고성능 디지털 신호 처리 프로세서상에서 효율적인 모듈로 스케쥴링을 위한 전처리 기법
(Preprocessing Methods for Effective Modulo Scheduling on High Performance DSPs)

조 두 산 [†] 백 윤 흥 [††]
(Doosan Cho) (Yunheung Paek)

요 약 고성능 다중 이슈 DSP의 하드웨어 리소스 사용률을 높이기 위해서, 제공되는 상용 컴파일러는 일반적으로 반복 모듈로 스케쥴링(Iterative Modulo Scheduling)을 포함하고 있다. 하지만, 통신 및 미디어 처리 응용의 루프에 존재하는 과도한 순환 데이타 의존관계는 모듈로 스케쥴링 자유도를 제한하고 있다. 결과적으로, 멀티 이슈를 위한 DSP의 병렬 기능 유닛들은 완전히 사용되고 있지 못하다. 이러한 하드웨어 리소스 저사용 문제를 해결하기 위하여, 이 논문은 효율적인 모듈로 스케쥴링을 위한 새로운 컴파일러 전처리 기법을 기술하고 있다. 제안하는 전처리 기법은 두 가지로서 클로닝과 디스맨틀링으로 불리우며, 이 두가지 기법들은 StarCore SC140 DSP 컴파일러에 구현하여 검증하였다.
키워드 : 컴파일러, 소프트웨어 파이프라이닝, 고성능 다중이슈 DSP, 모듈로 스케쥴링

Abstract To achieve high resource utilization for multi-issue DSPs, production compiler commonly includes variants of iterative modulo scheduling algorithm. However, excessive cyclic data dependences, which exist in communication and media processing loops, unduly restrict modulo scheduling freedom. As a result, replicated functional units in multi-issue DSPs are often under-utilized. To address this resource under-utilization problem, our paper describes a novel compiler preprocessing strategy for effective modulo scheduling. The preprocessing strategy proposed capitalizes on two new transformations, which are referred to as cloning and dismantling. Our preprocessing strategy has been validated by an implementation for StarCore SC140 DSP compiler.
Key words : compiler, software pipelining, high performance multi-issue DSP, iterative modulo scheduling

1. INTRODUCTION

As communication and media signal processing applications are getting more complex, system designers seek programm-able high performance fixed-point Digital Signal Processors (DSPs) as a flexible platform [1]. Recent *multi-issue* high performance DSPs[1)] are designed to meet such demand by supporting (1) multiple functional units, (2) advanced issue logic that allows a variable number of instructions to be dispatched in parallel, and (3) providing optimizing compilers that automatically tune algorithms written in C for performance [2-6].

In particular, to exploit multiple functional units available for mutli-issue DSPs, optimizing compilers commonly use software pipelining strategy. Software pipelining is a global loop scheduling concept

· This work was partially funded by the MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031), MIC under Grant A1100-0501-0004 and IT R&D Project, the Korea Ministry of Science and Technology(MoST) under Grant M103BY010004-05B2501-00411, Nano IP/SoC promotion group of Seoul R&BD Program in 2007.
† 학생회원 : 서울대학교 전기공학과
mew26@snu.ac.kr
†† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
ypaek@snu.ac.kr
논문접수 : 2006년 8월 16일
심사완료 : 2007년 2월 14일

1) Dominant market players in multi-issue DSPs are ADI/INTEL, Blackfin ADSP-BF53x, Motorola/Agere StarCore SC140, and Texas Instruments TMS320C64x.

that exploits instruction level parallelism across loop iteration boundaries. Since the critical path for a software pipelined loop will be shorter, resource utilization can be drastically improved. For this merit, production compilers commonly adopt variants of iterative modulo scheduling pioneered by Rau and Glaser [7]; for a detailed introduction, consult [8]. Although existing iterative modulo scheduling approaches are proven to be effective [3,5,9-12], excessive cyclic data dependences, which are frequently observed in communication and media processing loops, unduly restrict modulo scheduling freedom [13]. As a result, replicated functional units in multi-issue DSPs are often left being unused.

To address resource under-utilization problem for VLIW processors, Lavery and Hwu developed a preprocessing strategy, which uses loop-unrolling, for IMPACT iterative modulo scheduler [14]. Sakar complemented Lavery and Hwu research by developing (1) a more detailed cost model that calculates loop-unrolling factors and (2) code generation algorithms that generate more compact code than unroll-and-jam transformation [15]. However, loop-unrolling based preprocessing strategy for effective modulo scheduling can potentially incur significant increase in both code size and register pressure. Considering stringent constraints on both memory and register file size for DSPs, loop-unrolling based preprocessing pose tremendous challenges for DSP compilers to implement.

To address the same resource under-utilization problem for multi-issue DSPs, this paper presents an alternative preprocessing strategy for effective modulo scheduling, which capitalizes on two new compiler transformations, referred to as cloning and dismantling. Since these two transformations directly relax excessive cyclic data dependences with a partial aid from unused functional resources, neither code duplication nor additional hardware support are required. Therefore, cloning and dismantling transformations are easier for DSP compilers to implement. However, application of cloning and dismantling inevitably increases resource contention and indiscreet application can potentially make overall resource utilization even worse. Thus, the real preprocessing challenge is how best to apply these two transformations subjected to the constraint of resource pressure increase. The proposed strategy for effective modulo scheduling responds to this challenge. To measure feasibility and effectiveness of our strategy, StarCore SC140 processor [4,16] is used as the representative for multi-issue DSPs.

2. PRELIMINARIES

2.1 SC140 MULTI-ISSUE DSP ARCHITECTURE

The SC140 is a high performance general purpose fixed point DSP core. It supports multi-issue functionality consists of three main functional blocks,

1. Program Sequencer (PSEQ),
2. Data Arithmetic and Logic Unit (DALU), and
3. Address Arithmetic Unit (AAU).

PSEQ performs instruction fetch, instruction dispatch, and exception processing. To support high computing needs, DALU has 4 units of ALU and AAU has 2 units for address generation. To make all ALUs and AAUs operational at the same time, sixteen 40-bit data registers (d0-d15) and sixteen 32-bit address registers (r0-r15) are provided as general purpose registers (GPR). The ALU has three main components: multiply-accumulate (MAC) unit, a bit field unit and eight data bus shifter/limiters. AAU implements four types of arithmetic: linear, modulo, multiple wrap-around modulo, and reverse-carry.

A significant design point in SC140 ***is Variable Length Execution Set*** (multi-issue). Most SC140 instructions are 16-bits wide and they can be grouped into multi-issue packets of up to 128-bits. This allows for multiple instructions to be issued with reasonable instruction code density. In order to keep dispatching one multi-issue packet per cycle, the processor has 5-stages pipeline and the first 3 stages are dedicated for PSEQ. Therefore, most of control-free ALU and AAU instructions require delay of one clock cycle except AAU instructions that contain advanced addressing mode operation, such as indexed addressing.

To saturate 4 units of ALU and 2 units of AAU per cycle, compiler software piplines signal processing loop kernels with iterative modulo scheduling algorithm described in [8].

2.2 NOMENCLATURE: Iterative Modulo Scheduling

Definition 1 A candidate loop for an iterative modulo scheduler is the loop with branch-free body[2] that can run in DSP hardware looping mode [17].

Definition 2 Initiation Interval (II) of a candidate loop is the rate at which new loop iteration can be started.

Definition 3 Data Dependence Graph (DDG) of a candidate loop is the graph that defines a partial order, denoted by a tuple (latency l, iteration difference ω) between every two instructions in the loop body.

Definition 4 A recurrence circuit is a data dependence circuit that exists in a DDG, which is formed from an instruction to an instance of itself.

Definition 5 For a given recurrence circuit, rc, II_{rc} is defined as where Ω_{rc} = sum of individual iteration differences, ω_i and L_{rc} = sum of individual latencies, l_i, existing in the recurrence circuit.

Definition 6 Minimum Recurrence bound (RecMII) is the maximum of all II_{rc} which can meet the deadlines imposed from all the recurrence circuits existing in a candidate loop.

Definition 7 Minimum Resource bound (ResMII) is the smallest II which can meet the total resource requirements to complete one loop iteration of a candidate loop.

Definition 8 Excessive RecMII (Ex-RecMII) is the difference between RecMII and ResMII, iff RecMII > ResMII.

Definition 9 Minimum Initiation Interval (MII) is the maximum of RecMII and ResMII.

3. MOTIVATION: Excessive RecMII (Ex-RecMII)

According to our benchmark for SC140, various signal processing loop kernels manifest that excessive RecMII (Ex-RecMII) is the dominant limiting factor that either fails candidate loops to be modulo scheduled or modulo schedules with excessively large II.

3.1 LOOP-CARRIED TRUE DEPENDENCE

As the first example of Ex-RecMII, consider C code fragment shown in Figure 1(a) that implements Fast Fourier Transformation (FFT) algorithm. For the shaded candidate loop body in Figure 1(a), compiler produces highly optimized assembly code as shown in Figure 1(b), which is yet to be *modulo scheduled.*

For iterative modulo scheduling, II of candidate loop in Figure 1(b) is initially set equal to MII, which is computed as follows. First, each iteration of branch-free loop body shown in Figure 1(b) requires 7 units of ALU and 6 units of AAU, and SC140 multi-issue DSP can supply at most 4 units of ALU and 2 units of AAU per cycle. Thus, ResMII is 3, which is max($\left\lceil \frac{7}{4} \right\rceil, \left\lceil \frac{6}{2} \right\rceil$). Second, this candidate loop body contains several data dependence recurrence circuits. According to DEFINITION 6, RecMII is the maximum II_{rc} of all recurrence circuits, which is 6 for FFT and one such circuit is depicted in Figure 2(a).[3] Since MII is max(ResMII, RecMII), II is initially set to 6 for a modulo schedule.

For analysis, consider the loop-carried data dependence in Figure 2(a). This dependence is true since the value of induction variable r1 in the 9th *instruction is referenced by the* 1st *instruction in* the subsequent loop iteration. In addition, dependence chain from the 1st instruction down to the 9th instruction is transitively true. Due to this cyclic true dependence, FFT candidate loop fails to be modulo scheduled since MII of 6 is the ratio which can be achieved by local acyclic scheduling.

3.1.1 CLONING to relax Ex-RecMII due to true dependences

Since the recurrence circuit in Figure 2(a) is formed with cyclic true dependence, Ex-RecMII of FFT candidate loop deems irreducible. However, careful analysis on this circuit leads us to observe following:

2) Compiler performs if-conversion to allow more loops to be modulo scheduled.

3) The other RecMII circuit is omitted since the type of loop-carried data dependence is same as that of the circuit in Figure 2(a).

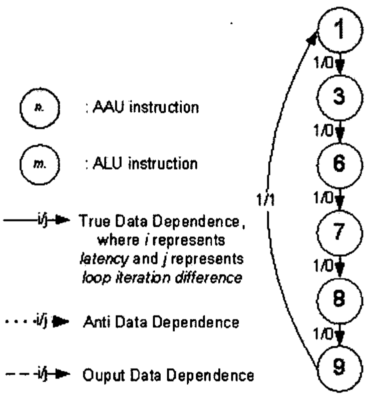
```
for (n=0; n<6; n++) {
   for (k=64; k<128; k++) {
        twa_1 = re_ac[m][k];
        twa_2 = im_ac[m][k];
        re_tw = twiddle[n][j];
        j++;
        im_tw = twiddle[n][j];
        j++;
        re_ac[m][k] = (twa_1*re_tw) - (twa_2*im_tw);
        re_ac[m][k] = re_ac[m][k] >> 10;
        im_ac[m][k] = (twa_1*im_tw) + (twa_2*re_tw);
        im_ac[m][k] = im_ac[m][k] >> 10;
   }
```

(a) A candidate loop from FFT

SC140 Instruction	Comments
1. move.w (r1),d3	Load *re_ac[m][k]* to d3
2. move.w (r6)+n3,d4	Load *twiddle[n][j]* to d4 and postincrement the array index by 4 bytes
3. *impy* d4,d3,d5	d5 <- *twa_1* * *re_tw*
4. move.w (r0),d6	Load *im_ac[m][k]* to d6
5. move.w (r3)+n3,d7	Load *twiddle[n][j]* to d7 And postincrement the array index by 4 bytes
6. imac -d7,d6,d5	d5 <- d5 – (*twa_2* * *im_tw*)
7. sxt.l d5	d5[39:32] <- d5[31]
8. asrr #<10,d5	d5 <- (d5 >> 10)
9. move.l d5,(r1)+	Store d5 to *re_ac[m][k]* and postincrement the array index by 4 bytes
10 impy d4,d6,d8	d8 <- *twa_2* * *re_tw*
11. imac d7,d3,d8	d8 <- d8 + (*twa_1* * *im_tw*)
12. asrr #<10,d8	d8 <- (d8 >> 10)
13. move.l d8,(r0)+	Store d8 to *im_ac[m][k]* and postincrement the array index by 4 bytes

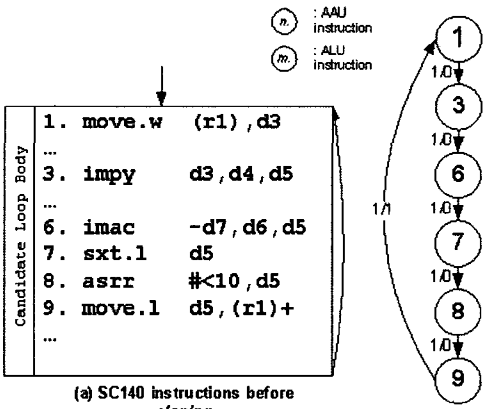
n. AAU instruction m. ALU instruction

(b) An assembly code of figure (a)


n. : AAU instruction
m. : ALU instruction
i/j True Data Dependence, where i represents latency and j represents loop iteration difference
i/j Anti Data Dependence
i/j Ouput Data Dependence
1
3
6
7
8
9
1/0
1/0
1/0
1/0
1/0
1/1

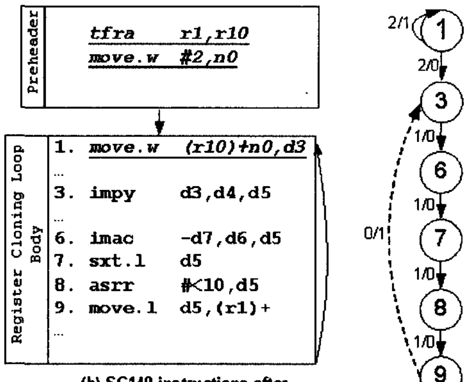
(c) A simplified data dependence graph of figure (b)

Figure 1 C Code fragment from FFT and corresponding loop body in SC140



```
1. move.w    (r1),d3
...
3. impy      d3,d4,d5
...
6. imac      -d7,d6,d5
7. sxt.l     d5
8. asrr      #<10,d5
9. move.l    d5,(r1)+
...
```

(a) SC140 instructions before *cloning*



```
tfra      r1,r10
move.w    #2,n0
```

```
1. move.w    (r10)+n0,d3
...
3. impy      d3,d4,d5
...
6. imac      -d7,d6,d5
7. sxt.l     d5
8. asrr      #<10,d5
9. move.l    d5,(r1)+
...
```

(b) SC140 instructions after *cloning*

Figure 2 RecMII=6 recurrence circuit of FFT candidate loop

1. The loop-carried true dependence in Figure 2(a) is an artifact from scheduling-insensitive addressing mode optimization to reduce resource pressure on critical AAU units in SC140.
2. There is no memory (store-load) dependence along this loop-carried true dependence.

When these two conditions are observed in a given recurrence circuit, we find that loop-carried true dependence edge can be safely removed by introducing additional induction variable that clones the behavior of an existing induction variable. As an illustration, the loop-carried true dependence in Figure 2(a) can be removed in following steps:

1. Allocate one additional register to replicate induction variable r1 and initialize it at the loop preheader. For this step, assuming that r10

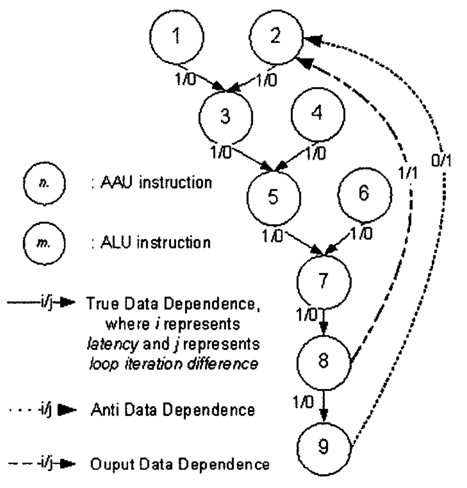
```
...
for (i=0; i<=bound; i++) {
   L_sum = L_mac(L_Round,pswVOld[i],pswQntRc[j]);
   L_sum = L_mac(L_sum,pswVOld[-i],pswQntRc[j]);
   L_sum = L_mac(L_sum,pswPold[i],pswQntRcSqd[j]);
   L_sum = L_msu(L_sum,pswPOld[i],SW_MIN);
   pswPNew[i] = extract_h(L_sum);
}
...
```

(a) A candidate loop from half rate GSM

	SC140 Instruction	Comments
1.	move.f (r11)+,d8	Load *pswVOld[i]* to **d8** and postincrement the array index by 2 bytes
2.	move.l #32768,d10	Load *L_ROUND*, which is 0x8000, to **d10**
3.	mac d8,d5,d10	**d10** <- **d10** + (*pswVOld[i]*pswQntRc[j]*)
4.	move.f (r13)-,d9	Load *pswVOld[-i]* to **d9** and postdecrement the array index by 2 bytes
5.	mac d14,d9,d10	**d10** <- **d10** + (*pswVOld[-i]*pswQntRc[j]*)
6.	move.f (r14)-,d12	Load *pswPOld[i]* to **d12** and postdecrement the array index by 2 bytes
7.	mac d3,d12,d10	**d10** <- **d10** + (*pswPOld[i]*pswQntRcSqd[j]*)
8.	mac -d3,d12,d10	**d10** <- **d10** - (*pswPOld[i]*pswQntRcSqd[j]*)
9.	move.f d10,(r4)+	Store **d10**, which is *L_sum*, to *pswPNew[i]* and postincrement the array index by 2 bytes

▢ *n.* AAU instruction ▢ *m.* ALU instruction

(b) An assembly code of figure (a)



1
2
3
4
5
6
7
8
9
1/0
1/1
0/1
n. : AAU instruction
m. : ALU instruction
i/j True Data Dependence, where i represents latency and j represents loop iteration difference
i/j Anti Data Dependence
i/j Ouput Data Dependence

(c) A data dependence graph of figure (b)

Figure 3 C Code fragment from half-rate GSM and corresponding loop body in SC140

register is available, place one copy instruction tfra r1, r10 to the loop preheader as shown in Figure 2(b).

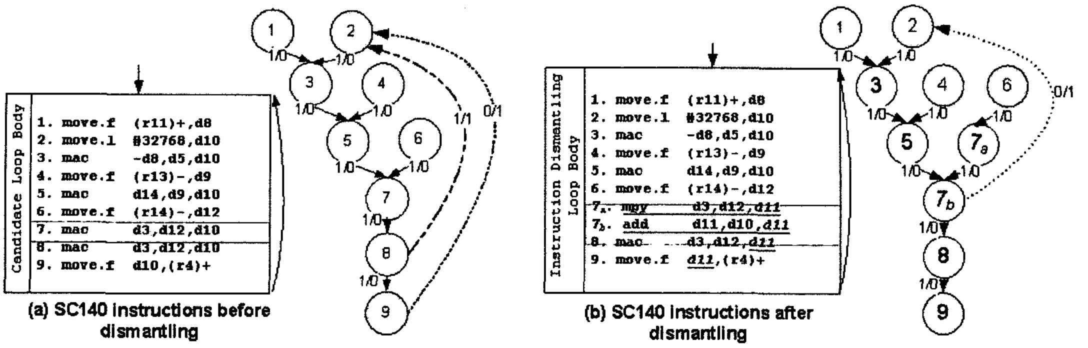
2. Place one additional operation to clone r1 with additional induction variable r10 prior to the update of r1 value. Note that, to minimize resource pressure on AAU units, post increment addressing mode is exploited for the 1st instruction[4], as shown in Figure 2(b).
3. Finally, update the original loop-carried true dependence by making the replicated value being referenced instead. Since the 1st instruction is already amended to reference cloned value r10, no additional change is required.

As a result of this transformation, the original loop-carried true dependence is removed. By applying cloning to other RecMII=6 recurrence circuit that exists in Figure 1, MII is reduced from 6 to 4. Without any modification to an existing modulo scheduler, higher loop initiation rate as 4 is effectively achieved.

For iterative modulo scheduling, candidate loop II is initially set equal to MII, which is computed as follows. First, each iteration of branch-free loop body shown in Figure 3 requires 4 units of ALU and 5 units of AAU, and SC140 multi-issue DSP can supply at most 4 units of ALU and 2 units of AAU per cycle. Thus, ResMII is 3, which is max $(\left\lceil \frac{4}{4} \right\rceil, \left\lceil \frac{5}{2} \right\rceil)$. Second, according to DEFINITION 6, RecMII for half-rate GSM is 5 and corresponding RecMII recurrence circuits are depicted in Figured 4(a). Since MII is max(ResMII, RecMII), *II* is initially set to 5 for modulo scheduling.

For analysis, consider two loop-carried data dependences in Figure 4(a). First, the dependence from the 9th back to the 2nd instructions is anti since d10 value claimed from the 9th instruction is generated by the 2nd instruction. Second, the dependence from the 8th back to the 2nd instructions is output since both instructions store results to

4)Since the memory stride between the 1st and 9th instructions differs by two bytes, indexed post increment addressing mode is selected.



```
Candidate Loop Body
1. move.f  (r11)+,d8
2. move.l  #32768,d10
3. mac     -d8,d5,d10
4. move.f  (r13)-,d9
5. mac     d14,d9,d10
6. move.f  (r14)-,d12
7. mac     d3,d12,d10
8. mac     d3,d12,d10
9. move.f  d10,(r4)+
```

(a) SC140 instructions before dismantling

```
Instruction Dismantling Loop Body
1. move.f  (r11)+,d8
2. move.l  #32768,d10
3. mac     -d8,d5,d10
4. move.f  (r13)-,d9
5. mac     d14,d9,d10
6. move.f  (r14)-,d12
7a. mpy     d3,d12,d11
7b. add     d11,d10,d11
8. mac     d3,d12,d11
9. move.f  d11,(r4)+
```

(b) SC140 instructions after dismantling

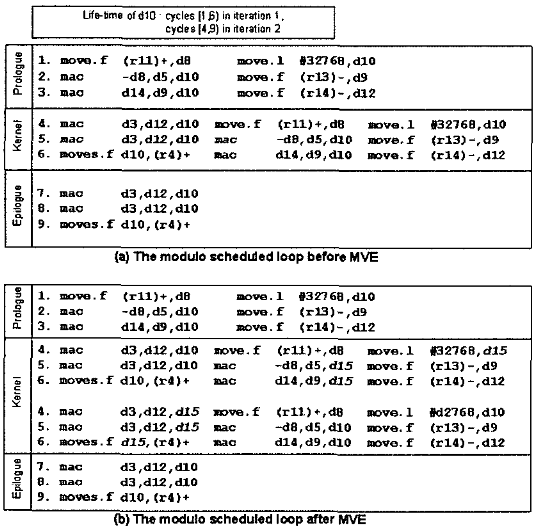
Figure 4 RecMII=5 Circuit in half rate GSM candidate loop

d10. Due to these two RecMII=5 recurrence circuits, half-rate GSM candidate loop fails to be modulo scheduled with II smaller than 5.

3.2.1 Modulo Variable Expansion to relax Ex-RecMII due to false dependences

To eliminate two excessive loop-carried false dependences in Figure 4(a), we apply Modulo Variable Expansion (MVE) proposed by T. Gary and et. al [18]. Figure 5(a) shows loop kernels before and after MVE; loop kernel in Figure 5(b) achieves II=3 modulo schedule by unrolling loop kernel in Figure 5(a) once. The operation is as follows.

- Identify live too long variables which build loop carried dependences by their long life time cross loop iterations. This step is described in Section 4.2. d10 is live too long variable in Figure 5.



Life-time of d10 : cycles [1,6] in iteration 1, cycles [4,9] in iteration 2

```
Prologue
1. move.f  (r11)+,d8        move.l  #32768,d10
2. mac     -d8,d5,d10       move.f  (r13)-,d9
3. mac     d14,d9,d10       move.f  (r14)-,d12
Kernel
4. mac     d3,d12,d10  move.f  (r11)+,d8    move.l  #32768,d10
5. mac     d3,d12,d10  mac     -d8,d5,d10   move.f  (r13)-,d9
6. moves.f d10,(r4)+   mac     d14,d9,d10   move.f  (r14)-,d12
Epilogue
7. mac     d3,d12,d10
8. mac     d3,d12,d10
9. moves.f d10,(r4)+
```

(a) The modulo scheduled loop before MVE

```
Prologue
1. move.f  (r11)+,d8        move.l  #32768,d10
2. mac     -d8,d5,d10       move.f  (r13)-,d9
3. mac     d14,d9,d10       move.f  (r14)-,d12
Kernel
4. mac     d3,d12,d10  move.f  (r11)+,d8    move.l  #32768,d15
5. mac     d3,d12,d10  mac     -d8,d5,d15   move.f  (r13)-,d9
6. moves.f d10,(r4)+   mac     d14,d9,d15   move.f  (r14)-,d12

4. mac     d3,d12,d15  move.f  (r11)+,d8    move.l  #d2768,d10
5. mac     d3,d12,d15  mac     -d8,d5,d10   move.f  (r13)-,d9
6. moves.f d15,(r4)+   mac     d14,d9,d10   move.f  (r14)-,d12
Epilogue
7. mac     d3,d12,d10
8. mac     d3,d12,d10
9. moves.f d10,(r4)+
```

(b) The modulo scheduled loop after MVE

Figure 5 MVE for half rate GSM candidate loop

- Determine unrolling factor for modulo variable expansion: RecII can potentially be lowered from 5 to 3 by unrolling loop body twice in Figure 5. The unrolling factor is decided by that minimum initiation interval divides life time of live too long variable. For instance, life time of d10 is 5, and II is 3 decide by Definition 9. So, unrolling factor is 2 in this code. When a candidate loop is unrolled, life-time of variable can be separated by register renaming. However, in order to rename all live too long values if there are many long variables, additional data registers are required which can be beyond the SC140 VLES DSP can support. Thus, check whether enough data registers exists for the required renaming.
- Check resource constraint for unroll factor: Unrolling factor requires additional units of ALU and units of AAU, and SC140 VLES DSP can supply at most 4 units of ALU and 2 units of AAU per cycle. Thus, if there are no available functional units for loop unrolling, modulo variable expansion should stop.
- Finally, unroll loop body, rename data value generated from the beginning of the new loop iteration with available data registers, and change the loop counter to be incremented by unrolling factor. However, since loop iteration count is kept being reset from outside of this loop, the original candidate loop requires to be strip-mined to execute the modulo scheduled loop kernel in hardware looping mode. Since there are two additional candidate loops like this in the same loop nest, code size increase required for MVE alone was more than 200%.

3.2.2 DISMANTLING to relax Ex-RecMIIdue to false dependences

To avoid code size increase from MVE, we attempted to split excessive lifetimes of registers by moving data values. This technique is known as a software way of emulating rotating registers [19]. However, data move instruction sometimes has one undesired side effect, which resets scaling bit in Status Register (SR) [4]. To ensure output bit exactness, we instead divide excessive lifetime values by dismantling destructive instructions that require to use same register for source and destination.

As an illustration, the 7th mac instruction in Figure 4(a) can be dismantled into 7_a^{th} and the 7_b^{th} instructions followed by a proper register renaming as shown in Figure 4(b). When this modification is made, the loop-carried anti dependence from the 9th back to the 2nd instruction and the loop-carried output dependence from the 8th back to the 2nd instruction in Figure 4(a) are both eliminated; Figure 4(b) depicts dismantled recurrence circuits. As a result of dismantling, the original RecMII=5 for half rate GSM candidate loop is effectively lowered to 3 and therefore, the higher loop initiation rate is achieved without loop kernel unrolling and strip-mining.

4. PREPROCESSING STRATEGY FOR EFFECTIVE MODULO SCHEDULING

In order to ease task of cloning and dismantling, optimizing compiler puts a candidate loop body such that intra-loop false dependences are removed whenever possible. In that setting, for a given candidate loop, the main preprocessing task is to reduce excessive RecMII (Ex-RecMII) as follows:

- Eliminate loop-carried true dependences of Ex-RecMII recurrence circuit by register cloning.
- Relax loop-carried false dependences of Ex-RecMII recurrence circuit by dismantling.

Note that cloning and dismantling transformations do not come for free. The cloning inevitably increases register pressure and/or resource constraints due to additional operations that are required to replicate induction variables. The dismantling also increases both register pressure and resource constraints due to additional instructions that are required to split destructive instructions.

Nevertheless, considering RecMII as a dominant limiting factor that fails candidate loops to be modulo scheduled, the increase in ResMII that makes Ex-RecMII decrease is always beneficial. Since the greater decrease in Ex-RecMII means the better II, the real preprocessing challenge is how to effectively reduce Ex-RecMII with cloning and dismantling subjected to the constraint of ResMII increase. Therefore, the mission of preprocessing strategy for effective modulo scheduling is defined as follows.

For Ex-RecMII of a candidate loop, find an optimal application sequence of cloning and dismantling that reduces Ex-RecMII by the largest degree.

4.1 ALGORITHM

Since DDG of a candidate loop can contain exponentially many recurrence circuits, our optimal application sequence finding problem is reduced to a bounded resource allocation problem, which is NP-complete. In response to this complexity, following two heuristics are employed:

Heuristic 1 Cloning and dismantling are considered only when additional operations for these transformations are guaranteed not to increase Ex-RecMII.

Heuristic 2 An optimal application sequence of cloning and dismantling is sought only for Ex-RecMII recurrence circuits under the constraint of Heuristic 1.

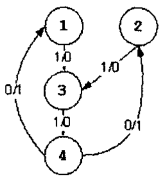
In particular, Heuristic 1 is designed to guarantee that cloning and dismantling never make loop schedule worse. As an illustration, consider Figure 6(a) that contains two RecMII=3 recurrence circuits rc_A=(1-3-4) and rc_B=(2-3-4). When the 3rd mac instruction is indiscretionally dismantled for rc_A, the dependence height of rc_B increases by 1 as shown in Figure 6(b) and as a result, RecMII of the candidate loop increases by 1. Heuristic 1 serves as a safeguard against this detrimental case.

To instrument the heuristics described above for desired optimal application sequence, we exploit divide-and-conquer algorithm. Since Ex-RecMII is

```
1.  move.f  (r7),d8
2.  move.f  (r8)+,d6
3.  mac     d6,d7,d8
4.  move.l  d8,(r12)+

RecMII=2 recurrence circuits A
and B:

Critical Path A:   1->3->4
Critical Path B:   2->3->4
```

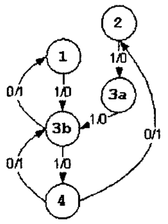


(a) Before Dismantling

```
1.  move.f   (r7)+,d8
2.  move.f   (r8)+,d6
3a. mpy      d6,d7,d15
3b. add      d8,d15,d2
4.  move.l   d2,(r12)+

RecMII=3 recurrence circuit B:

Critical Path B:   2->3a->3b->4
```



(b) After Dismantling RecMII=3 recurrence circuit A

Figure 6 An example for the Heuristic I

the dominant limiting factor for a modulo schedule, such RecMII recurrence circuits are first identified as a divide step. Section 4.2 describes an algorithm that effectively finds all Ex-RecMII recurrence circuits in a given candidate loop. Second, Heuristic 2 is used as a conquer step and iterative worklist algorithms, which implement Heuristic 2, are described in Section 4.3. This divide and conquer steps are integrated into an unified framework, which described in Section 4.4. Since the preprocessing, which capitalizes on this unified framework, iterates until there is no further change in Ex-RecMII, search space for a desired optimal application sequence is typically exhausted.

4.2 DIVIDE STEP: Finding Ex-RecMII Recurrence Circuits

To identify all recurrence circuits which account for Ex-RecMII in a given candidate loop, we use Tiernan's algorithm that finds Elementary Circuits (EC) of a Data Dependence Graph (DDG) [20]:

1. When Tiernan's algorithm confirms a non-trivial recurrence circuit rc=($inst_1$, $inst_2$, ..., $inst_n$), each dependence arc (edge) in rc is retrieved from DDG to estimate II_{rc} according to DEFINITION 5.
2. The confirmed rc is added to EC list, which is sorted in descending order using II_{rc} as key. For this step, C data structures in Figure 7 are used; struct EM_CT for a recurrence circuit and struct LIST for EC list.

```
/* Elementary Circuit (Recurrence Circuit) */
struct EM_CT {
    unsigned char   head;   // Inst number: head of the circuit
    unsigned char   tail;   // Inst number: tail of the circuit
    unsigned char II_rc ;   // DEFINITION 5
    unsigned char *   P;    /* Elementary Path building array
                                 used in Tiernan's algorithm */
    bvect          circuit;  /* Circuit representation
                                 in bit vector */
    struct LIST * i_ecs;   /* Other recurrence circuits that
                                 intersect with this circuit */
    struct LIST * p_ecs;   /* Other recurrence circuits
                                 that are properly contained */
    unsigned int p_inst;   /* Inst number: where the desired
                                 dismantling will be placed*/
    int             status;  // DRYRUN|DONE|CLONE|DISMANTLE
};
/* List of Ex-RecMII Recurrence Circuit */
static struct LIST *ECs;
```

Figure 7 Elementary Circuit (EC) and EC list C data structures

3. Prior to rc insertion, for each rc_i in EC list, the following two fields for both rc_i and rc are updated.
   - i_ecs: set of intersecting recurrence circuits, and

- p_ecs: set of properly contained recurrence circuits.

These two fields are used to implement **Heuristic II** in following conquer step. In particular, to perform set related operations in a constant time, we additionally represent each recurrence circuit as a bit; for this, **bvect** data type is added to EM_CT recurrence circuit data structure. When circuit confi rmation process completes, II_{rc} of the head node in EC list is RecMII, according to DEFINITION 6. If RecMII > ResMII, then all subsequent nodes, which share the same value of II_{rc} in EC list are Ex-RecMII circuits.

4.3 CONQUER STEP: Iterative Worklist Algorithms

For a set of Ex-RecMII recurrence circuits, which were obtained from previous divide step, the task of **Heuristic II** requires finding an optimal cloning and dismantling application sequence.

Theorem 1 Unless all loop-carried true dependences of Ex-RecMII circuits are eliminated, RecMII of a candidate loop cannot be reduced by dismantling.

Proof: Consider a loop with circuit set C={c_1,c_2,c_3, ..., c_n}, ExRecMII of the loop is |c_k| which is consisted of flow and anti dependence circuits simultaneously. Let's apply dismantling technique to this loop. Assuming the largest anti circuit c_k is composed by a backedge from a read operand w to a write operand w while it has iteration distance more than 1. dismantling performs to rename the w of an instruction on c_k to an unused register using dismantled instruction for lowering ExRecMII. The w used by other instructions located under renamed operand is also renamed to the operand. However, all w operands affected in the flow dependence circuit must be also renamed to the new one since there is naming recurrence on the flow circuit. As a result, dismantling can not resolve the recurrence circuit in this case. □

Corollary 1 Cloning must be applied prior to dismantling to reduce Ex-RecMII.

Corollary 2 Order of circuit selection for cloning does not affect optimal dismantling sequence in Ex-RecMII recurrence circuits.

In order to implement the task of Heuristic II, Corollaries 1 and 2, which deduced from Theorem 1, are used as follows:

1. Partition the set of Ex-RecMII recurrence circuits into c-worklist and dworklist, where c-worklist is a set of circuits whose loop-carried dependences are true and d-worklist is a set of recurrence circuits whose loop-carried dependences are false.
2. Find an optimal cloning sequence from c-worklist with an iterative worklist algorithm in Figure 8, which builds OptSeqECs list [21]. According to Corollary 2, algorithm in Figure 8 simply chooses the head circuit from c-worklist and appends it to OptSeqECslist.

```
struct LIST * c_worklist = {rc1,rc2,...,rci};
struct LIST * d_worklist = {rci+1,rci+2,...,rcn};
int CurResMII = ResMII;  /* Current Resource bound    */
struct BITSET *   NotAvailRegs = set to bitwise union of
                          1. set of registers which are coming
                             in alive to the candidate loop block
                          2. set of registers which are defined
                             within the candidate loop block
struct List *OptSeqECs = NULL;
int possible;
/* Iterative worklist algorithm for clone worklist */
WHILE (c_worklist is not empty) DO {
  struct EM_CT *rc = get_head(c_worklist);
  /* precisely estimate the resource and register
     requirement for each Ex-RecMII circuit selection */
  IF (!(possible = can_perform_clone(rc,
                          &CurResMII, NotAvailRegs)))
    RETURN FALSE;
  ELSE {
    append_to_tail(&OptSeqECs, rc);
    c_worklist = c_worklist - {rc};
  }
} /* end_of_WHILE */
/* Continue the iterative work list algorithm for
   dismantle_worklist */
```

Figure 8 Iterative worklist algorithm for c-worklist

3. Find an optimal dismantling sequence from d-worklist. Since dismantling one Ex-RecMII circuit can result in either reducing or stretching other recurrence circuits, these side-effects must be precisely accounted for optimal dismantling circuit selection.
4. Append resulted optimal dismantling sequence to OptSeqECs. Upon completion of steps desribed above, an optimal preprocessing sequence for a set of Ex-RecMIIrecurrence circuits can be obtained from OptSeqECs, if such a sequence exists.

4.3.1 CLONING: Iterative worklist algorithm

For a given recurrence circuit rci in c-worklist, algorithm in Figure 8 (in particular, can perform clone()) determines the following:

1. Applicability of cloning to rc_i, as explained in Section 3.1.1.
2. Availability of resources and registers required for transformation.

When cloning deems applicable to rc_i, the placement of additional operation required for cloning must be identified. Since, as shown in Figure 2(b), insertion of additional cloning operation can be made with no resource pressure increase, the worklist algorithm (in particular, can perform clone()) exploits this merit:

- If increment/decrement operation of a replicated induction variable can be encoded into an existing instruction of rc_i, cloning transformation can be made with no increase in resource pressure. In this case, set p inst to an *instruction, where such encoding is possible.*
- Otherwise, set p_inst to $\left\lceil \frac{II_{rc}}{2} \right\rceil$. Since this helps in reducing total number of Ex-RecMII recurrence circuits for subsequent round of preprocessing, it indirectly reduces overall resource requirements for desired preprocessing.

4.3.2 DISMANTLING: Iterative worklist algorithm

Optimal dismantling sequence in d-worklist can be sought only when the following side-effects are accurately forcasted:

1. The number of circuits in d-worklist, which can be simultaneously dismantled by dismantling rc_i in the same worklist.
2. The prediction whether dismantling rc_i in d-worklist results in stretching RecMII of other recurrence circuits.

The estimation of the side-effects described above poses a tremendous computational challenge since (1) number of circuits, which can be dismantled, varies depending on which instruction in rc_i is selected for dismantling and (2) the selection must be made subjected to the constraint that RecMII in other recurrence circuits must not be stretched.

The algorithm in Figure 9 uses a brute-force approach, which attempts to find an optimal dismantling sequence by iterating over each circuit in d-worklist and its constituent instructions. *However, to reduce the number of necessary* iterations, *i_ecs* and *i_pcs* fields of struct EM_CT, which described in Section 4.2, are exploited. Upon the completion of algorithm described in Figure 9, the desired dismantling sequence can be found in D-Metric if such sequence exists.

4.4 UNIFIED FRAMEWORK: Divide-and-Conquer

To respond to the complexity of finding an optimal cloning and dismantling sequence, our preprocessing problem, we decomposed the original preprocessing task into a set of sub-tasks. Sections 4.2 and 4.3 describe how each of the sub-tasks are implemented.

Figure 10 represents an unified algorithm that integrates each of the sub-tasks to effectively perform cloning and dismantling such that the Ex-RecMII *of a given candidate loop can* be reduced by the largest degree. Since this unified process continues until there is no further change in the Ex-RecMII, most of the search space for the desired optimal preprocessing sequence is typically exhausted.

5. EXPERIMENTAL RESULTS

This section describes the results of a set of experiments to illustrate the effectiveness of the uni fied preprocessing strategy described in Figure 10, which is implemented for the StarCore

SC140 academic compiler backend. The experi-

```
/* Data Structure used for D-Metric */
struct DISMANTLE_ANALYSIS {
  struct EM_CT *d_rc;      // a recurrence circuit in the d_worklist */
  struct LIST * p_ecs;     /* set of all properly contained circuits in
                               the d_worklist for d_rc */
  unsigned int  p_inst; // inst number of d_rc selected for dismantling
  struct LIST * d_metric;/* list of circuits in d-worklistthat can be
                            simultaneously dismantled by dismantling d_rc*/
};
struct LIST * D_Metrics;// list of DISMANTLE_ANALYSIS object
struct EM_CT *rc;          // temporary pointer to iterate d_worklist
unsigned int  BIT;         // index to Instructions[] global array

FOR each rc in d_worklist DO {
   struct DISMANTLE_ANALYSIS *da =
       local_alloc(sizeof(struct DISMANTLE_ANALYSIS));
   da->d_rc = rc;
   da->p_ecs = find_p_ecs_d_worklist(rc);
   WHILE (1) {
     da->p_inst = -1; da->d_metric = NULL;

     FOR each BIT set in rc->circuit DO {
       IF (BIT == rc->head || BIT == rc->tail)
         CONTINUE;
       p_inst_max_D_Metric(da, BIT);
     }
     IF (da->p_inst == -1)
       RETURN (struct LIST *)NULL; // fail
     ELSE IF (heuristic1_preserved(da) == TRUE)
       BREAK;
     ELSE
       unset(rc->circuit, da->p_inst);
   } // end of while (1)
   /* add da to the D_Metric list in descending order using the number of
      recurrence circuits that can be simultaneously dismantled */
   add_list(&D_Metrics, da->d_metric->size);
} /* end of FOR */
RETURN (D_Metrics); // return linked list for D-Metrics
```

Figure 9 Dismantling algorithm that computes D-Metric

```
WHILE (1) {
   int reducible;
   struct EH_CT *rc;

   /* 4.3 DIVIDE STEP         */
   Find_Ex_RecMII_Recurrence_Circuits();
   /* 4.4 CONQUER STEP        */
   Partition_Ex_RecMII_Recurrence_Circuits_into_Two();
   /* 4.4.1 CLONING           */
   IF (!(reducible =
        Iterative_Worklist_Algorithm_for_c_worklist()))
      BREAK;
   /* 4.4.2 DISMANTLING       */
   IF (!(reducible =
        Iterative_Worklist_Algorithm_for_d_worklist()))
      BREAK;

   /* 3.2 and 3.4 Cloning and Dismantling Techniques  */
   FOR each rc in OptSeqECs DO {
     IF (rc->status | CLONE)
        commit_cloning(rc);
     ELSE
        commit_dismantle(rc);
   }
   /* Rebuild the Data Dependence Graph for the next round */
   build_DDG();
}
```

Figure 10 Integrated Preprocessing Algorithm

mental input is a set of candidate loops obtained from DSPStone [22], MediaBench [23], half-rate GSM, enhanced full rate GSM , and other industry signal application kernels. Table 1 lists the benchmarks used for our experiments.

In order to isolate the impacts on performance and code size purely from our preprocessing, two sets of executables for the SC140 multi-issue DSP are produced for the benchmarks listed in Table 1;

- ORIG: fully optimized one with original compiler, and
- PRE: fully optimized one with the revised compiler with our preprocessing proposed.

With these two sets of executables, we measured (1) cycle counts with the StarCore cycle count accurate simulator simsc100, and (2) code size with the StarCore utility tool, sc100-size. The performance improvements (decrease in cycle counts) and code size increase due to our preprocessing were measured in percent, using the formula **((ORIG − PRE)/ORIG)** * 100.

Figure 11 reports the performance improvements achieved by applying the unified algorithm in Figure 10, which is based on cloning and dismant-

Table 1 Benchmarks used in the Experiments

Program	*Acronym*	*Description*
FIR2DIM	**FD**	2 dimensional Finite Response Filter
LMS	**LMS**	Filter benchmarking in DSP stone
Convolution	**Conv**	convolution
complex FFT	**ComFFT**	128 point complex FFT
FFT	**FFT**	Integer stage scaling FFT
Biquad_N_section	**BiNsec**	One IIR biquad
N_complex_updates	**Ncomup**	Complex multiply
Matrix	**Matrix**	Generic matrix multiply
FIR	**FIR**	Complex FIR filter
Matrix 1×3	**Mat 1×3**	1×3 matrix
IIR	**IIR**	IIR filter
Lattice synthesis	**Latsyn**	Typical DSP multiply two vector operation
Panama cryptographic module	**Pcrypto**	Panama stream/hash module
GSM	**GSM (sc,af,ut,dec,ad,sy)**	v_search, aflatRecursion, utcount, decode, add, syn_fil modules from Global System for Mobile telecommunication

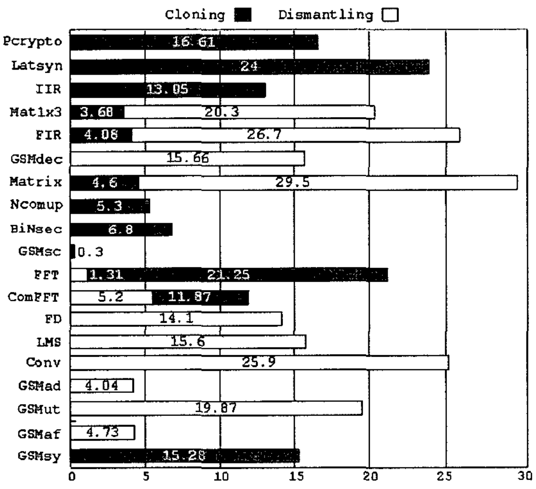


Figure 11 Percent wise performance improvement (number of cycles reduction) compared to the original

ling techniques respectively. The overall performance improvement from the preprocessing ranges from 0.3% to 29.5%, and the average. performance improvement is 12.9%. Considering there is no modi fication made to the existing iterative modulo scheduler and the performance comparison is made to highly optimized SC140 DSP code, the performance gain from our preprocessing was impressive. In particular, the performance improvements on Mat1x3, FIR, FFT and ComFFT benchmarks were brought to our attention, since

1. the iterative application of cloning followed by dismantling for an existing modulo scheduler can deliver huge performance gain by effectively reducing the Ex-RecMIIs of a candidate loop, and
2. the preprocessing strategy described in Figure 10 can detect and exploit such opportunities for an effective modulo scheduling.

Note that none of the benchmarks in Figure 11 reports the performance degradation. This is not a mishap, but due to the fact that our algorithm is designed to apply cloning and dismantling techniques only when the additional operations for our preprocessing can be placed in non-RecMII recurrence circuits.

Figure 12 reports the code size increase due to the unified algorithm as described in Figure 10. Since cloning and dismantling reduces the Ex-RecMIIs of a given candidate loop, the existing modulo scheduler discovers instruction level parallelism across more loop iteration boundary and as a result, achieves a better modulo schedule. Considering the size of the prologue and epilogue grow proportionally as more loop iterations of the candidate loop get overlapped for a final schedule, the code size increase is unavoidable. However, we also observed that the existing modulo scheduler can find a better loop schedule for a given number of loop iteration boundaries when our preprocessing is applied. This is the reason why our preprocessing to IIR, GSMdec, GSMad and GSMsy

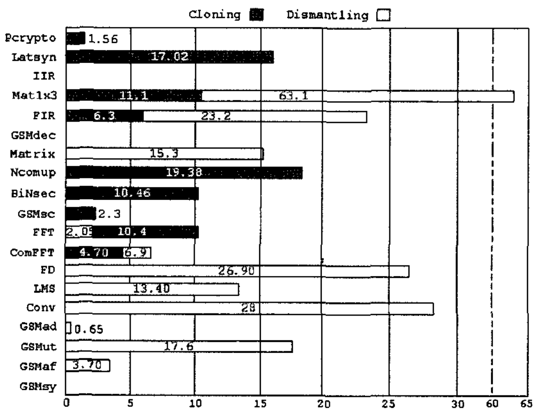


Figure 12 Percent wise code size increase compared to the original

benchmarks, reports significant performance improvements with negligible increase in code size.

For the benchmarks listed in Figure 12, the overall code size increase from the preprocessing strategy ranges from 0% to 63.1%, and the average increase is 13.99%. However, note that the benchmarks in Table 1 are critical loop kernels which typically account for 5%-10% of entire application code size. By carefully applying the preprocessing to the mission critical loops with profiling, the overall code size increase can be moderated.

6. RELATED WORK

The detrimental effects of Ex-RecMII from loop-carried dependences were also noticed by Lam. In particular, she observed that Ex-RecMII is typically caused between a value being defined by a high latency operation (e.g., multiplication and memory load) and its subsequent use. To effectively lower this Ex-RecMII, Lam pioneered a compiler technique, referred to as Modulo Variable Expansion (MVE), that removes loop-carried anti and output dependences in recurrence circuits [24]. Since MVE achieves the desired removal with loop unrolling followed by register renaming, high loop unrolling factor might incur tremendous increase in code size and register pressure. Another drawback of this scheme is that those candidate loops which execute for a multiple number of times the unrolling factor can only be properly accommodated. To overcome this problem, either peeling candidate loops for some number of loop iterations or adding a branch out of the unrolled loop body are required [25].

To duplicate the effect of MVE without loop unrolling, Huff proposed an innovative rotating register files as an architectural feature in a hypothetical VLIW processor similar to Cydrome's Cydra 5 [26]. Since Huff technique still requires a large number of architected rotating registers to support MVE without code expansion, Tyson and et al. ameliorated Huff technique with register queues and rq-connect instruction [18]. In their technique, register queues share a common namespace with physical register files. As a consequence, the architected rotating register space is no longer a limiting factor.

However, contrary to high performance VLIW machines which were the target for rotating register files and register queues, most of operations in SC140 multi-issue DSP typically require no latency. Due to this lack of operation latency, the number of *registers whose lifetimes extend IIis expected* fairly low compared to that of high performance VLIW machines. According to our benchmarking with various signal processing kernels, one register on average is reported whose lifetime exceeds II. We believe that this small number is fairly difficult to justify rotating register files for multi-issue DSPs since adding extra register file and creating read-write ports to the file are known to be quite expensive. In addition, this small number is neither good enough to justify the internal register map table and additional hardware logics for register queues to emulate register renaming feature in rotating register files.

To duplicate the effect of MVE without loop unrolling and rotating register files, Stotzer and Leiss exploited TMS320C6x DSP microarchitecture which has fixed operation latencies and no pipeline interlocks. The operations with latency > 1 are referred in-flight until they complete execution. TMS320C6x DSP architecture allows multiple in-flight operations to have pending writes to the same register. With this in-flight feature, Stotzer and Leiss demonstrated that they can emulate Huff's slack scheduling without requiring rotating register files [26,9]. However, there are two major drawbacks in duplicating the effect of MVE with in-flight feature. First, note that the longest in-flight operation in TMS320C6x is a branch whose latency is 6. Since modulo scheduling potentially overlaps 6 iterations of a candidate loop the code size increase for prologue and epilogue can be quite huge. Second, note that TMS320C6x does not allow interrupts to be serviced when a branch operation is in-flight. When II of a modulo scheduled loop is less than 6, there is always a branch which is in-flight during the loop execution and thereby, system response time can be unduly delayed.

The work reported in this paper is an extension of our early work. In [27], we proposed an approach that resolved only false dependencies by branch and bound scheme with formal definition of the dependence circuit problem. That work differs from our current work since it focused only on a false dependence.

7. CONCLUSION

To address resource under-utilization problem for multi-issue DSPs, this paper first presents two new transformations, cloning and dismantling, that reduce excessive RecMII with a partial aid from under-utilized functional resources. Second, since the greater decrease in Ex-RecMII means the better II, this paper presents a novel preprocessing strategy that reduces Ex-RecMII by the largest degree with cloning and dismantling subjected to the constraint of ResMII increase. The proposed preprocessing techniques and strategy are implemented for SC140 multi-issue DSP compiler. As a result of implementation, 12.9% average runtime improvement was reported for benchmarks in Table 1; this runtime improvement was made at the expense of 13.99% average code size increase. Considering there is no modification made to existing modulo scheduler and performance comparison is made to highly optimized SC140 DSP code, the gain was impressive. However, we also observe that Modulo Variable Expansion (MVE) strategy described by M. Lam and et. al. can be beneficial since one register on an average is reported whose lifetime exceeds II for benchmarks listed in Table 1. Future experimentation may assess how our unified preprocessing strategy can be orchestrated with MVE as a postprocessing technique for effective modulo scheduling.

REFERENCES

[1] P. Labsley, J. Bier, and E. Lee: DSP Processor Fundamentals -Architecture and Features. *In IEEE Press*, ISBN 0-78033-405-1, 1996. Sep 1996.

[2] J. Eyre and J. Bier: The Evolution of DSP Processors. *In A BDTI White Paper, ACM SIGARCH Computer Architecture News*, 2000.

[3] Blackfin processor compiler and code density. http://www.analog.com.

[4] StarCore, Inc.: SC140 DSP Core Reference Manual. Atlanta, GA, 2001.

[5] Texas Instrument, Inc.: Code generation tools: compile tools for TMS320C6000 and TMS320C5000 DSPs. http://dspvillage.ti.com.

[6] E. Tan and W. Heinzelman: DSP architectures: past, present and futures. In *ACM SIGARCH Computer Architecture News*, Vol.31, No.3, pages 6-19, June 2003.

[7] B. Rau and D. Glaser: Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14 Ann. Microprogramming Workshop*, Nov 1981.

[8] B. Rau: Iterative modulo scheduling. In HP Laboratories Technical Report, HPL94115, Nov 1995.

[9] E. Stotzer and E. Leiss: Modulo Scheduling for the TMS320C6x VLIW DSP Architecture. In *Proceedings of the SIGPLAN'99 Workshop on Languages, Compilers, and Tools for Embedded Systems*, May 1999.

[10] Cosy DSP Compiler Development System. www.ace.nl/compiler/DS CoSy DSP.pdf.

[11] Code Warrior for StarCore DSP. http://www.testechelect.com/metrowerks/starcore.html.

[12] Green Hills: Embedded software development tools -StarCore Family. www.ghs.com/product/starcore development.html.

[13] J. Sias, H. Hunter, and W. Hwu: Enhancing loop buffering of media and telecommunications applications using low-overhead predication. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, Dec 2001.

[14] D. Lavery and W. Hwu: Unrolling-Based Optimizations for Modulo Scheduling. In *Proceedings of the 28th annual international symposium on Microarchitecture*, page 327-337, 1995.

[15] V. Sarkar: Optimized Unrolling of Nested Loops. In *International Journal of Parallel Programming*, Vol.29, No.5, Oct 2001.

[16] T. Halfhill: Motorola enhances StarCore DSP,

SC140e core offers new instructions, caches, and task protection. In *Microprocessor Report, INSTAT/MDR*, www.MPRonline.com, Oct 20, 2003.

[17] G.R.Uh, Y. Wang, D. Whalley, and et al.: Compiler Transformations for Effectively Exploiting Zero Overhead Loop Buffer. *Software-Practice & Experience*, Vol 35, pages 393-412, 2005.

[18] G. Tyson, M. Smelyanskiy, and E. Davidson: Evaluating the Use of Register Queues in Software Pipelined Loops. In *IEEE Transactions on Computers*, Vol.50, No.8, pages 769-783, Oct 2001.

[19] TMS320C6000 Optimizing C Compiler Tutorial. In Literature Number: SPRU425A, August 2002.

[20] J. Tiernan: An efficient search algorithm to find the elementary circuits of a graph. In *Communications of the ACM*, pages 12-35, Dec 1970.

[21] S. Muchnick: Advanced Compiler Design Implementation. In Morgan Kaufmann Publishers, ISBN 1-55860-320-4, 1997.

[22] V. Zivojnovic, J. Velarde, C. Schager, and H. Meyr: DSPStone -A DSP oriented Benchmarking Methodology. In *Proceedings of International Conference on Signal Processing Applications and Technology*, 1994.

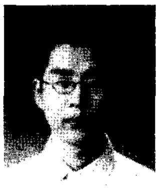
[23] C. Lee, M. Potkonjak, and W. Smith: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov 1997.

[24] M. Lam: Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June, 1988.

[25] H. Allan, B. Jones, M. Lee, J. Allan: Software Pipelining. In *ACM Computing Surveys*, Vol.27, No.3, Sep 1995.

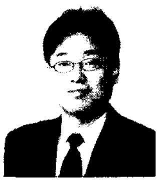
[26] R. Huff: Lifetime-Sensitive Modulo Scheduling. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June, 1993.

[27] D. Cho, Y. Paek: Instruction Reselection for Iterative Modulo Scheduling on Multi Issue DSPs. In *Proceedings of the 1th international workshop on embedded software optimization, Aug 2006.*



조 두 산

2001년 한국외국어 대학교 디지털 정보 공학 학사. 2003년 고려대학교 전기전자 공학 석사. 2003년~현재 서울대학교 전기컴퓨터 공학 박사과정. 관심분야는 메모리 시스템 설계 및 최적화, 저전력, HW/SW 동시설계



백 윤 홍

1988년 서울대학교 컴퓨터공학 학사. 1990년 서울대학교 컴퓨터공학 석사. 1997년 University of Illinois Urbana Champaign 전산학 박사. 1997년~1999년 New Jersey Institute of Technology 조교수. 1999년~2002년 KAIST 조교수. 2002년~2003년 KAIST 부교수. 2003년~현재 서울대학교 부교수 관심분야는 고속 ASIP 프로토타이핑을 위한 EDA툴, 재거냥성 최적화 컴파일러, 임베디드 프로세서 설계