional conditional branch checks if the epilogue or the prologue code is executed,
- extra instructions to do the calculation based on the iteration count (division in first figure and subtraction in second figure) of the unrolled loop.

A modulo or division operation is expensive which can also be replaced by cheaper shift operations if the unroll factor is a power of two, and the loop bounds are of unsigned type. Subtraction is relatively a little inexpensive operation, thus placing the extra code as epilogue is better. However, in general, a division operation or a subtraction is an extra calculation. If the unrolled loop is either nested inside another loop or is inside a function being called frequently, then there is lot of overhead.

2) Loop flattening methods:: Unlike other loop optimization methods that just consider innermost loops, loop flattening tends to effect the nested and sequenced loops [6], [3]. It doesn't duplicate any blocks; reduces the initiation interval merging all the blocks of a nested loop into a single loop body, thus regulating when various blocks execute and how data/values flow between these blocks [8]. From the table I, only perfect and semi-perfect loops can be flattened. Loop flattening can benefit from if-Conversion as it eliminates back-edges of flattened loops.

Perfect Loops	**Semi-perfect Loops**	**Other types**
Only the inner most loop has body (contents)	same as perfect loop	Should be converted to perfect or semi-perfect loops
There is no logic specified between the loop statements	same as perfect loop	
Loop bounds are constant	Outer most loop bound can be variable	

TABLE I: Loop Types

A very detailed view is observed in Figure 3 which explains the significant benefits obtained from flattening of the nested loops. The original code had nested loops, resulting in a total of 36 loop calls. After the loop flattening transformation, the enhanced code has 28 loop calls.

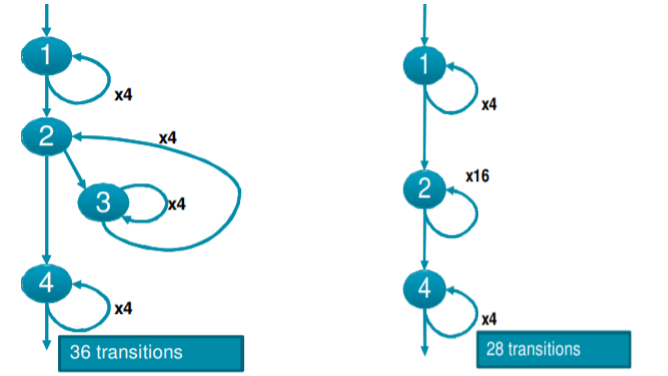
```
for (i = m; i < n; i++){
  ...
  }
```

Fig. 1: Loop Unrolling Example: 'for' loop construct program snippet [4]

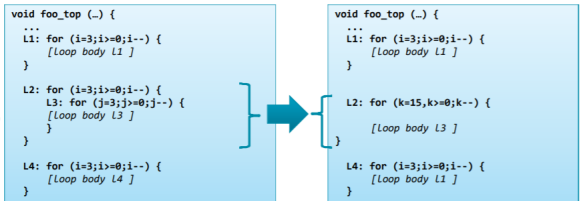
```
rem = (n - m) % unrollfactor + 1;
for (i = m; i < m + rem; i++){
  ...
  }
for (i = m + rem; i < n;
  i +=unrollfactor + 1){
  ...
  }
```

```
maxval = n - unrollfactor;
for (i = m; i < max_val;
     i +=unrollfactor + 1){
  ...
  }
for (j = i; j < n; j++) {
  ...
  }
```

Fig. 2: Leftover iterations inserted as prologue(fig 1) and epilogue(fig 2) [4]



(a) Before Loop Flattening (b) After Loop Flattening



(c) Loop flattening transformation

Fig. 3: Loop flattening example [8]

```
Loop_outer: for (i=3;i>=0;i--) {
   Loop_inner: for (j=3;j>=0;j--) {
      [Loop body]
   }
}
```

(a) Perfect Loop

```
Loop_outer: for (i=3;i>N;i--) {
   Loop_inner: for (j=3;j>=0;j--) {
      [Loop body]
   }
}
```

(b) Semi perfect loop

```
Loop_outer: for (i=3;i>N;i--) {
   [Loop body]
   Loop_inner: for (j=3;j>=M;j--) {
      [Loop body]
   }
}
```

(c) Other Loop

Fig. 4: Types of Loop for flattening [8]

C. Algorithm Implemented

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.

Our algorithm first reads the MiniC file and inlines all function calls. Acyclic code regions are predicated, i.e., all instructions are given a predicate condition that must 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 initial predicate condition $\varphi =$ **true**:

- $\mathsf{Pred}(\varphi, \lambda = \tau) := \lambda \stackrel{\varphi}{=} \tau$
- $\mathsf{Pred}(\varphi, \lambda_1, \lambda_2 = \tau) := \lambda_1, \lambda_2 \stackrel{\varphi}{=} \tau$
- $\mathsf{Pred}(\varphi, S_1\ S_2) := \mathsf{Pred}(\varphi, S_1)\ \mathsf{Pred}(\varphi, S_2)$

- $\mathrm{Pred}(\varphi, \textbf{if}(\sigma)\ S) := \mathrm{Pred}(\varphi \wedge \sigma, S)$
- $\mathrm{Pred}(\varphi, \textbf{if}(\sigma)\ S_1\ \textbf{else}\ S_2)$
  $:= \mathrm{Pred}(\varphi \wedge \sigma, S_1)\ \mathrm{Pred}(\varphi \wedge \neg\sigma, S_2)$
- $\mathrm{Pred}(\varphi, \textbf{while}(\sigma)\ S)$
  $:= \textbf{if}(\varphi)\ \textbf{while}(\sigma)\ \mathrm{Pred}(\textbf{true}, S)$
- $\mathrm{Pred}(\varphi, \textbf{do}\ S\ \textbf{while}(\sigma))$
  $:= \textbf{if}(\varphi)\ \textbf{do}\ \mathrm{Pred}(\textbf{true}, S)\ \textbf{while}(\sigma)$

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).

In this project report, we are developing our loop unrolling algorithm and determine empirically an optimal loop unroll factor by to overcome the above observed issues, as in Figure 2. Our main focus was observing whether our algorithm gives a better ILP and if yes, how does it vary with different loop combinations, summarizing the results observed here as well as from the previous project report.

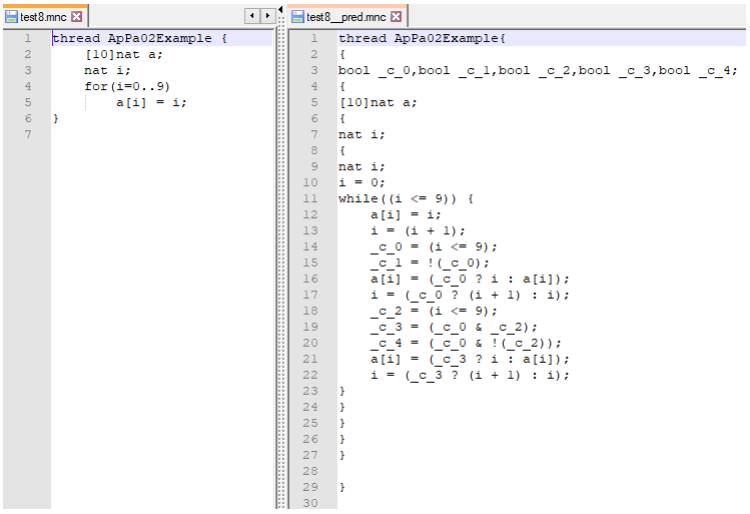


Fig. 5: Code transformation of a 'for' loop program

III. Algorithm

After the application of the above mentioned 'if-conversion transformation', 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. The 'loop transformation' algorithm then reiterates loops according to the following code transformations if the loop bodies do not contain other loops:

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

The above rules may be applied for a given number of times ℓ, thus creating $\ell + 1$ copies of an innermost loop body.

It is clearly observed that in our loop unrolling algorithm, there is no explicit correlation between the unrolling factor and loop iteration count. We make the inline copies of loop body, where the number of copies is defined by the 'unroll-factor' given in command line during the program execution. Neither for compile-time nor for execution-time, the iteration count need not be multiples of an unroll factor as mentioned in other research papers. This avoids the overhead caused from the left over iterations as seen in Figure 2.

For example, Figure 9 shows a simple algorithm for iterating an array a of 10 natural numbers. We explicitly pass the unroll factor during the execution. Figure 5 shows the code transformation when the loop is unrolled twice. As can be seen, when unrolled, the 'for' loop is replaced with 'while' and the loop body, i.e; when unrolled twice, the loop body is duplicated 3 times. Inline function expansion replaces frequently invoked function calls with the function body itself. Function inlining increases the code size minimally and providing the benefit of decreased number of function calls. According to [5], late application of loop unrolling is good. Less predication is required because loop unrolling cannot introduce new common sub expressions. Loop body would be considered as single block. After unrolling, the single earlier block is now increased in size due to the unroll-factor times.

As already noted from Figure 5, loop unrolling results in an increase of the code size. Our goal is to see if the algorithm gives a better ILP, and how many times a loop needs to be unrolled to get an efficient output. We now define a hypothesis: *The exact number of times a loop needs to be unrolled for optimized ILP cannot be defined, but a default value of '2' gives better result with a decent increase in code size due to expansion*. We have to evaluate this hypothesis by our empirical experiment.

IV. Experimental Evaluation

To measure the effect on the available ILP by our loop unrolling algorithm, we first obtain the control-flow graph (CFG) from the MiniC compiler[1], 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 improving the amount of ILP.

The benchmarks were chosen based on the combinations of loop types as mentioned in Table III. 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

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

	Benchmark	Description
1	Test9	Calculate vector length using the formula
2	Fibonacci	Calculate 'n' Fibonacci numbers
3	First Order Linear	Find recurrence relation of the type First Order
4	Insertion Sort	Sorting an array
5	Inner Product	Dot product of 'n' dimension vector space
6	Euclid	Compute Greatest Common Divisor
7	Heron	Integer Approximation of the sqrt of natural number
8	SumUp	Summation of 'n' consecutive numbers
9	Test7	Liveness Analysis
10	Test8	for loop
11	Insertion MaxSort	Another version of insertion sort
12	Test14	'for' loop with independent loop body
13	Daxpy	Vector Processing

TABLE II: Description of Benchmarks used by Experiments.

Loop Combination	Benchmark used
for	Test8, Daxpy
'for'+ independent loop body	Test14
while	Test10, SumUp, Euclid
if + while	Fibonacci
do + while	Test7, Heron
for + while	InsertionSort
while + while	InsertionMaxSort

TABLE III: Various Loop Combinations and Benchmarks

required to execute the basic block. For each basic block of the unpredicated and the predicated CFG, we use 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 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. The standard used to measure performance is Cycle Per Instructions (CPI).

The enhanced algorithm shows that loop unrolling plays a major role in determining the ILP of the program, due to the higher ILP obtained from the block containing the loop. Unrolling increases the size of the basic block containing the loop. This implies that the ILP of the entire program can be defined by the ILP of the block whose size is increased, since there is a cyclic entry to that block, as seen from Figure 6. Here, benchmark Test8 is unrolled twice and thrice, respectively. We observe that ILP and the size of other basic blocks without loop construct remains fixed even after unrolling as many times.

A. Comparison of Results

In this section, we present the results of our study across different benchmarks as well as across 2 different optimization algorithm approaches (If-Else, Loop unrolling). The study was conducted in 2 sets: In the first set, we performed the ILP calculations on programs catering to various types of branching constructs (see Table II). Figure IV presents the results in terms of percentage increase in the ILP. These programs contained if-then-else, function calls constructs, and loop statements. We can summarize that there has been an increase in the performance from the visualization obtained by listing the results in Figure 7.

In the second part, we performed the experiment on different benchmarks based on the combinations of loops as specified in Table III. We measured the impact on ILP due to loop unrolling passing different unroll factors like 2,3, and 4 during execution time from the command line. Since the ILP of non-loop blocks remains fixed, we discard the ILP of these basic blocks and consider only the ILP of the loop-unrolling blocks which is a comparatively bigger number. For the unroll factors 3 and 4, the change in ILP is very minimal which shows that unrolling had very limited effect. Figure V presents the results in terms of variations in ILP with respect to the loop unrolling factor. A significant improvement in ILP is observed graphically from Figure 8 which boldly states that unrolling loops heavily impacts ILP.

There has not been a significant increase in ILP when loops are unrolled more than twice. Thus, we can state that it is better to use **default unroll factor as 2**. Because the larger the unroll number, the more aggressive the loop unrolling, we get a very marginal improvement in ILP, but at the cost of very huge dataflow graph and huge increase in code size. It is noteworthy that code optimization techniques whose purpose is to improve the performance of a program, should not increase the executable code size significantly. Specially in areas of embedded processors, the increase in the size of the executable can offset the gains provided by unrolling. After the investigation and experimentation on loop unrolling, the result thus supports the hypothesis of using the default unroll factor as 2.

V. Conclusion and Future Work

Based on the results obtained from the experiments, we observe a significant improvement of the amount of ILP by predicating and unrolling the loops in the programs. If-conversion converts multiple basic blocks into a single

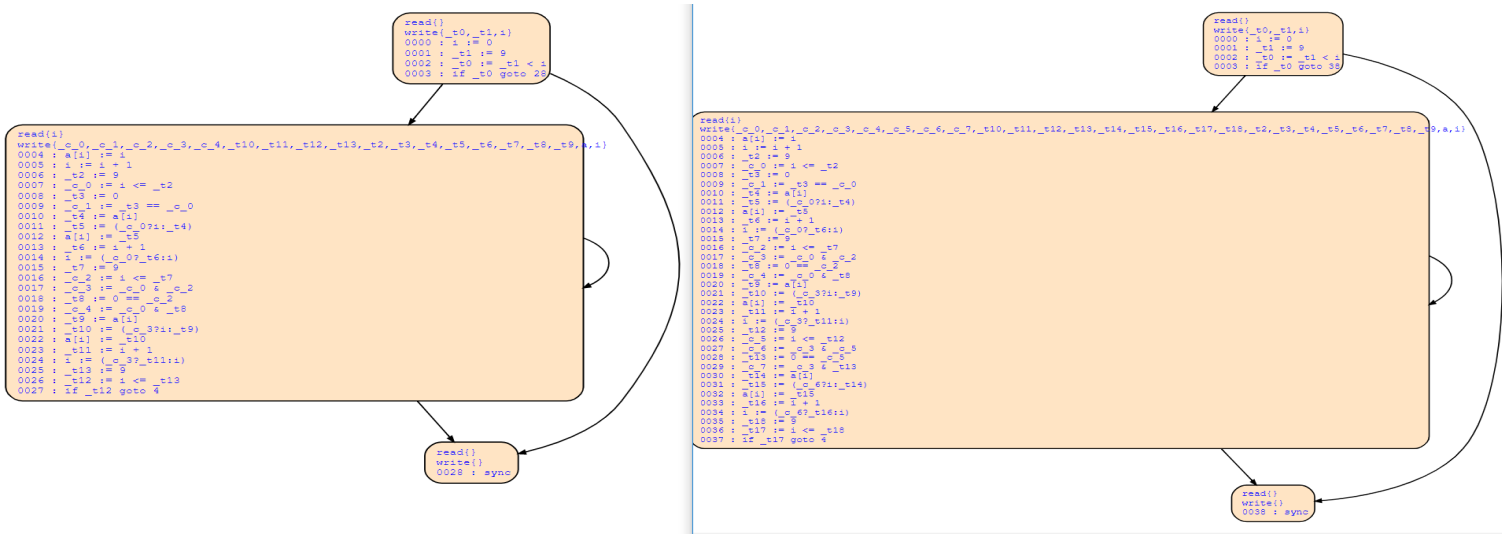


Fig. 6: CFG of Benchmark Test8 MiniC program, with loop unroll factor 2 and 3 in the first and second CFG respectively.

ILP Comparison on Predicated Execution(see Appendix for Benchmark Programs)			
Benchmark Program	**ILP Before Predication**	**ILP After Predication**	**% Increase in ILP**
Insertion Sort	1.29	1.76	36.43
Bubble Sort	1.39	1.64	17.98
Inner Product	1.42	1.6	12.68
First Order Linear Rec-1	2.47	3.04	23.07
Sum Up	1.22	1.3	5.9
Fibonacci	1.68	1.81	7.74
Test 9	2.55	3	17.65

TABLE IV: Measuring the Increase of ILP of Various Benchmark Programs.

ILP measure on Loop Unrolling(see Appendix for Benchamrk programs)			
Benchmark Program	**Factor = 2**	**Factor = 3**	**Factor=4**
Euclid	2.44	2.58	2.66
Test 10	3.53	3.65	3.87
SumUp	1.75	2	1.95
Test 8	1.91	1.83	1.87
Daxpy	1.4	1.4	1.46
Fibonacci	2.7	3.45	3.77
Test 7	1.91	2.06	2.04
Test 14	2.73	2.8	2.93
Heron	1.31	1.34	1.36
Insertion Sort	2.54	2.55	2.55
Insertion MaxSort	2.36	2.55	2.36

TABLE V: Observation of ILP variation for different Unroll Factors.

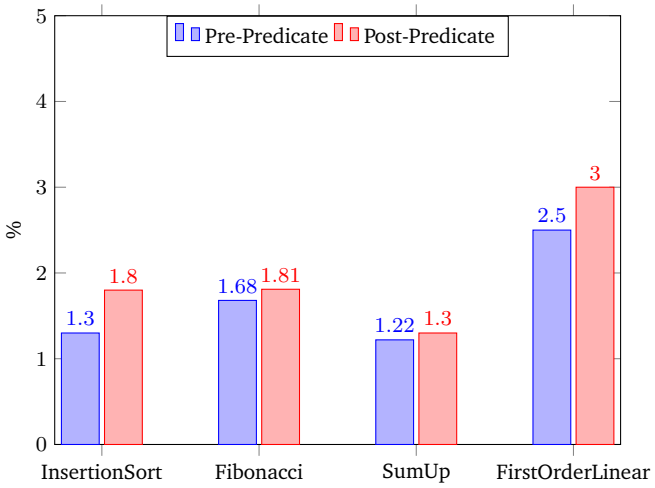


Fig. 7: Relative Increase of ILP with Predication.

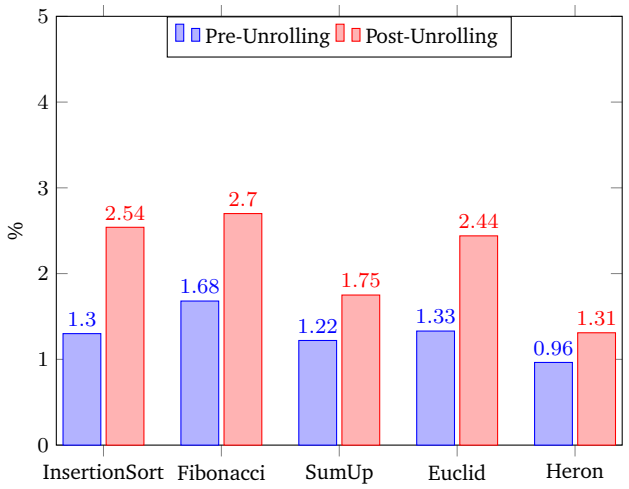


Fig. 8: Relative Increase of ILP with Loop Unrolling.

hyperblocks eliminating the conditional branches. Loop unrolling further enlarges the size of the single basic block by eliminating the conditional and branch jump overheads of the unrolled loop body. The enhanced algorithm shows that loop unrolling plays a major role in determining the ILP of the program due to the larger ILP obtained from the block containing the loop. Based on the selection of efficient loop unroll factor, we can determine that programs' executions is dominated by loop branches. Loop unrolling dynamically leads to increased code size due to code expansion. But with the appropriate choice of an unroll factor, we consider the ILP of the unrolled-loop basic block alone, since it has high weight. If the loop body is too thin, then no major improvement is observed by unrolling the loop body. At the same time, too fat loop bodies also lead to register spilling. Just like loop unrolling, function calls also lead to better ILP but increase the size of the block affecting the instruction table. Since we envision to apply these optimization algorithm on SCAD machines which has FIFO buffers and no overhead of register limitation resulting in out of order execution based on the availability of operands, we are not interested in the code size increase.

Loop flattening may be used as an additional performance booster by merging perfectly nested loops into single loop with additional logic for data flow. In future, we envision to implement all the ILP optimization techniques discussed here in SCAD machines to observe the behavior, as the registers are replaced by FIFO buffers. Execution time of real-time applications is a very important factor in determining the performance of the application. Our proposal of combining loop unrolling followed by predication applied on the SCAD architecture which totally supports asynchronous data execution, is fully expected to give a better execution time .

Some techniques like Single Static Assignment (SSA) or Predicated SSA (PSSA) are only exploitable for programs containing many variables involvements, and when processor architectures consist of registers involving register renaming. 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. Adjustment of hyperblocks during scheduling 'partial reverse if-conversion' as mentioned by [1] is still questionable as it is explorable when aggressive scheduling is performed. So, we can conclude that it depends on the code-transformation logic implementation of if-conversion.

References

[1] D. August, W.-M. Hwu, and S. Mahlke. The partial reverse if-conversion framework for balancing control flow and predication. *International Journal of Parallel Programming*, 27(5):381–423, October 1999.

[2] S. Bejai. Evaluating the effect of predication on instruction level parallelism. Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany, April 2019. Project.

[3] A. Carminati, R. Starke, and R. de Oliveira. Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software. *Applied Computing and Informatics*, 13(2):184–193, July 2017.

[4] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming*, 28(6):563–588, December 2000.

[5] J. Davidson and S. Jinturkar. An aggressive approach to loop unrolling. Department of Computer Science, Thornton Hall University of Virginia, 2001.

[6] S. Hauck. Enhanced loop flattening for software pipelining of arbitrary loop nests. University of Washington, WA, USA, 2010.

[7] L. Pozzi. Compilation techniques for exploiting instruction level parallelism, a survey. unpublished manuscript, 1999.

[8] Xilinx. All programmable, improving performance. Department of Electrical and Computer Engineering, University of Texas,USA.

APPENDIX

```
thread ApPa02Example {
   [10]nat a;
   nat i;
   for(i=0..9)
      a[i] = i;
}
```

Fig. 9: Simple 'for' loop

```
thread ApPa02Example {
   nat a,b,c,N;
   a = 0;
   do {
      b = a+1;
      c = c+b;
      a = b*2;
   } while(a<N)
}
```

Fig. 10: Liveness analysis

```
thread daxpy {
  [7]nat x,y;
  nat i,a;
  for(i=0..6)
    y[i] = a * x[i] + y[i];
}
```

Fig. 11: Vector processing

```
thread t {
   [10]nat x,y,z;
   nat i;
    for(i=0..9) {
      x[i] = i+1;
      y[i] = i+1;
      z[i] = x[i] + y[i];
   }
}
```

Fig. 12: 'for' loop with independent loop body

```
function heron(nat a) : nat {
  nat xold,xnew;
  xnew = a;
  do {
    xold = xnew;
    xnew = (xold + a/xold)/2;
  } while(xnew < xold)
  return xold;
}

thread Heron {
  nat z;
  z = heron(121); // compute the square root of 121
}
```

Fig. 13: Find the integer approximation to the square root of a natural number(Heron's algorithm)

```
function euclid(nat a,b) : nat {
   nat t;
   while(b!=0) {
     t = b;
     b = a % b;
     a = t;
   }
   return a;
}

thread Euclid {
   nat x,y,z;
   x = 147;
   y = 693;
   z = euclid(x,y);
}
```

Fig. 14: Find GCD using Euclid's algorithm.

```
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. 15: Find the n-th Fibonacci number in linear time.

```
thread test {
   int x,u,y,x1,u1,y1,dx,a;
   while(x<a) {
      x1 = x + dx;
      u1 = (u − ((+3 ∗ x) ∗ (u ∗ dx))) − (+3 ∗ y ∗ dx);
      y1 = y + u ∗ dx;
      x = x1;
      u = u1;
      y = y1;
   }
}
```

Fig. 16: Solve the differential equation: y" + 3xy' + 3y = 0 using forward Euler method

```
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. 17: Insertion sort

```
thread InsertionMaxSort {
   [10]nat x;
   nat y,n,i,j,jmax;
   // now start sorting
   i = n−1;
   while(1 < i){
      // compute maximum of x[0..N−1−i]
      jmax = 0;
      j = 1;
      while(j <= i) {
      if(x[jmax] < x[j])
         jmax = j;
      j = j+1;
      }
      // now x[jmax] is the maximum of x[0..N−1−i] and →
           is swapped with x[i]
      if(jmax != i) {
         y = x[i];
         x[i] = x[jmax];
         x[jmax] = y;
      }
      i = i−1;
   }
}
```

Fig. 18: Another version of insertion sort

```
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. 19: Summation of first 'n' natural numbers using n*(n+1)/2