Preprocessing Strategy for Effective Modulo Scheduling on Multi-issue Digital Signal Processors^{*}

Doosan Cho², Ravi Ayyagari¹, Gang-Ryung Uh¹, and Yunheung Paek²

¹ Computer Science Dept., Boise State University

² Electrical and Computer Science Dept., Seoul National University

Abstract. To achieve high resource utilization for multi-issue *Digital Signal Processors* (DSPs), production compilers commonly include variants of the *iterative modulo scheduling* algorithm. However, excessive cyclic data dependences, which exist in communication and media processing loops, often prevent the modulo scheduler from achieving ideal loop initiation intervals. As a result, replicated functional units in multi-issue DSPs are frequently underutilized. In response to this resource underutilization problem, this paper describes a compiler preprocessing strategy that capitalizes on two techniques for effective modulo scheduling, referred to as *cloning1* and *cloning2*. The core of the proposed techniques lies in the direct relaxation of cyclic data dependences by exploiting functional units which are otherwise left unused. Since our preprocessing strategy requires neither code duplication nor additional hardware support, it is relatively easy to implement in DSP compilers. The strategy proposed has been validated by an implementation for a StarCore SC140 optimizing C compiler.

1 Introduction

As communication and media signal processing applications get more complex, system designers seek programmable high performance fixed-point Digital Signal Processors (DSPs). Recent *multi-issue* high performance DSPs¹ are designed to meet such demand by providing (1) multiple functional units, (2) advanced issue logic that allows a variable number of instructions to be dispatched in parallel, and (3) optimizing compilers that automatically tune C algorithms for performance [8,17].

In particular, to exploit the multiple functional units available in multi-issue DSPs, optimizing compilers commonly use a software pipelining strategy. Software pipelining is a global loop scheduling concept which exploits instruction level parallelism across loop iteration boundaries. Optimizing C compilers for multi-issue DSPs commonly adopt variants of the *iterative modulo scheduling* pioneered by Rau and Glaser [2]. Although existing iterative modulo scheduling approaches [7,17] are proven to be effective, excessive cyclic data dependences, which are frequently observed in communication and

^{*} Research is supported in part by NASA Idaho EPSCoR, Intel Corporation, Korea MIC grant #A1100-0501-0004, Korea MOST grant #M103BY010004-05B2501-00411, Korea IITA grant #IITA-2005-C1090-0502-0031, and NANO IP/SOC Promotion Research Program.

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

media processing loops, restrict modulo scheduling quality [13]. As a result, replicated functional units in multi-issue DSPs are often left underutilized.

To address this resource utilization problem, the objective of this paper are twofold: (1) analyzing the nature of the data dependences existing in various signal processing applications, and (2) engineering an effective compiler preprocessing strategy for multi-issue DSPs to help an existing modulo scheduler achieve a high quality loop schedule. For this, the paper describes our preprocessing that directly relaxes excessive cycle data dependences with two techniques, referred to as *cloning1* and *cloning2*. Since these two techniques exploit underutilized functional resources, neither code duplication nor additional hardware support are required, and therefore, it is relatively amenable to implement in DSP compilers. To measure the feasibility and effectiveness of our preprocessing strategy for multi-issue DSPs, the StarCore SC140 DSP processor is used as the representative.

2 Motivation: Excessive RecMII

We formally define the commonly used loop scheduling terms in this paper; for the definitions of other modulo scheduling terms, consult [2].

Definition 1. A **candidate loop** for an iterative modulo scheduler is the loop with the branch-free body that can run in DSP hardware looping mode [10].²

Definition 2. A **recurrence circuit** is a data dependence circuit that exists in a Data Dependence Graph (DDG), which is formed from an instruction to an instance of itself.

Definition 3. **ExRecMII** is the difference between **RecMII** and **ResMII**, iff $\text{RecMII} \geq \text{ResMII}$.

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

2.1 Loop-Carried True Dependence

As the first example of **ExRecMII**, consider the C code fragment shown in Figure 1(a) that implements the *Fast Fourier Transformation* (FFT) algorithm. For the shaded candidate loop body in Figure 1(a), the SC140 optimizing 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 the candidate loop in Figure 1(b) is initially set equal to **MII**, which is computed as follows. First, each iteration of the branch-free loop body shown in Figure 1(b) requires 7 units of *Arithmetic and Logic Unit* (ALU) and 6 units of *Address Arithmetic Unit* (AAU), and the SC140 multi-issue DSP can supply at most 4 units of ALU and 2 units of AAU per cycle³. Thus, **ResMII** is 3, which is $\max(\lceil \frac{7}{4} \rceil, \lceil \frac{6}{2} \rceil)$. Second, this candidate loop body contains several data

² The optimizing C compiler performs if-conversion to allow more loops to be modulo scheduled.

³ To support high computing needs, StarCore 140 has 4 ALU units and 2 AGU units.[17]

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

dependence recurrence circuits. $RecMII$ is 6 for the FFT and one such circuit is depicted in Figure 1(c).⁴ Since $MII = \max(ResMII, RecMII)$, MII is initially set to 6 for a modulo schedule.

For analysis, consider the loop-carried data dependence in Figure 1(c). 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, the dependence chain from the 1st instruction down to the 9th instruction is transitively **true**. This type of cyclic **true** dependence is often created by the compiler when a source address for a computation is used as the destination address to store the result of the computation, which is a very common pattern in DSP applications. Due to this cyclic **true** dependences, the FFT candidate loop fails to be modulo scheduled since MII of 6 is the ratio which can be achieved by local acyclic scheduling.

2.2 Loop-Carried False Dependence

As the second example for $ExRecMII$, consider the C code fragment in Figure 2(a) that implements the half-rate *Global System for Mobile communication (GSM)* algorithm. For the candidate loop body in Figure 2(a), that uses the *European Telecommunications Standards Institute (ETSI)* compliant C macros [6], the optimizing compiler produces highly optimized assembly code as shown in Figure 2(b), which is yet to be modulo scheduled.

⁴ The other $RecMII$ circuit is omitted since the type of loop-carried data dependence is same as that of the circuit in Figure 1(c).

For iterative modulo scheduling, candidate loop Π is initially set equal to MII , which is computed as follows. First, each iteration of the branch-free loop body shown in Figure 2(b) requires 4 units of ALU and 5 units of AAU. Thus, $ResMII$ is 3, which is $\max(\lceil \frac{4}{4} \rceil, \lceil \frac{5}{2} \rceil)$. Second, $RecMII$ for the half-rate GSM is 5 and the corresponding $RecMII$ recurrence circuits are depicted in Figure 2(c). Since MII is $\max(ResMII, RecMII)$, Π is initially set to 5 for modulo scheduling.

```

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

n. AAU instruction

m. ALU instruction

(b) An assembly code of figure (a)

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

Fig. 2. C Code fragment from half-rate GSM and corresponding loop body in SC140

For analysis, consider the two loop-carried data dependences in Figure 2(c). 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 `d10`. This type of composite loop carried dependences is often observed when DSP specific instructions are selected by the code generator. Due to these two $RecMII=5$ recurrence circuits, half-rate GSM candidate loop fails to be modulo scheduled with Π smaller than 5.

Note that modulo scheduling requires a candidate loop Π be selected before scheduling is attempted. A smaller Π corresponds to a shorter execution time. Since the MII is a lower bound on the smallest possible value of Π for which a modulo schedule exists, the candidate loop Π is initially set equal to the MII and increased until a modulo schedule is obtained. Therefore, a preprocessing strategy that lowers the MII by reducing $RecMII$ can be quite an effective preparation to achieve high loop initiation rate modulo schedules.

3 Problem Formulation

The compiler eases our preprocessing task by putting every candidate loop body such that intra-loop **false** dependences are removed whenever possible. In that setting, our preprocessing reduces ExRecMII by exploiting underutilized functional resources by capitalizing on the following two techniques.

- **Cloning1**: eliminate loop-carried **true** dependences of RecMII recurrence circuits by cloning the value of an induction register, and
- **Cloning2**: relax loop-carried **false** dependences of RecMII recurrence circuits by splitting and cloning the excessive lifetime data value used in a *destructive* instruction that requires use of the same register for source and destination, i.e. *mac* [14].

Since few functional resources are typically left for the preprocessing, the challenge is to find an optimal allocation of critical resources for cloning1 and cloning2, which reduces RecMII by the largest degree, subject to the constraints of ResMII increase.

Definition 4. Splittable points: *splittable points are candidate instructions to which cloning1 and cloning2 can potentially be applied for RecMII data dependence circuits. According to the corresponding data dependences, these points are classified into **true** and **false** types. In particular, cloning1 is engineered to remove the loop-carried **true** dependence created by a load/store instruction with postincrement (or postdecrement) addressing mode. Thus, the splittable point for cloning1 takes only the load/store instruction that forms the loop-carried **true** dependence. Cloning2 is engineered to split the excessive lifetime of the data value created by a sequence of destructive DSP instructions [14]. Therefore, for loop-carried **false** dependences, each destructive instruction of the RecMII data dependence circuit is a potentially splittable point.*

3.1 Benefit Estimation

Once splittable points are identified for a set C of RecMII recurrence circuits, these points need to be partially ordered to make the best use of underutilized resources for our preprocessing. First, the potential benefit of the preprocessing for the k^{th} RecMII circuit $\in C$, is estimated by the following equation:

$$B_k = \text{RecMII}_k - \text{RecMII}_k' \quad (1)$$

where RecMII_k' is the largest dependence length of the dismantled recurrence circuits when cloning1 and cloning2 are applied to the k^{th} RecMII circuit. B_k is used as a metric to estimate the local benefit of the preprocessing. Second, the overall (global) benefit of clonings for C is estimated by the following equation:

$$B_C = \text{RecMII} - \text{MAX}(\text{RecMII}_k') \quad \text{for all } \text{RecMII}_k \in C \quad (2)$$

However, Equations 1 and 2 are not sufficient to achieve the desired partial ordering since a splittable point can be shared by multiple RecMII recurrence circuits. For this reason, the benefit estimation can potentially make our preprocessing fail to find the desired partial ordering.

Note that since the preprocessing techniques cloning1 and cloning2 require additional registers and functional resources, the proposed preprocessing is required to check the availability of these resources while estimating benefit.

3.2 Register Constraint and Resource Constraint

The number of architected registers of a processor is denoted as R_T , where T represents a register file type, i.e., data or address, which depends on architecture characteristics. First, to avoid spill code in a candidate loop, our preprocessing applies cloning1 and cloning2 only when the register budget allows. This register budget is the first constraint, which requires that additional register need N for clonings must not exceed the difference between R_T and the number of loop variables kept in registers.

Fig. 3. A negative effect of cloning2

Improper application of clonings to a given set of RecMII circuits either makes no improvement on II by simply wasting resources or may even make II worse due to the excessive increase in ResMII. As an illustration, consider the example candidate loop in Figure 3(a), which contains a RecMII = 3 recurrence circuit (2-3-4) and ResMII = 1. When the data register d5 of the 3rd mac instruction is cloned with additional data register d8, MII is reduced by 1 in Figure 3(b).⁵ Nevertheless, in case clonings increase ResMII more than the decrease in RecMII, net effect can make MII worse.

To prevent such an indiscriminate application of clonings which may require more resources than the architecture can possibly provide, resource budget is additionally considered as the second constraint for our preprocessing. The resource budget is modeled using Rau's reservation table [2], which represents the resource occupation of each loop instruction in a partial schedule.

3.3 Problem Formalization: MAX-MIN

The solution to our resource allocation problem, which searches for an optimal sequence of splittable points, requires the ability to identify the best splittable point I_s from all possible permutations of splittable points. Obviously, a backtracking-based algorithm cannot be a viable approach since the runtime complexity of this combinatorial problem grows exponentially in terms of the number of splittable points. To respond to this intractability,

1. *Max-Min* problem is formulated that requires a solution to maximize the decrease in II while minimizing both register pressure and resource bound, and
2. *Branch and Bound* approach is employed to effectively search for an optimal split point I_s .

⁵ ExRecMII is reduced from 2 to 0.

In particular, the Max-Min problem for our preprocessing is to seek an I_s that maximizes *profit* under the register and resource constraints described in Section 3.2.

$$\text{profit}(I_s) = (B_C/N) \quad (3)$$

where B_C is computed by Equations 1 and 2, and N is the number of additional registers required for clonings. Note that, since the benefit estimation with Equations 1 and 2 is not sufficient to find an optimal partial ordering, the branch and bound approach may produce a suboptimal solution.

4 Preprocessing Strategy for Effective Modulo Scheduling

Since a candidate loop can have exponentially many recurrence dependence circuits, the proposed preprocessing strategy sets up the Max-Min problem described in Section 3.3, and exploits a *divide-and-conquer* principle to effectively search for a suboptimal splittable point I_s .

4.1 Divide Step: Detecting ExRecMII Recurrence Circuits and Finding Splittable Points

To identify all recurrence circuits which account for ExRecMII in a candidate loop, we use Tiernan's algorithm [12] with the C data structures shown in Figure 4:

```

/* 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; // Initiation interval of rc
    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 */
    int status; // DRYRUN|DONE|CLONE1|CLONE2
};

/* List of Ex-RecMII Recurrence Circuit */
static struct LIST *ECs;

```

Fig. 4. Elementary Circuit (EC) and EC list C data structures

1. When Tiernan's algorithm confirms a non-trivial⁶ recurrence circuit $rc=(inst_1 \rightarrow inst_2 \rightarrow \dots \rightarrow inst_n)$, each dependence edge in rc is retrieved from DDG to estimate II_{rc} .

⁶ A trivial recurrence circuit is a self recurrence circuit whose dependence arc is from an operation to itself.

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 4 are used; `struct EM_CT` for a recurrence circuit and `struct LIST` for EC list.
3. Prior to rc insertion, for each rc_i in EC list, the following two `EM_CT` fields of rc_i and rc are updated.
 - `i_ecs`: set of intersecting recurrence circuits, and
 - `p_ecs`: set of properly contained recurrence circuits.

The `i_ecs` and `p_ecs` fields are later exploited to find an optimal splittable point for cloning2. Since the initialization of these two fields for each recurrence circuit potentially requires the algorithm to determine set relationships with all other recurrence circuits, the upper bound for set operations is $O(n^2)$, where n is the number of recurrence circuits of a candidate loop body. Therefore, in order to perform a set operation in constant time, we represent the constituent instructions of a recurrence circuit as a bit vector, which are encoded into the `circuit` field of `EM_CT`.

When the circuit confirmation completes, the II_{rc} of the head circuit in the EC list is the `RecMII` recurrence circuit. If $RecMII > ResMII$, then all the circuits that share the same value of II_{rc} in the EC list are `ExRecMII` circuits. For each `ExRecMII` recurrence circuit, splittable points are determined as a divide step. In particular, starting backward from the loop instruction indexed by `tail` to the head, splittable points are detected while performing a local reaching definition analysis to estimate the register need N for clonings.

4.2 Conquer Step

PHASE 1: Simplification

Theorem 1. *Unless all loop-carried **true** dependences of `ExRecMII` circuits are eliminated by cloning1, the `RecMII` of a candidate loop cannot be reduced by cloning2.*

Proof. For a candidate loop that contains `ExRecMII` circuits, $C = \{c_1, c_2, \dots, c_n\}$, assume that c_k and $c_l \in C$ are dependence recurrence circuits which share a splittable point I_s . Without loss of generality, let the loop-carried dependence of c_k be **false** for cloning2 and let the loop-carried dependence of c_l be **true** for cloning1. When cloning2 is first applied to I_s along c_k dependence path, the technique introduces additional data transfer instruction that clones the value of destination operand u to other register v . Since c_l also shares the splittable point I_s , subsequent references to u in c_l must be renamed to v to preserve the original semantic. Therefore, cloning2 cannot reduce `RecMII` since `ExRecMII` of c_l remains unchanged.

Theorem 1 implies the following corollaries which simplifies the desired search into three steps.

Corollary 1. *Cloning1 must be applied prior to applying cloning2 to reduce `ExRecMII`.*

Corollary 2. *The optimality in allocating functional resources is not affected by the order of splittable points selection for cloning1.*

1. Partition the set of ExRecMII recurrence circuits into *c-worklist* for cloning1 and *d-worklist* for cloning2, where *c-worklist* is a set of ExRecMII circuits whose loop-carried dependences are **true** and *d-worklist* is a set of ExRecMII recurrence circuits whose loop-carried dependences are **false**.
2. According to Corollaries 1 and 2, randomly select a sequence of splittable points from *c-worklist* as long as register and resource constraints can be met.
3. Find an optimal sequence of splittable points from *d-worklist* using a branch-and-bound search algorithm.

PHASE 2: Search for an optimal solution from a d-worklist

An optimal sequence of splittable points for a d-worklist can be found only when the following side effects are accurately estimated.

1. The number of circuits in the d-worklist whose RecMII can be simultaneously improved when a particular splittable point is cloned.
2. The prediction whether the overall loop schedule gets worse when the particular point is cloned.

As an instance of the first side effect, consider Figure 2(c) that shows two $\text{RecMII}=5$ recurrence circuits. When the 7th `mac` instruction is cloned, the loop-carried **false** dependences from these two circuits can be simultaneously relaxed with no increase in ResMII .⁷

Fig. 5. An example of indiscriminate cloning2

points $\{I_{S_1}, I_{S_2}, \dots, I_{S_n}\}$ from d-worklist and p is the profit function that estimates the benefit of a given splittable point in terms of overall number of delisted circuits from d-worklist when the point is cloned. For this profit estimation, the `i_ecs` and `p_ecs` fields in `EM_CT` which are described in Section 4.1 are exploited.⁸ Second, the

As an illustration of the second side effect, consider the example candidate loop in Figure 5(a), which contains two $\text{RecMII}=3$ recurrence circuits. When the 3rd `mac` instruction is accidentally cloned for rc_A , the dependence path of rc_B is increased by 1 as shown in Figure 5(b), and as a result, the overall loop schedule can get worse.

To consider these two side effects in finding an optimal sequence of splittable points from d-worklist, our pre-processing first formulates an instance of Max-Min problem, that has the form (S, p) , where S is the set of splittable

⁷ This happens because the resource bound for half-rate GSM is determined by the contention on AAU in SC140 DSP.

⁸ When the second side effect is forecasted for a given splittable point, the profit function returns a negative value.

preprocessing exploits the branch and bound search approach [4] and effectively searches for an optimal splittable point $I_{S_x} \in S$ whose $p(I_{S_x}) \geq p(I_{S_y})$ for all $y=1,2,\dots,n$ and $y \neq x$.

1. **Initialization:** the maximal *Priority Queue*(PQ) [5] as an abstract data type is initialized with a set of splittable points defined in Definition 4, where the priority of each splittable point $I_{S_i} \in S$ is set to $p(I_{S_i})$. PQ has four basic operations: *isEmpty()*, *delMax()*, *validate(I_S)*, and *insert(I_S)*. Second, the branching factor bf , which is the lower bound of the profit function p , will be initialized to 0.
2. **Analyze and Bound:** the *delMax()* returns the current maximum splittable point I_{S_i} together with the EM_CT rc_i . First, the resource and register constraints for I_{S_i} will be checked. Second, if these constraints can be satisfied, the profit $p(I_{S_i})$ will be compared to the current lower bound to determine whether it is beneficial to explore the search space rooted at I_{S_i} . Third, if I_{S_i} is beneficial, the lower bound bf will be reset to $p(I_{S_i})$.
3. **Branch:** If I_{S_i} is not beneficial, then the profit $p(I_{S_i})$ will be adjusted and I_{S_i} will be enqueued with *insert(I_S)*. Otherwise, the current context before the branching will be saved so that remaining search space rooted at other splittable points can be explored.
4. **Iteration:** steps 2 and 3 will be repeated until *isEmpty()* returns true.

PHASE 3: Code Generation with Cloning1 and Cloning2

Fig. 6. RecMII=6 recurrence circuit of FFT candidate loop

Phase 3 - Cloning1: The recurrence circuit in Figure 6(a) highlights RecMII=6 of the FFT loop illustrated in Figure 1. Since this circuit is formed with cyclic **true** dependences, the ExRecMII deems irreducible. However, careful analysis on this circuit leads us to observe the following:

1. The loop-carried **true** dependence is an artifact of scheduling insensitive loop optimization, such as induction variable elimination and addressing mode optimization.
2. The loop-carried **true** dependence has no memory (store-load) dependence.

When these two conditions are met, the loop-carried **true** dependence edge can be removed by cloning1, which replicates an original induction register. As an illustration, cloning1 removes the loop-carried **true** dependence in Figure 6(a) as follows.

1. Allocate one additional induction register to clone the value of the original induction register `r1`, and initialize it at the loop preheader. First, live-variable analysis followed by local reaching definition indicates the availability of additional register `r10` to clone the value of `r1`. Second, a transfer instruction `tfr a r1, r10`, which initializes the cloned register `r10`, is placed at the loop preheader as shown in Figure 6(b).
2. Prepare one additional operation that clones the induction register `r1` used in the 1st instruction, which was selected according to Definition 4. Place this additional operation prior to the update of `r1` value. In particular, to minimize resource pressure on AAU units, postincrement addressing mode is exploited at the 1st instruction. Note that, since the memory stride between the 1st and 9th instructions differs by two bytes, indexed postincrement addressing mode is selected as shown in Figure 6(b).
3. Finally, eliminate the original loop-carried **true** dependence by making the cloned operand being referenced. Since the 1st instruction is already amended to reference clone operand `r10`, no additional change is required.

As a result of this transformation, the original loop-carried **true** dependence is removed. By applying `cloning1` to other `RecMII=6` recurrence circuit that exists in Figure 1(b), `MII` is reduced from 6 to 4. Without a single modification to an existing modulo scheduler, a higher loop initiation rate of 4 is effectively achieved and the modified schedule results in a 17% performance improvement.

Fig. 7. GSM code applied the solution

Phase3 - Cloning2: Figure 7(a) shows the candidate loop body, which contains two `RecMII=5` recurrence circuits highlighted in Figure 2(c). To relax the loop-carried **false** dependences from these `RecMII` recurrence circuits without strip-mining the original loop and unrolling the loop kernel, `cloning2` technique is engineered, which splits excessive lifetimes of registers by moving data values around. In particular, `cloning2` targets for complex [14] and destructive instructions that requires use of the

same register for source and destination. As an illustration, cloning2 relaxes the `RecMII` loop-carried **false** dependences that exist in Figure 7(a) as follows.

1. BB search in phase 2 selects the 7th `mac` instruction as the splittable point.
2. Allocate one additional register to clone the excessive lifetime of `d10` of the `mac`. For this task, the same live-variable analysis, followed by local reaching definition analysis, tells us the availability of `d11`.
3. Place one additional transfer instruction, the 7_ath instruction in Figure 7(b), to split the lifetime of `d10` with the cloned `d11`.

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 7(a) are both relaxed; Figure 7(c) depicts relaxed recurrence circuits. As a result of cloning2, the original `RecMII`=5 for half rate GSM candidate loop is effectively lowered to 3 and the modified schedule results in a 4.7% performance improvement.

4.3 Unified Framework: Divide-and-Conquer

For a given candidate loop, the actual cloning1 and cloning2 techniques will be performed only when the analysis from divide-and-conquer steps forecasts the entire recurrence circuits in c-worklist and d-worklist can be simultaneously relaxed. Since this process iterates until there is no further change in `ExRecMII`, most of the search space for an optimal sequence of splittable points, which means an optimal allocation of underutilized functional units in a multi-issue DSP, is typically exhausted.

5 Experimental Results

This section describes results of a set of experiments to illustrate effectiveness of the unified preprocessing strategy described in Section 4.3, which is implemented for the SC140 optimizing C compiler. The experimental input is a set of candidate loops obtained from DSPStone [18], MediaBench [3], half-rate GSM, enhanced full rate GSM, and other industry signal application kernels. Table 1 lists benchmarks used for our experiments.

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

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

With these two sets of executables, we measured (1) cycle counts with StarCore *cycle count accurate* simulator `simsc100`, and (2) code size with 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 formula $((\text{ORIG} - \text{PRE})/\text{ORIG}) * 100$.

Table 1. Benchmarks used in experiments

Acronym	FD	LMS	Conv	ComFFT	FFT	Matrix	FIR
Program	FIR2DIM	LMS	Convolution	complex FFT	FFT	Matrix	FIR
Description	2 dimensional Finite Response Filter	Least mean squared adaptive filters	Convolution	128 point complex FFT	Integer stage scaling FFT	Generic matrix multiply	Finite impulse response filter
Acronym	Mat1x3	Ncomp	BiNsec	IIR	Latsyn	Pcrypto	GSM (sc, af, ut, dec, ad, sy)
Program	Matrix1x3	N_complex_updates	Biquad_N_section	IIR	Lattice synthesis	Panama cryptographic module	GSM
Description	1x3 matrix multiply	Complex multiply	One IIR biquad	IIR filter	Typical DSP multiply two vector operation	Panama stream/hash module	V_search, affatRecursion, utcount, decode, add, syn, fi modules from Global System for Mobile Telecommunication

Figure 8(a) reports performance improvements achieved by applying the unified algorithm in Section 4.3, which is based on cloning1 and cloning2 techniques respectively. The overall performance improvement from our preprocessing ranges from 0.3% to 29.5%, and the average performance improvement is 12.9%. Considering there is no modification made to existing iterative modulo scheduler and the performance comparison is made to highly optimized SC140 DSP code, 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. iterative application of cloning1 followed by cloning2 for an existing modulo scheduler can deliver more performance gain by effectively reducing the `ExRecMII` of a candidate loop, and
2. the preprocessing strategy described in Section 4.3 can detect and exploit such opportunities for an effective modulo scheduling.

Note that none of benchmarks in Figure 8(a) reports performance degradation. This is not a coincidence, since our algorithm is designed to apply cloning1 and cloning2 only when the additional operations for these techniques can be placed in non-critical recurrence circuits.

Figure 8(b) reports code size increase due to the unified algorithm. Since cloning1 and cloning2 reduces `ExRecMII`, existing modulo scheduler discovers instruction level parallelism across more loop iteration boundary and as a result, achieves a better modulo schedule. Since the size of the prologue and the epilogue grow proportionally as more loop iterations of a candidate loop get overlapped for a final schedule, code size increase is unavoidable. However, we also observed that existing modulo scheduler can find a better loop schedule for a number of loop iteration boundaries when our preprocessing is applied. This is the reason why our preprocessing to IIR, GSMdec, GSMad and GSMsy benchmarks, reports significant performance improvements with negligible increase in code size.

For benchmarks listed in Figure 8(b), overall code size increase from the proposed preprocessing ranges from 0% to 63.1%, and the average increase is 13.99%. However, note that benchmarks in Table 1 are critical loop kernels which typically account for

Fig. 8. Percent wise performance improvement (number of cycles reduction) and code size increment compared to the original

5% - 10% of entire application code size. By carefully applying the preprocessing to mission critical loops with profiling, overall code size increase can be hold to a moderate amount.

6 Related Work

To effectively lower this ExRecMII, Lam pioneered a compiler technique, referred to as Modulo Variable Expansion (MVE), that removes loop-carried **anti** and **output** dependencies in recurrence circuits [15]. Since MVE achieves the desired removal with loop unrolling followed by register renaming, a high loop unrolling factor might incur an 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 [11].

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 [16]. Since the Huff technique still requires a large number of the architected rotating registers to support MVE without code expansion, Tyson and et al. ameliorated Huff technique with *register queues* and *rq-connect* instruction [9]. In their technique, register queues share a common name-space with physical register files. As a consequence, the architected rotating register space is no longer a limiting factor.

7 Conclusion

This paper describes compiler optimizations that preprocess loop kernels of signal processing applications to relax their intrinsic data dependencies and thereby, complementing iterative modulo scheduler. The presented strategy is implemented for the

StarCore SC140 optimizing C compiler backend. As a result of the implementation, a 12.9% average runtime improvement is reported for benchmarks in Table 1; This runtime improvement is made at the expense of a 13.99% average code size increase. Considering that no modification is made to existing modulo scheduler and that the performance comparison is made to highly optimized SC140 DSP code, we believe that this gain is impressive.

Acknowledgement

We thank Bharadwaj Yadavalli, John Griffin, Ramesh Peri, Robert Cohn, and Sanjay Jinturkar for many valuable discussions that improved the quality of this paper. We thank the anonymous reviewers for their constructive comments and suggestions.

References

1. A. Stoutchinin. An Integer Linear Programming Model of Software Pipelining for the MIPS R8000 Processor. In *Proceedings of the 4th International Conference on Parallel Computing Technologies*, 1997.
2. B. Rau. Iterative modulo scheduling. In *HP Laboratories Technical Report, HPL94115*, Nov 1995.
3. 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.
4. D. Smith. Random trees and the analysis of branch and bound procedures. In *Journal of the Association for Computing Machinery*, Jan 1984.
5. R. Sedgewick. Algorithms in C. Third Edition, Addison-Wesley Pearson Education, 2003.
6. D. Batten, S. Jinturkar, J. Glossner., M. Schulte, and P. D'Arcy. A new approach to DSP intrinsic functions. In *Proceedings of the Hawai International Conference on Systems and Science*, Jan 2000.
7. 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.
8. 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.
9. 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.
10. G. Uh, Y. Wang, D. Whalley, and et al. Compiler Transformations for Effectively Exploiting Zero Overhead Loop Buffer. In *Software-Practice & Experience*, Vol 35, pages 393-412, 2005.
11. H. Allan, B. Jones, M. Lee, J. Allan. Software Pipelining. In *ACM Computing Surveys*, Vol.27, No.3, Sep 1995.
12. J. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. In *Communications of the ACM*, pages 12-35, Dec 1970.
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. R. Leupers and P. Marwedel: Instruction selection for embedded DSPs with complex instructions. In *Proceedings of European Design Automation Conference*, Sep 1996.
15. 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.
16. R. Huff. Lifetime-Sensitive Modulo Scheduling. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June, 1993.
17. StarCore, Inc. SC140 DSP Core Reference Manual. Atlanta, GA, 2001.
18. 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.