Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software

Sokratis Koseoglou

Loop Unrolling-
Code Predication

Motivational Example

Authors’ Optimization Approach

Evaluation & Results

Worst-Case Execution Time (WCET)

- Worst-Case Execution Time (WCET) is an important parameter for a Real-Time System
- Time Analyzers are used in order to evaluate the WCET
- Goal of this paper is to reduce the WCET of a real-time system
- There are two ways to reduce the WCET

Faster Processor

Software Optimization

Loop Unrolling

- Loops are frequently good targets for compiler optimizations

```
for(int i = 10; i  != 0; i = i - 1){
      a[i] = a[i] + b;
}
```

LOOP:	LW	r2, 0(r1)
	ADD	r2, r2, r3
	SW	r2, 0(r1)
	ADDI	r1, r1, -4
	BNE	r1, r5, LOOP

5 Instr. x 10 Iter. = 50 Instructions

Loop Unrolling

Unrolling Factor = 1

```
for(int i = 10; i  != 0; i = i - 2){
      a[i] = a[i] + b;
      a[i - 1] = a[i - 1] + b
}
```

LOOP:	LW	r2, 0(r1)
	ADD	r2, r2, r3
	SW	r2, 0(r1)
	LW	r2, -4(r1)
	ADD	r2, r2, r3
	SW	r2, -4(r1)
	ADDI	r1, r1, -8
	BNE	r1, r5, LOOP

8 Instr. x 5 Iter. = 40 Instructions

Loop Unrolling – keil µVision ARM Example

Demo Code for Loop Unrolling

main.c

```c
void loop(int* A, int b){

  for(int i = 10; i != 0; i = i - 1){
    A[i] = A[i] + b;
  }

}

void unrolledLoop(int* A, int b){

  for(int i = 10; i != 0; i = i - 5){
    A[i] = A[i] + b;
    A[i - 1] = A[i - 1] + b;
    A[i - 2] = A[i - 2] + b;
    A[i - 3] = A[i - 3] + b;
    A[i - 4] = A[i - 4] + b;
  }

}

int main(void ){

  int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int b = 1;

  loop(A, b);
  //unrolledLoop(A, b);

  return 0;
}
```

Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	1366
Sec	0.00011383
FPU	

Rolled Loop Time

Unrolled Loop Time

Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	1347
Sec	0.00011225
FPU	

Branch Prediction vs Code Predication

Branch Prediction

- Predict whether a branch is taken or not
- Synchronous Architectures have about 90% Accuracy
- No penalty if correct prediction
- Huge penalty if wrong prediction

Code Predication

- Do work of both directions
- Throw away instructions from the wrong path
- Waste up to 50%

- Big if-else → Branch Prediction
- Small if-else → Code Predication

```
        BEQ     r1, 0, else
        ADDI    r2,r2,1
else:
        ADDI    r2,r2,-1
```

```
If (f){
    a++;
}else{
    a--;
}
```

```
(f)      ADDI   r2,r2,1
(not f)  ADDI   r2,r2,-1
```

Loop Unrolling-
Code Predication

Motivational Example

Motivational Example Loop

```
void loop(int a){
    int i, j = 0, k = 0;

    for(i = 0; i < a ; i++){
        j++;
        k++;
    }
}

int main(int a){
        loop(90);
}
```

Listing 1. Simple data-dependent loop.

- Data-Dependent Loop (since loop iterations variable is not static)
- Compiler generated a Control Flow Graph (CFG)
- A problem with Data-Dependent loops is the difficulty to choose an effective unrolling factor
- We need to add exit conditions for each body replication

for(i = 0; i < a; i++)	(Header)
j++; *k++;*	(Body)

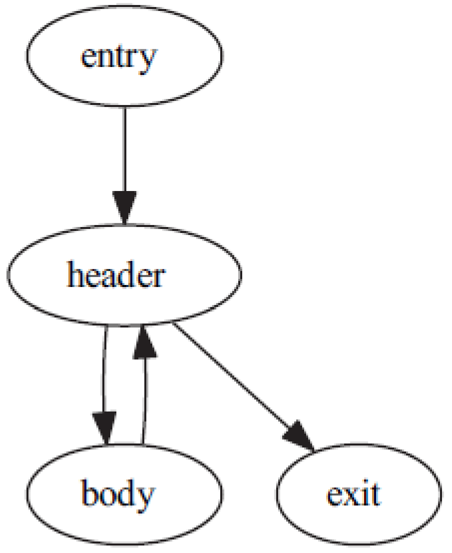

Fig. 1. Control flow graph of Listing 1.

Loop Unrolling – keil µVision ARM Example

Demo Code for Loop Unrolling with exit conditions

main.c | startup_stm32f401xe.s

```c
void falseUnroll(int* A, int b){

  for(int i = 10; i != 0; i = i - 3){
    A[i] = A[i] + b;
    A[i - 1] = A[i - 1] + b;
    A[i - 2] = A[i - 2] + b;
  }

}

void correctUnroll(int* A, int b){

  for(int i = 10; i != 0;){
    A[i] = A[i] + b;
    i--;
    if(i != 0){
      A[i - 1] = A[i - 1] + b;
      i--;
    }
    if(i != 0){
      A[i - 2] = A[i - 2] + b;
      i--;
    }
  }

}

int main(void ){

  int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int b = 1;

  //loop(A, b);
  //unrolledLoop(A, b);
  falseUnroll(A, b);
  correctUnroll(A, b);

  return 0;
}
```

Motivational Example Loop

```
void loop(int a){
    int i,j = 0,k = 0,l = 0;

    for(i = 0; i < a; i+=l){
        l = 0;
        j++;
        k++;
        l++;
        if(i+l >= a) break;
        j++;
        k++;
        l++;
        if(i+l >= a) break;
        j++;
        k++;
        l++;
    }
}
```

Listing 2. Unrolled loop.

- Note that, instructions increased
- If the target architecture supports code predication, we can consider the following code, in which we are able to do *if-conversion*

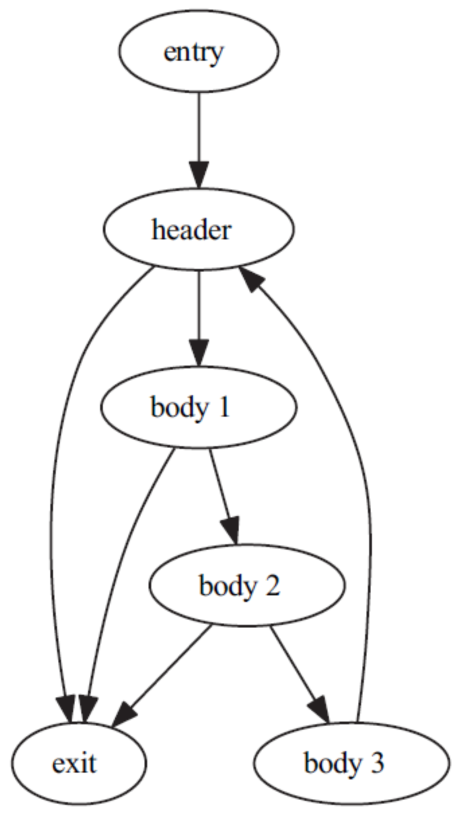

Fig. 2. Control flow graph of Listing 2.

Motivational Example Loop

```
void loop(int a){
    int i, j = 0, k = 0;

    for(i = 0; i < a;){
        j++;
        k++;
        i++;

        if(i < a){
            j++;
            k++;
            i++;
        }
        if(i < a){
            j++;
            k++;
            i++;
        }
    }
}
```

- The prefix (p) means that the execution of blocks *body2* and *body3* are conditioned to some predication guard p

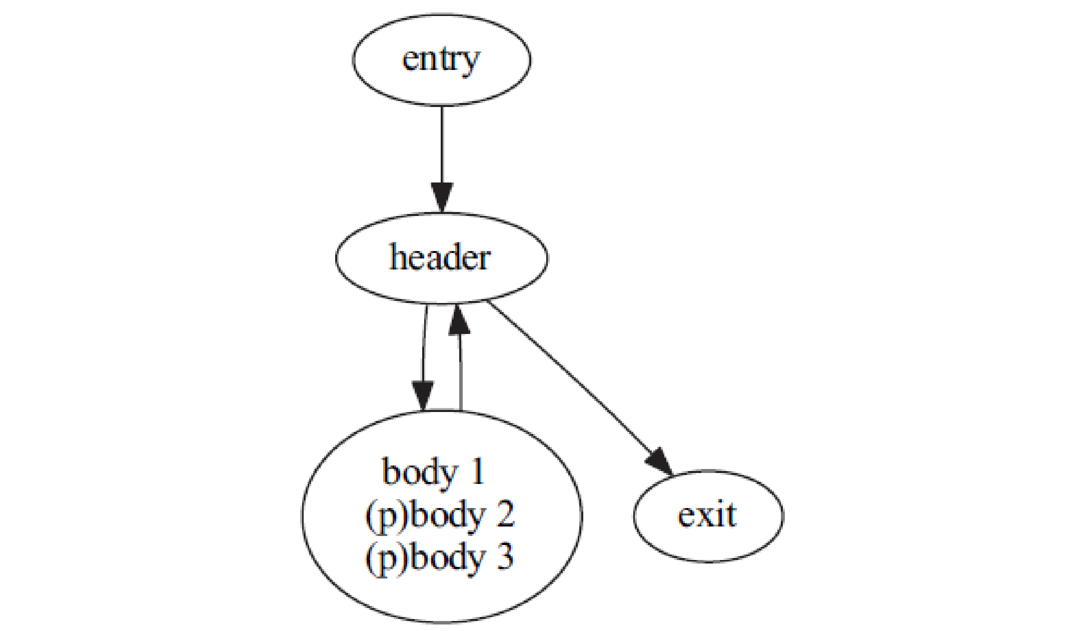

Fig. 4. Control flow graph representing an *If-Conversion* of the code from Listing 3.

Loop Unrolling-
Code Predication

Motivational Example

Authors’ Optimization Approach

Predicated Loop Unrolling Algorithm


```
procedure PREDICATEDLOOPUNROLLING(Loop, U, P)
  ▷Unroll loop u times
    Header ← loopHeader(Loop)
    Body ← loopBody(Loop)
    removeUncondBranch(Body)
    BodyCopy ← createCopy(Body)
    for i ← 1 to U − 1 do
        NewHeader ← createCopy(Header)
        removeConditionalBranch(NewHeader)
        changeCompareOutput(NewHeader, P)
        Body ← unify(Body, NewHeader)
        NewBody ← createPredCopy(BodyCopy, P)
        Body ← unify(Body, NewBody)
    end for
    insertUncondBranch(Body, Header)
end procedure
```

- Identify Header & Body of the loop
- Remove unconditional branch and place it in the end
- Create a body copy which will be always executed
- Create predicated body copies inside the for loop, with U iterations, where U is the *unrolling factor*

Example of our Approach

```
void loop(int a){
    int i, j = 0, k = 0;

    for(i = 0; i < a ; i++){
        j++;
        k++;
    }
}

int main(int a){
        loop(90);
}
```

Listing 1. Simple data-dependent loop.

Assembly

```
   add $r8 = $zero, 0
   add $r9 = $zero, 0
   add $r10 = $zero, 0
HEADER:
   cmplt $br0, $r9, $r16
   brf $br0, $EXIT
BODY:
   add $r10 = $r10, 1
   add $r8 = $r8, 1
   add $r9 = $r9, 1
   goto $HEADER
EXIT:
```

Listing 4. Example of loop in assembly code.

Example of our Approach

Standard Approach with branch for exit conditions

```
   add $r8 = $zero , 0
   add $r9 = $zero , 0
   add $r10 = $zero , 0
HEADER:
   cmplt $br0 , $r9 , $r16
   brf $br0 , $EXIT
BODY0:
   add $r10 = $r10 , 1
   add $r8 = $r8 , 1
   add $r9 = $r9 , 1
   cmplt $br0 , $r9 , $r16
   brf $br0 , $EXIT
BODY1:
   add $r10 = $r10 , 1
   add $r8 = $r8 , 1
   add $r9 = $r9 , 1
   goto $HEADER
EXIT:
```

Listing 6. Example of unrolled loop using the standard approach.

Predicated Approach

```
   add $r8 = $zero , 0
   add $r9 = $zero , 0
   add $r10 = $zero , 0
HEADER:
   cmplt $br0 , $r9 , $r16
   brf $br0 , $EXIT
BODY:
   add $r10 = $r10 , 1
   add $r8 = $r8 , 1
   add $r9 = $r9 , 1
   cmplt $p , $r9 , $r16
   (p)add $r10 = $r10 , 1
   (p)add $r8 = $r8 , 1
   (p)add $r9 = $r9 , 1
   goto $HEADER
EXIT:
```

Listing 5. Example of unrolled loop using code predication.

- We can see that the predicated version has fewer instructions

Optimization Choice Algorithm

Optimizations Algorithm that is executed by the compiler

```
procedure OptimizeLoops(Program)
    LoopList ← getLoops(Program)
    for each loop ∈ LoopList do
        if not loop.isDataDep then
            simpleLoopUnrolling(loop, loop.uf)
        else if loop.hasCall then
            branchedLoopUnrolling(loop, loop.uf)
        else
            predicatedLoopUnrolling(loop, loop.uf, P)
        end if
    end foreach
end procedure
```

- Compiler runs this code for each loop that is in the WCEP

- Standard (without branches)

  Used for non-data dependent loops (with fixed executions counts)

- Standard (with branches)

  Used for data dependent loops that have call instructions in the body

- Predicated

  Used for data dependent loops that have simple bodies and do not use call instructions

Unrolling Factor Selection


```
procedure CALCULATEUNROLLINGFACTORS(Program)    ▷ Algorithm executed by the optimization planning tool
    LoopList ← getLoops(Program)
    wcetData ← calculateWCET(Program)
    for each loop ∈ LoopList do
        if isInWCEP(loop, wcetData) then
            lastUF ← 0
            for i ← 2 to 17 do
                if not loop.isDataDep then
                    if not divides(loop.bound, i) then
                        continue
                    end if
                end if
                if parity(loop.bound) = parity(i) then
                    loop.uf ← i
                    recompile(Program)
                    newWcet ← calculateWCET(Program)
                    if newWcetData.value >= wcetData.value then
                        loop.uf ← lastUF
                    else
                        wcetData ← newWcetData
                        lastUF ← i
                    end if
                end if
            end for
        end if
    end for each
end procedure
```

- It is necessary to find the optimal *Unrolling Factor* for each loop that resides inside the WCEP
- If it is in the WCEP, we test different *Unrolling Factors*, from 2 to 17
- For Data-Independent loops the *Unrolling Factor* must exactly divide the execution count of the loop in order to avoid corner exit conditions
- For Data-Dependent loops we consider *Unrolling Factors* whose parity is equal to the loop bounds’ parity
- Then we recompile the program and test for WCET changes
- Algorithm complexity is $O(n^2)$
- Better WCET with the cost of higher compilation time

Loop Unrolling-
Code Predication

Motivational Example

Authors’ Optimization Approach

Evaluation & Results

Target Architecture

- Supports a simplified complete Predication Support
- 32-bit Microprocessor with RISC Instructions
- Direct-Mapped Cache Memory(32 blocks of 256 bits, thus 1024kB)
- No Data-Cache
- No out-of-order execution
- Four issue, five stage static scheduled pipeline

Results

Table 1
Obtained results

Benchmark	Initial WCET	Initial code size	Optimized WCET	Optimized code size	WCET reduction (%)	Code increase (%)
adpcm.c	19,607	10,208	19,373	14,816	1.19	31.10
bsort100.c	272,623	432	271,985	560	0.23	22.86
cnt.c	9046	752	8566	864	5.31	12.96
compress.c	140,139	4912	137,162	5488	2.12	10.50
crc.c	113,846	2048	112,671	2624	1.03	21.95
duff.c	1859	592	1397	816	24.85	27.45
edn.c	96,871	2336	73,042	3296	24.60	29.13
expint.c	113,473	1540	76,661	1812	32.44	15.01
fft1.c	1,034,634	19,984	727,844	44,272	29.65	54.86
fir.c	509,930,221	976	444,387,758	1616	12.85	39.60
insertsort.c	2720	304	2111	432	22.39	29.63
jfdctint.c	3947	1264	3568	1536	9.60	17.71
lms.c	352,360,015	14,016	303,674,957	70,768	13.82	80.19
ludcmp.c	43,902	3296	43,622	4320	0.64	23.70
matmult.c	268,362	1008	225,657	1968	15.91	48.78
ndes.c	146,612	5376	144,282	7248	1.59	25.83
qsort-exam.c	503,582	2528	480,816	2784	4.52	9.20
st.c	1,480,353	5088	1,198,582	5856	19.03	13.11
				Average	6.72	15.56
				Maximum	32.44	80.19

$$WCET\ Reduction = \frac{Initial\ WCET - Optimized\ WCET}{Initial\ WCET} * 100$$

$$Code\ Increase = \frac{Initial\ Code - Optimized\ Code}{Initial\ Code} * 100$$

- A total of 18 from 33 benchmarks are shown, since there was no gain in the other ones
- We can see that the combination of techniques was able to reduce the WCET of half of the benchmarks

Thank you for your time!

Questions?