Instruction Re-selection for Iterative Modulo Scheduling on High Performance Multi-issue DSPs*

Doosan Cho[1], Ayyagari Ravi[3], Gang-Ryung Uh[3], and Yunheung Paek[1,2]

[1] School of Electrical Engineering and Computer Sciences,
Seoul National University, Seoul 151-744, Korea
ypaek@snu.ac.kr, dscho@compiler.snu.ac.kr
[2] Center for SoC Design Technology in Seoul National University
[3] Department of Computer Science,
Boise State University, 1910 University Drive, ID 83725, USA
{uh, rkayyagari}@onyx.boisestate.edu

Abstract. An iterative modulo scheduling is very important for compilers targeting high performance multi-issue digital signal processors. This is because these processors are often severely limited by idle state functional units and thus the reduced idle units can have a positively significant impact on their performance. However, complex instructions, which are used in most recent DSPs such as `mac`, usually increase data dependence complexity, and such complex dependencies that exist in signal processing applications often restrict modulo scheduling freedom and therefore, become a limiting factor of the iterative modulo scheduler.

In this work, we propose a technique that efficiently reselects instructions of an application loop code considering dependence complexity, which directly resolve the dependence constraint. That is specifically featured for accelerating software pipelining performance by minimizing length of intrinsic cyclic dependencies. To take advantage of this feature, few existing compilers support a loop unrolling based dependence relaxing technique, but only use them for some limited cases. This is mainly because the loop unrolling typically occurs an overhead of huge code size increment, and the iterative modulo scheduling with relaxed dependence techniques for general cases is an NP-hard problem that necessitates complex assignments of registers and functional units. Our technique uses a heuristic to efficiently handle this problem in pre-stage of iterative modulo scheduling without loop unrolling.

Keywords: code generation and optimization, application specific embedded software design, software pipelining, dependence analysis, high performance DSPs.

1 Introduction

Software pipelining uses the global cyclic scheduling concept to restructure loops such that each iteration in a pipelined loop is made from instructions scheduled from different

* This work was partially funded by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Inofrmation Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031), KRF contract D00191, 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 2006.

X. Zhou et al. (Eds.): EUC Workshops 2006, LNCS 4097, pp. 741–754, 2006.

iterations of the original loop. Thus, the height of the rescheduled loop can be shortened and therefore, the resource utilization for multiple functional units can be drastically improved. Production C compilers for multi-issue DSPs typically use variants of iterative modulo scheduling to implement software pipelining [5]. However, intrinsic cyclic data dependences that exist in signal processing applications often restrict modulo scheduling freedom and therefore, leave replicated functional units in DSPs underutilized.

To address this resource utilization problem, the objective of this document is twofold; to analyze the nature of the data dependences existing in various signal processing applications, and to engineer effective compiler preprocessing strategy to help an existing modulo scheduler achieve a high quality loop schedule by relaxing data dependence constraints. In particular, this document presents an effective preprocessing technique. Since this technique directly amend intrinsic cyclic data dependences, neither code duplication nor additional hardware support is required and therefore, it is easier for C compilers to implement. To measure the feasibility and effectiveness of our preprocessing technique, the StarCore SC1400 DSP processor is used as the representative for multi-issue based DSPs.

2 Terminology

Definition 1. *A* **candidate loop** *for an iterative modulo scheduler is the loop with branch-free body.*[1]

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

Definition 3. *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 4. Minimum recurrence bound *(*`RecMII`*) is the maximum of all* $\texttt{II}_{rc}$ *which can meet the deadlines imposed from all the recurrence circuits existing in a candidate loop.*

Definition 5. Minimum resource bound *(*`ResMII`*) is the smallest* `II` *which can meet the total resource requirements to complete one loop iteration of a candidate loop.*

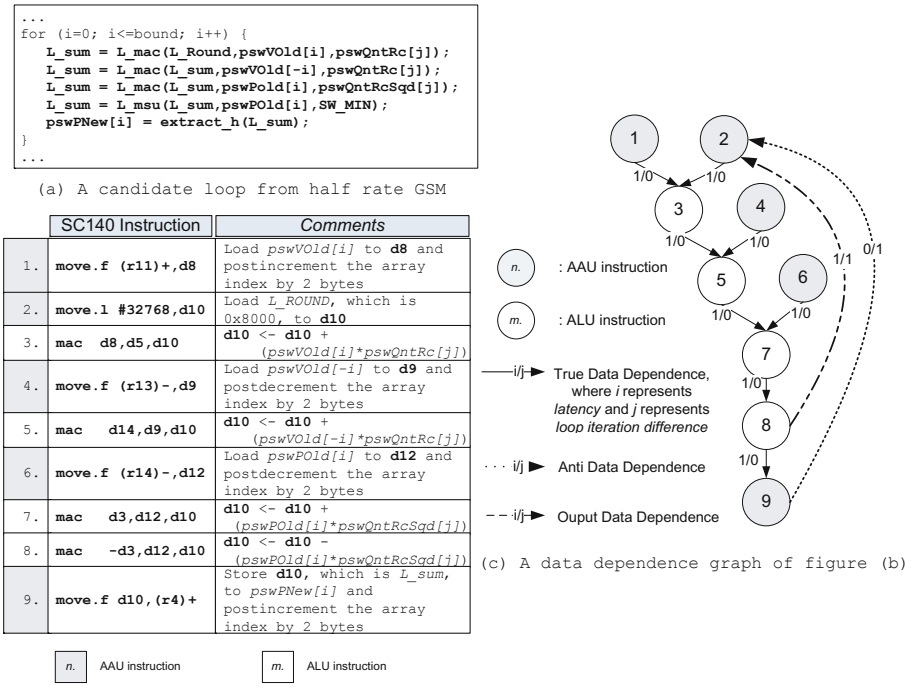
Definition 6. Minimum Initiation Interval *(*`MII`*) is the maximum of* `RecMII` *and* `ResMII`.

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

[1] Production C compiler performs if-conversion to allow more loops to be modulo scheduled.

3 Motivation

According to our benchmark with media applications for high performance DSP, various applications manifest that critical path consisting by dependence relation is dominant limiting factor that either fails candidate loops to be modulo scheduled or modulo schedules with large critical path length. The C code fragment shown in Figure 1.(a) that implements Global System for Mobile (GSM) algorithm. For the innermost loop body, SC1400 production C compiler produces highly optimized assembly code as shown Figure 1.(b), which is yet to be modulo scheduled.



```
...
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)

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

Fig. 1. An example of motivation

For analysis, let's consider the anti and output dependencies in Figure 1.(c). These dependencies are produced by that the result of *move.l* instruction is referred to the 8st and 9st instructions in the loop body. Due to these recurrence circuits, the $RecMII$ is calculated to be 5 greater than $ResMII$[2] as [5/2]=3 by Rau's algorithm [5], and thus the GSM candidate loop can be archived modulo scheduled loop in MII of 5. If it is able to reduce $RecMII$ to $ResMII$, a modulo scheduler can build a better scheduled loop body. To this end, there will be explained what is a problem and how to achieve $RecMII$ to $ResMII$ in a candidate loop in section 4 and section 5 respectively.

[2] To support high computing needs, StarCore 1400 has 4 ALU units and 2 AGU units.

4 Problem Formulation

To determine the amount of available instruction level parallelism in a candidate loop, a data dependence analysis is performed. To that end our algorithms are performed on data dependence graph, which is a directed graph with weighted vertices. We assume that a data dependence graph (DDG) supplemented with delay information is used to represent a loop. In a DDG, nodes represent the instructions in a program. An edge from node i to node j indicates that there is a data dependence between them. Loop carried dependences are indicated by edges with positive numbers beside them indicating their dependence defined below. If node i produces a result at the current iteration and the result will be used h iterations later by node j, then we say that the edge (i,j) has a dependence distance λ and we use λ_{ij} to indicate it. So for a data dependence not crossing iterations, the λ_{ij} is 0. δ_i (called delay) is used to indicate the number of clock cycles node i needs to finish its execution.

Definition 7. Data Dependence Graph. *A data dependence graph is a tuple (N, E, λ, δ) where N is the set of nodes, E is the set of edges, λ = { λ_{ij},$\forall$(i,j)$\in$E } is the dependence distance vector on edge set E, and δ = { δ_i,$\forall i \in N$ } is the delay function on node set N.*

For a pair of nodes on path P in a candidate loop, the minimum schedule distance can be computed by: $D_P = \delta_i - MII \times \lambda_{ii}$, where the i are nodes on a DDG. It is a dependence constraint of iterative modulo scheduling [5].

Definition 8. Critical Recurrence Path. *Let v be a node in DDG, for all cyclic paths P passing through node v, if a path passes v and the schedule distance $D_{P_a}(v,v) \geq D_{P_b}(v,v)$ where are $\{a, b \in P\}$, the path a is called* **the longest recurrence path** *or* **critical recurrence path**. *Length of a critical recurrence path is then defined as $\sum_{i,j \in I} \delta_i/\lambda_{ij}$, where I is an instruction set on P_a , in case of true, output dependencies. An exception case that an anti dependence needs to minus latency of the last instruction on it, since the last instruction of a recurrence path does not make any latency to the first instruction of a path.*

It is the shortest time it takes to complete a task represented by a critical recurrence path in a candidate loop. However, instructions in a loop can be carried out in parallel with finite resource of multi-issue processors, the time to complete the overall task may be reduced by an additional hardware resource. Thus, maximum performance in the candidate loop body will therefore be obtained by making the critical path as short as possible using parallel execution of independent instructions cross iterations. We achieve it by instruction re-selection to reduce length of a critical recurrence path.

Definition 9. Re-selectable Instruction Set. *A reselectable instruction set means that an instruction has semantically same instructions provided by instruction set architecture (ISA). For instance, an* imac da,db,dc *instruction is to perform integer multiplication and accumulation by one cycle which designed for frequent signal processing on various DSPs. The* imac *can be transformed to integer multiply and add instructions that are* impy da,db,de *and* add de,dc,df. *Similarly, a complex instruction I can*

be transformed to a semantically equivalent instruction set I'. We will refer the I instruction to a re-selectable instruction. *In this paper, the following set of instructions in StarCore 1400 ISA are considered for reselectable instructions: mac, imac, increment, arithmetic shift right(left), sign extension, etc.*

Definition 10. Critical Operand. *The critical operand means that an operand composes a critical path by naming recurrence. If there is a critical operand used in a re-selectable instruction, we can reselect the instruction into semantically same one with one or two unused registers. Since this unused register reduces live range of a critical operand, consequently it reduces length of a critical path if and only if it is anti or output dependence.*

To find such critical operands, we use a two dimensional array `Reg_live`, which includes all variables's live range of each basic block. By using this, we can easily find that the critical operand in Figure 1 is `d10` which composes the critical recurrence path. However, it can not be applied register renaming for resolving a dependence constraint, since the operand has roles of definition and use simultaneously. Thus, the dependence circuit can not eliminate by traditional optimizing techniques such as `register renaming` or `SSA transformation` without any hardware support. In that case, the proposed instruction re-selection is a unique way to resolve such a dependence constraint.

Definition 11. Benefit Calculation. *Original critical path length (CPL) is $RecMII$ in a candidate loop according to Definition 8. Assuming we were successful in identifying re-selectable instructions and critical operands for a given candidate loop. We can then reselect instruction I with critical operand into one of the semantically same instructions I' with an unused register. The unused register changes the critical operand's live range in short way which is to reduce $RecMII$ consisting anti and output dependence. Definition of benefit calculation for individual critical path is as follows:*

$CPL > CPL'$, where CPL' is critical path length applied instruction reselection

$B_k = CPL_k - CPL'_k$, where k is critical path $k \in P$

It can be redefined for critical path set P:

$$B_P = RecMII - MAX(CPL')$$

To calculate the CPL', we implemented a dry run stage which performs to replace I to I', and then reconstruct DDG for computation the CPL'. The dry run stage does not actually change a candidate loop code, it is just for measuring the CPL'.

Definition 12. Minimizing register need with Maximizing benefit problem. *In the Min-Max problem, along with Definition 9,10 and 11, we can reselect an instruction I into I' with reducing $RecMII$. The problem is to identify a solution I in a powerset of reselectable instructions, which is a basic candidate solution, S and to satisfy the register file size constraint in Definition 13 and limited resource bound constraint in Definition 14 with benefit per register need in maximized. If a semantically same instruction of $I \in S$ is assigned to a re-selectable instruction with critical operand on DDG of a candidate loop, it should maximize*

cost(I) = ($B_P/Q(N_P)$),
where the Q(N) is register need described in Definition 13,

found an S and critical path set P. This optimizing procedure may increase register need which occurs an overhead of code size increment, and thus we have trying to reduce it with maximum cost.

Since search space S to find the optimum solution is consisting of all combination of reselectable instructions, the search space is exponentially increased. We try to efficiently resolve this problem by three phase algorithm which is specifically described in next section.

Min-Max problem have to satisfy below two constraints simultaneously because a proposed solution does not make any negative side effects.

Definition 13. Register File Size Constraint. *We will refer the amount of register to integer value as R_T. The R_T has a type of register file T=$\{$integer, float, address$\}$ which kinds of type depend on architecture characteristics. This constraint is that a register need $Q(N)$ plus the number of physical registers R in a candidate loop code should be less than R_T. Thus, a proposed technique guarantees it does not make any spill code in a candidate loop.*

The amount of register need of a candidate loop by instruction reselection is represented by the function $Q(N)$ which is calculated based on the number of new operands for equivalent instructions selected by branch and bound search strategy as described in section 5.2.

Definition 14. Limited Resource Bound Constraint. *The minimum resource bound is classified to $ResMII_{alu}$ and $ResMII_{agu}$ which are a minimum bound of arithmetic logic unit(alu) and address generation unit(agu). Since a proposed technique does not make any candidate loop worse scheduled case by increasing* `ResMII` *in a candidate loop, MAX($ResMII'_{alu}$, $ResMII'_{agu}$) does not exceed* `RecMII'`.

5 Solution of the Min-Max Problem

The critical path of Figure 1.(c) that decides completion time of the loop, even though it has enough hardware resource for accelerating performance of the loop. This section presents to resolve such a limitation using a novel framework based on a data dependence graph.

A compiler eases a proposed preprocessing task by putting the candidate loop body such that critical path are reduced whenever possible. For a given candidate loop, the main task of our preprocessing solution is as follows.

Figure 2 shows overview of the proposed algorithm. It reduces a critical path of a recurrence circuit which being formed by a complex (or compact) instruction by means of the instruction reselection technique, which replaces it with a sequence instructions that archives semantically equivalent instructions.

- Register File Size Constraint: spill code,
- Limited Resource Bound Constraint: detrimental to performance and code size,
- Bound Condition Constraint: compile time and code size increment.

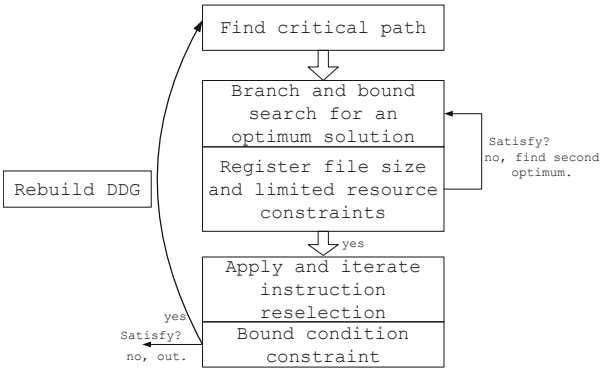


Fig. 2. Algorithm overview

If each phase does not satisfy three constraints of Min-Max problem, the procedure must be stop its execution process or it abandon to find an optimum solution. Because those safe-guards are designed to guarantee that the proposed technique never makes a loop schedule worse in above various point of view.

5.1 Phase 1: Find Critical Recurrence Paths in a Candidate Loop

To reduce critical path length, it is essential to identify all recurrence circuits which account for the critical path. For this task, we used the Tiernan's[1] algorithm which uses an exhaustive search to find all of the elementary circuits of a graph. The algorithm is easily modified to find all of the elementary circuits with length. When the Tiernan's algorithm confirms the ith recurrence circuit P_i, which is (inst$_{i_1}$→inst$_{i_2}$→. . .→inst$_{i_n}$) into array P shown in Figure 3, each edge in P_i is retrieved from the graph G to estimate *initiation interval (II)*. If the $II > ResMII$, the P_i is added to the EC list, which is sorted in descending order by *II* as a key. For this task, Figure 3 `C` data structures used to implement the Tiernen's algorithm.

The II of the first element in the ECs list is $RecMII$ and all the following elements which share the same II are the $RecMII$ circuits as a problem set of proposed instruction reselection technique. The output of the first phase can classify ECs by below two types.

- Overlapped circuits by shared critical operands: a recurrence circuit path proper(or nonproper) overlapped the other recurrence circuit path by a shared critical operand on a candidate loop. For instance in Figure 1, there is proper overlapped recurrence circuits by a shared operand `d10` as {2-3-5-7-8} and {2-3-5-7-8-9}.
- Independent circuits: a recurrence circuit does not overlapped between other recurrence circuits.

In general, there are multiple overlapped recurrence circuits $\{c_1,c_2,\ldots,c_n\}$ in a candidate loop body. It can be found a shared operand set and re-selectable instruction set by means of live range analysis and reselectable instruction map table. If a solution I obtaining from reselectable instructions with a shared operand can concurrently resolve c_1 and c_2, consequently the register need $Q(N)$ is less than a case of individually resolved the c_1 and c_2. To this end, the main role of `bvect circuit` and `bvect op`

```
struct EM_CT {
    unsigned char head; // inst number: head of the circuit
    unsigned char tail; // inst number: tail of the circuit
    unsigned char II;   // initiation interval
    unsigned char *insn;// reselectable instruction building
                        // array used with instruction number
    unsigned char *P;   // recurrence circuit path building
                        // array used with instruction number
    bvect circuit;      // circuit representation in bit vector
    bvect op;           // critical operand in bit vector
};
/* List of Elementary circuit (Recurrence Circuit)*/
static struct List *Ecs; // linked list of Recurrence circuits,
                         // which is sorted in descending order
                         // with II as the key.
```

Fig. 3. Data structure EC

data structure entry of the EM_CT in Figure 3 are to effectively determine (1) whether the found $RecMII$ recurrence circuits overlap by a shared critical operand and (2) how many shared circuits can be simultaneously resolved by our preprocessing task with that additional registers is minimized. To perform set related operations in a constant time, the bvect structure of the EM_CT data structure is exploited.

Figure 1.(c) shows DDG from the GSM code. The Tiernan's algorithm confirms the proper shared $critical\ path = \{2, 3, 5, 7, 8\}$and {2,3,5,7,8,9} by an operation of intersection between bvect circuits, and the length is 5 as anti, output dependence respectively. At this time, this procedure finds a shared critical operand d10 and reselectable instruction as {3,5,7,8} on the critical path instructions. The critical operand is only $d10$ which composes the circuit with live range from instruction 2 to instruction 8.

When the circuit confirmation procedure described above completes, our instruction reselection technique consider that multiple RecMII recurence circuits $\{c_1, c_2, \ldots, c_n\}$ is required to perform the following analysis for code correctness before solution search phase.

1. When critical operands and reselectable instructions are obtained, check the availability of registers as additional operands and determine register need for the additional operands by the function Q(N) in Definition 13.
2. Determine whether an operand in between the $c \rightarrow$ tail and $c \rightarrow$ head instructions are used for memory operations; if true, then check for the memory dependence between head and tail instructions.
3. If there exists no memory dependence and there exists one more available registers, the next step is to search for maximum cost solution I by branch and bound strategy described in next phase.

5.2 Phase 2: Branch and Bound Solution Search Strategy

An instance Π of Min-Max problem has the form (S,f) where S consists of a set of candidate solutions, and f is a cost function as described in Definition 12. We use S' as subset of the S, which satisfy $S' \subseteq S$. (S_1,f) is a subinstance of (S_0,f) if $S_1 \subseteq S_0$. When no confusion can arise we will identify an instance by S', its set of feasible candidate solutions. An optimal solution to an instance S' is an object $x \in S'$ which has maximum

```
unsigned char *BB_Search(unsigned char *solution)
{

   /*i,k: integer
     solution: instruction number pointer of current feasible solution
     cand_solution: initialized a set of basic candidate solution
     PQ: priority queue
     cost: computes a cost
     Bound: check optimality of candidate solution*/

   Init_Bound();
   INSERT(cand_solution,PQ,cost(cand_solution));
   while(NONEMPTY(PQ)){            /*while there are feasible solutions ...*/
      cand_solution=SELECT(PQ);    /*find solution with most cost value*/
          if(Bound(cand_solution)){/*and explore it ...*/
             if(cost(cand_solution)>cost(solution)) /* if it find higher cost*/
                  solution = cand_solution;         /* then save it*/
             else{             /*otherwise apply the branching rule ...*/
                  Apply branching rule to generate power set of each basic
                  candidate solution as feasible solutions: N1, N2, ... ,Nk;
                              /*and store the power set of feasible solutions*/
                for(i=0;i<k;i++)
                  INSERT(Ni,PQ,cost(Ni));
             }
          }
   }
   return(solution);
}
```

Fig. 4. Branch and bound algorithm for finding maximum cost I as a solution

cost; that is, for all $y \in S'$, f(x) $>$ f(y). The goal of this branch and bound procedure is to find an optimal solution for any given instance of Π. The basic branch and bound procedure works as follows. An instance of a problem Π is analyzed, and if the maximum cost's solution is not easily extracted, then this procedure generates powerset of a set of basic candidate solution and cost is computed on the elements of the powerset. Those elements whose cost does not reach a lower bound of some known (perhaps non-optimal) solution's cost can be discarded since they can not contribute to generate an optimal solution. The remaining elements of powerset are repeatedly analyzed, generated, and bounded until an candidate solution whose cost does not exceed the most cost object on any element of powerset, hence that element is an optimal solution. This branch and bound procedure for finding a maximum cost I is used as follows.

Step1 A basic candidate solution set I_b is found by marking re-selectable instruction into `insn` entry in EM_CT which have equivalent instruction on critical paths in a candidate loop.

Step2 The branch and bound procedure is invoked for finding a maximum cost(I) with solution space S $\rightarrow 2^{I_b}$.

Step3 If a solution I does not satisfy constraints in Definition 13 and 14, this procedure should be stop, and remove it from the `insn` entry of `EM_CT`. This procedure goes to step2 until the candidate solution set will be empty, and search again in a solution space without the invalid I.

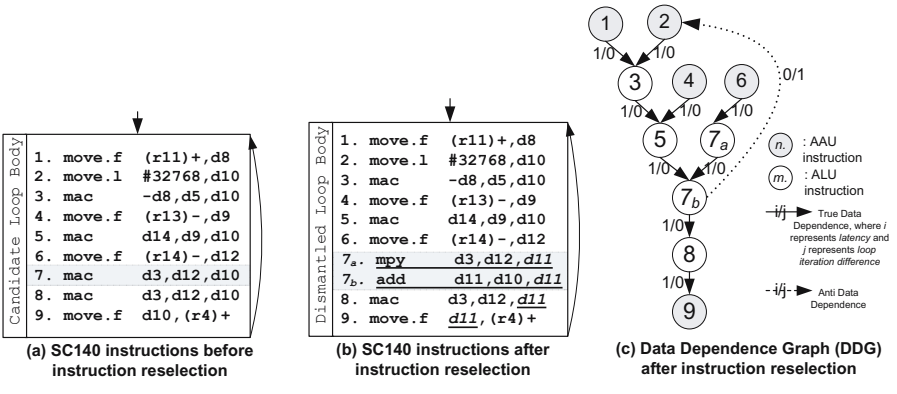
In Figure 4, the principal data structure is a priority queue which stores solutions with an associated priority given by the function $cost$. The queue is accessible only by the following three operators: NONEMPTY, which returns true if and only if the priority queue is nonempty; SELECT, which removes and returns the solution in

the priority queue with highest cost value; and INSERT, which inserts an solution into the priority queue with its associated cost value. The variable solution serves to record instruction number of the most-cost feasible solution currently known during the search process.

For instance of GSM code in Figure 1, critical path set P and critical operand set are identified in phase-1. From the results of the first phase, a basic candidate solution set I_b={3,5,7,8} is obtained from the candidate loop body by reselectable instruction map table. The basic candidate solutions generate a set of solution space $\{\phi$, 3, 5, 7, 8, (3,5), (3,7), (3,8), (5,7), (5,8), (7,8), (3,5,7), (3,5,8), (3,7,8), (5,7,8), (3,5,7,8)$\}$. But, the branch and bound procedure discards the element 3 and 8 by heuristically chosen lower bound ,which we used 0.5, because those elements does not contribute an optimum solution. Remaining solution space has cost(5)=2/2=1, cost(7)=2/1=2, cost(5,7)=2/2=1. Consequently, the search module has returned a solution instruction I as instruction 7.

5.3 Phase 3: Apply $I \rightarrow I'$ and Iterate of Instruction Reselection Procedure

Step1: Reselection. To explain our technique to lower the $RecMII$ of recurrence circuits formed by loop-carried anti and/or output dependencies, consider the SC1400 instruction sequence in Figure 1.(c), which forms the excessive $RecMII$=5 of the half rate GSM candidate loop. The careful analysis on recurrence circuit, that exists in this candidate loop, leads us to realize that these loop-carried dependencies are nothing but artifacts from the scheduling-insensitive DSP code generator which selects instructions whose source and destination requires to be the same register.



(a) SC140 instructions before instruction reselection

(b) SC140 instructions after instruction reselection

(c) Data Dependence Graph (DDG) after instruction reselection

Fig. 5. GSM code applied the solution

In order to eliminate loop-carried anti and/or output dependencies, which being created from the use of these types of instructions for a candidate loop. We selectively undo the DSP code generation to recover scheduling freedom for a candidate loop in the following two steps. First, reselect a complex (or a compact) instruction by replacing it with a sequence of instructions that achieves semantically equivalent instructions. Second, place the output value of the dismantled instruction to a different register to eliminate original loop-carried anti and/or output dependencies.

As an illustration, the 7th *mac* instruction in Figure 5.(a) can be reselected into 7(a)th and the 7(b)th in Figure 5.(b) instructions with register renaming as shown in Figure 5.(b). When this modification is made, the loop-carried anti dependence from the 9th back to the 2nd instructions and the loop-carried output dependence from the 8th back to the 2nd instructions in Figure 5.(a) are both eliminated. Figure 5.(c) depicts the dismantled recurrence circuits. As a result, the original $RecMII = 5$ of the half rate GSM candidate loop is effectively lowered to 3 and thereby, the higher loop initiation rate 3 is achieved. The modified schedule results in 20% and 4.7% improvement of the SC1400 DSP resource utilization and loop performance respectively.

Since the preprocessing is iterated until the `II` of a DDG cannot be further lowered, all the possible opportunities are typically exhausted with proposed instruction reselection technique.

Step2: Iteration. All above sequence is done, then the recurrence circuit is reduced to lower EC element. If the procedure satisfies the Definition 15, overall procedure is repeated to resolve until bound condition is achieved for the best loop performance.

Definition 15. Bound Condition Constraint, *It is $RecMII > ResMII$ which condition determines whether the reselection technique completely resolves the Min-Max problem of a candidate loop or not. If this condition does not satisfy at any moment, the algorithm procedure must be stop. The reason is that $ResMII$ is to become a limiting factor of loop performance in iterative modulo scheduling.*

6 Experiment Result

This section describes the results of a set of experiments to illustrate the effectiveness of the iterative preprocessing algorithm described in previous section, which is implemented for the StarCore SC1400 V0.96 production `C` compiler. The experimental input is a set of candidate loops obtained from DSPStone, MediaBench, half-rate GSM, enhanced full rate GSM, and other industry signal application kernels. Table 1 lists the benchmarks used in the experiments.

In order to isolate the impacts on performance and code size purely from our preprocessing techniques, two sets of executables for the SC1400 multi-issue DSP are produced for the benchmarks listed in Table 1;

Table 1. Benchmarks used in the 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**	**MPEG2s**	**MPEG2r**	**GSMad**	**GSMut**	**GSMdec**	**GSMaf**
Program	Matrix1x3	MPEG2 decoder	MPEG2 decoder	Full rate GSM	Full rate GSM	Full rate GSM	Half rate GSM
Description	1x3 matrix multiply	Spatscal module	Recon module	Add module (a part of decoder)	Utcount module	Efsabser2if module	aflateRecursio n function (a part of decoder)

- **ORIG:** fully optimized one with original V0.96 production compiler, and
- **PRE:** fully optimized one with the revised V0.96 production compiler with our preprocessing techniques 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 techniques were measured in percent, using the formula $((\mathbf{PRE} - \mathbf{ORIG})/\mathbf{ORIG}) * 100$.

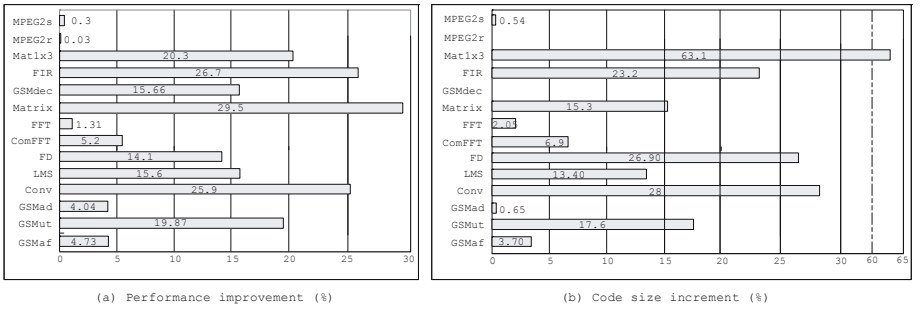


Fig. 6. Percent wise number of cycles reduction and code size increase compared to the original

Figure 6.(a) reports the performance improvements, which is based on the proposed technique. The overall performance improvement from the preprocessing techniques ranges from 0.03% to 29.5%, and the average performance improvement is 14.99%. Considering there is no modification made to the existing iterative modulo scheduler and the performance comparison is made to highly optimized SC1400 DSP code, the performance gain from our preprocessing was impressive.

Finally, none of the benchmarks in Figure 6.(a) reports the performance degradation. This is not a mishap, but due to the fact that our algorithm is designed to apply the techniques only when the additional operations for our preprocessing can be placed in non-`RecMII` recurrence circuits.

Figure 6.(b) reports the code size increase, which is based on proposed technique. Since the technique 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 the better modulo schedule. Considering the size of the prologue and epilogue grow proportionally as more loop iterations of the candidate loop get overlapped for the final schedule, the code size increase is unavoidable. However, due to our preprocessing to resolve restricted scheduling freedom, the existing scheduler achieves better modulo schedule with the same number of loop iterations. Therefore, there is no impact on the code size while achieving good improvement on performance such as MPEG2r, GSMdec, GSMad on Figure 6.(b). We had obtained two insight through this benchmarking as follows.

1. The techniques can take performance improvement without the overhead if it does not build more overlapped iterations in a candidate loop kernel. The reason is that

the code increment was composed almost of prologue and epilogue code that proportionally grow the number of overlapped iterations.
2. The code size impact can overwhelm the results of performance improvement in two cases of that the first is if a candidate loop code takes more than half of the whole code size, the second is if the execution time of a candidate loop code is relatively small to the total execution time such as Mar1x3, MPEG2s.

For the benchmarks listed in Figure 6.(b), the overall code size increase from the preprocessing techniques ranges from 0% to 63.1%, and the average increase is 14.38%. 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. In addition, we recognize a research on code size reduction technique for software-pipelined loops [7]. The reported solution is based on a novel retiming function to reduce prologue and epilogue code sections with predication. However, we did not employ this solution for our preprocessing techniques to reduce code size.

7 Related Work

The detrimental effects of excessive RecMII from loop-carried dependences were also noticed by Lam. In particular, she observed that excessive 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 excessive RecMII, Lam pioneered a compiler technique, referred to as Modulo Variable Expansion (MVE), that removes loop-carried anti and output dependences in recurrence circuits [3]. 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.

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 [4]. 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 [6]. 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.

8 Conclusion

This work has been motivated by our on-going project to build an optimizing compiler backend for a commercial multi-issue media processor under development. On our platform, we saw a variety of instructions specifically designed to accelerate media applications, and among them there were complex and compact instructions such

as imac, sxt.l. Not only those complex instructions may have good effect for their performance but also they make a dependence relation more complex than before. In our efforts to resolve this dependence constraint, we found that no previous compilers had addressed this optimization problem seriously before. For this reason, we opted to pursue our research to devise an effective algorithm that tackles this problem efficiently.

To address this particular problem, this paper describes compiler optimizations that preprocess embedded DSP applications loop kernels such that their intrinsic data dependences can be relaxed for effective modulo scheduling. The presented strategy is implemented for StarCore SC1400 version 0.96 production compiler backend. As a result of the implementation, 15% average runtime improvement is made for various signal processing applications.

References

1. J. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. In *Communications of the ACM*, pages 12-35, Dec 1970.
2. Douglas R. Smith. Random trees and the analysis of branch and bound procedures. In *Journal of the Association for Computing Machinery*, Jan 1984.
3. 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.
4. R. Huff. Lifetime-Sensitive Modulo Scheduling. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June, 1993.
5. B. Rau. Iterative modulo scheduling. In *HP Laboratories Technical Report, HPL94115*, Nov 1995.
6. 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.
7. Q. Zhuge, B. Xiao, and E. Sha. Code Size reduction technique and implementation for software-pipelined DSP applications. *ACM Transactions on Embedded Computing Systems*, Vol.2, No.4, pages 590-613, November 2003.