INFORMATIK

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Evaluating the Effect of Predication on Instruction Level Parallelism

Sapna Bejai
Department of Computer Science
TU Kaiserslautern
Date: 02.04.2019

Agenda

- Granularity of Parallelism
- Instruction-Level Parallelism (ILP)
- Synchronous Control Asynchronous Dataflow (SCAD): Overview
- Predicated Execution: Overview
- Predication Algorithm
- Experiments and Results
- Conclusion and Future Work

Granularity of Parallelism

Measure of the amount of computation done in parallel of an algorithm.

Granularity of Parallelism	Parallelized blocks	Availability
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

ILP: What and Why

- ILP: parallel execution of sequence of instructions derived from a sequential program.
- Applied on multiple function units within a single microprocessor.
- Amount of ILP is application-specific.
- Compiler transformations (predication) can increase the amount of ILP.
- Overlap the execution of individual operations without explicit synchronization.

Continued...

Approaches to exploiting ILP:

- hardware approach: dynamic parallelism.
- software approach: static parallelism at compile time.

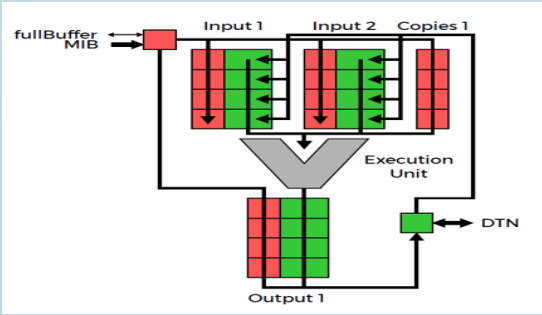
Compilers use:

➢ global knowledge about the program not available to the hardware.

➢ description of the target machine architecture to guide machine-specific optimizations.

ILP techniques on SCAD

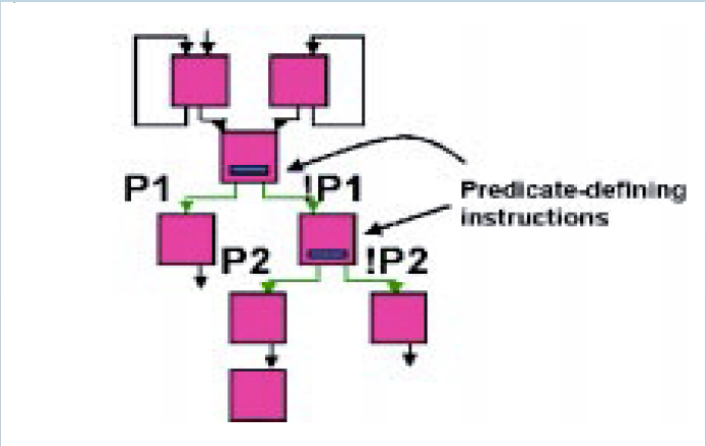
- code generation technique- breadth first traversal.
- operands are found in the correct order in the queue.
- no need for an additional memory.
- out-of-order execution leading to a good level of ILP.
- early availability of operands & independence of many instructions.



Compilation Techniques for ILP

- Basic block scheduling: may suffer from small size of basic blocks
- Extended basic block scheduling: groups of basic block scheduling

Predicated Execution



Loop Unrolling

```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    i = i + 1
}
```

```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    a[i+1] = b[i+2] + (i+2)/m
    b[i+1] = a[i] - (i+1)/m
    i = i + 2
}
```

Predicated Execution

- Predication (If-conversion) replaces a set of basic blocks of conditional controls with a predicated single block of instructions.

- Control dependencies are this way transformed into data dependencies.

Advantage:

➢ Some instructions are scheduled which are never executed, but more instruction can be scheduled in parallel to increase the amount of ILP.

➢ `then' and `else' parts are scheduled and executed simultaneously ; ignoring the instructions that were not supposed to be run.

Predication Example:MiniC Program Input

```
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);
}
```

Predicated Output(without Loop Unrolling)

```
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)
}
```

Predicated Output
(With Loop Unrolling)

```
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)
}
```

Experiment

- Remove all acyclic control-flow statements by predication.
- Reiterate innermost loop bodies for a given number of times.

➢ Compute CFG of the original and the predicated program, using the experimental MiniC compiler.

➢ Compute ILP= (Number of instructions) / (Length of the longest path of the data dependency graphs).

➢ Compare their respective ILPs.

Predicate Algorithm Definition

MiniC Statements:

➢ Atomic data types : bool, nat, int

➢ Arrays : [n]α for arrays on type α

- $\lambda = \tau$;
- $\lambda_1, \lambda_2 = \tau$;
- $S_1\ S_2$
- $\mathbf{if}(\sigma)\ S$
- $\mathbf{if}(\sigma)\ S_1\ \mathbf{else}\ S_2$
- $\mathbf{while}(\sigma)\ S$
- $\mathbf{do}\ S\ \mathbf{while}(\sigma)$
- $\mathbf{for}(i = m..n)\ S$
- $\mathbf{return}\ \tau$
- $\mathbf{sync}$

Predicate Assignment Rule:

$\varphi = \mathbf{true}$:

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

Continued...

Procedure:

- ➢ Inline all function calls.
- ➢ Reiterate loops and follows the code transformation rule.
- ➢ Predicated assignment rule works only if the predicate condition is true .

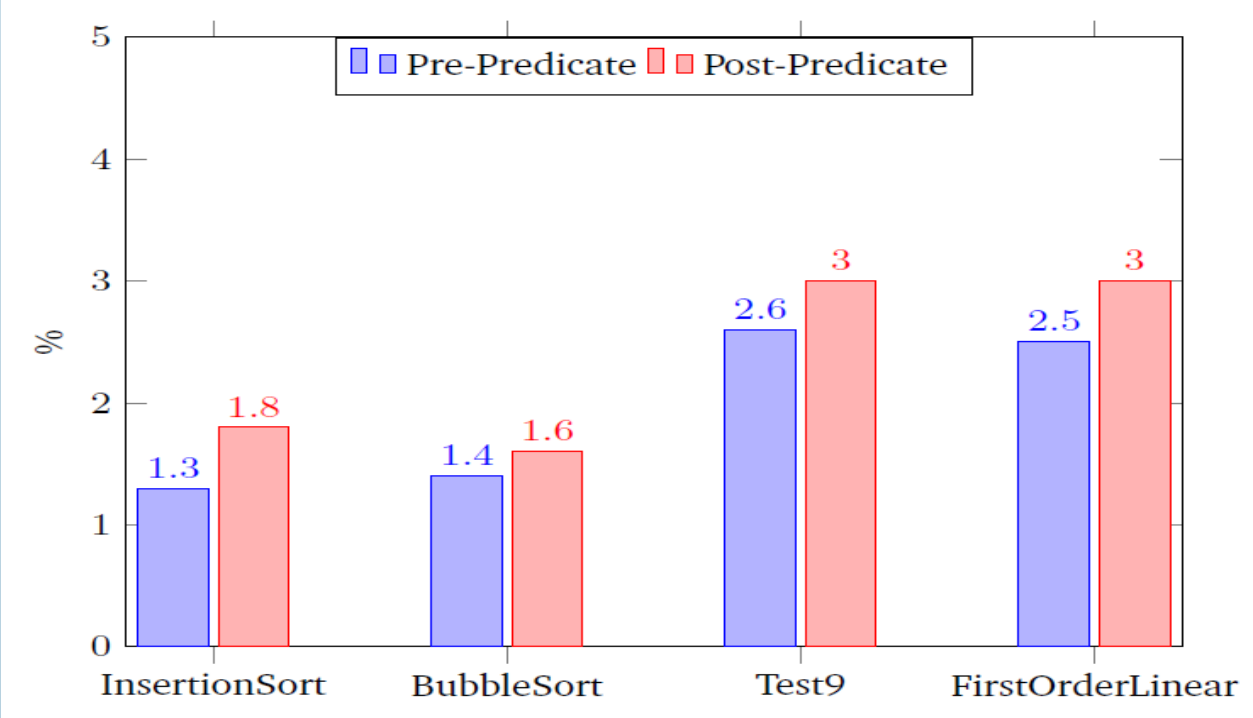
Code Transformation Rule:

- **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; $\}$

Compare block sizes (Bubble Sort - without unrolling)

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	

Result Comparison



Conclusion & Future Work

Limitation:
Single Static Assignment (SSA) can be exploited only when the program has various variables.

Future Work: Investigate on

- Domination of Program execution by loop or non-loop branches.
- Feasibility of increased code size due to Loop Unrolling.
- Discardment of other BBs when loop unrolled larger times.
- Relation between False dependencies and Reverse If Block technique.

Thank You

Any Questions ?