On Reducing Hidden Redundant Memory Accesses for DSP Applications

Meng Wang, Zili Shao, *Member, IEEE,* and Jingling Xue, *Senior Member, IEEE*

***Abstract*—Reducing memory accesses is particularly important for DSP applications since they are widely used in embedded systems and need to be executed with high performance and low power consumption. In this paper, we propose a machine-independent loop memory access optimization technique, REALM (REdundAnt Load Exploration & Migration), to explore hidden redundant load operations and migrate them outside loops based on loop-carried data dependence analysis. We implement REALM into IMPACT and Trimaran. To the best of our knowledge, this is the first work to implement** *the memory access reduction with loop-carried data reuse* **in real world compilers. We conduct experiments using a set of benchmarks from DSPstone and MiBench on the cycle-accurate VLIW simulator of Trimaran. The experimental results show that our technique significantly reduces the number of memory accesses.**

***Index Terms*—DSP applications, loop optimization, memory optimization, instruction scheduling.**

I. Introduction

The widening performance gap between processor and memory requires effective compiler optimization techniques for reducing memory accesses. This is particularly important for DSP (Digital Signal Processing) applications since they are widely used in embedded systems and need to be executed with high performance and low power consumption. On the other hand, loops are the most critical sections and consume most time and power for DSP applications. Therefore, memory access optimization for loops is vital for improving DSP performance. Computationally intensive loop kernels of DSP applications usually have a simple control-flow structure with a single-entry-single-exit and a single loop back edge. In this paper, thus, we develop a data-flow-graph-based loop optimization technique with loop-carried data dependence analysis to explore and eliminate hidden redundant memory accesses for loops of DSP applications.

Various techniques for reducing memory accesses have been investigated in previous work. Two classical compile-time optimizations, redundant load/store elimination and loop-invariant load/store migration [1]–[4], can reduce the amount of memory traffic by expediting the issue of instructions that use the loaded value. Most of the above optimization methods only consider removing existing explicit redundant load/store operations. Our technique can explore and eliminate hidden redundant loads across multiple loop iterations based on loop-carried data dependence analysis.

Over the last decade, memory-related issues have benefited from advances made in the fields of compilers [5]–[15] and high-level synthesis [16]–[24]. In loop optimization, many compiler transformation techniques have been applied to reduce memory accesses, such as loop partitioning [6], [25], array padding [26], partial redundancy elimination [27]–[30], scalar replacement [31] and array contraction [32], [33]. In this paper, we propose a data-flow-graph-based approach in which the code replacement pattern for eliminating the corresponding redundant load can be easily determined. Different from the above work, our data-flow-graph-based approach is more suitable for DSP applications which have loops with a simple control-flow structure.

Our work is closely related to *loop unrolling*. Loop unrolling [34]–[36] can be used to unfold a loop a few times to expose loop-carried data dependences among memory operations. However, with loop unrolling, how to determine the optimal unrolling factor is not known, and the code size expansion after unrolling is undesirable for embedded systems. Our technique can automatically exploit the loop-carried data dependences of memory operations using a graph, and achieve an optimal solution by removing all possible redundant memory accesses based on the graph. Moreover, our technique outperforms loop unrolling since it introduces little code size expansion.

In this paper, we propose a machine-independent loop memory access optimization technique, REALM (**RE**dund**A**nt **L**oad Exploration & **M**igration), to explore hidden redundant loads and migrate them outside loops. Our basic idea is to explore loop-carried data dependencies among memory operations. In our technique, hidden redundant loads are found and replaced with registers, and by using registers in such a way that we do not need prior memory accesses which are unchanged or unnecessary to be fetched again from memory over multiple loop iterations. In REALM, we first build up a data-flow graph to describe the inter-iteration data dependencies among memory operations. Then we perform code transformation by exploiting these dependencies with registers to hold the values of redundant loads and migrating these loads outside loops.

Our main contributions are summarized as follows.

- We study and address the memory access optimization problem for DSP applications which is vital both for improving performance and reducing memory power.

Manuscript received March 2009; revised November 2009.

Meng Wang and Zili Shao are with the Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong. Jingling Xue is with the School of Computer Science and Engineering at the University of New South Wales, Sydney, Australia.

The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF PolyU 5269/08E), the Hong Kong Polytechnic University (HK PolyU A-PJ17), the National 863 Program of China (2008AA01Z106), and the Australian Research Council Grant (DP0881330).

The corresponding author: Zili Shao (Email: cszlshao@comp.polyu.edu.hk. Phone: (852) 27667287.)

Different from the previous work, our technique can optimize both array-based and pointer-based code, which is very important as pointer arithmetic is widely used in DSP applications.
- We propose a data flow graph model to analyze loop-carried data dependencies among memory operations, and develop a technique called REALM (**RE**dund**A**nt **L**oad Exploration & **M**igration) for reducing memory accesses within the loop nests of programs. This approach is suitable for DSP applications which typically consist of simple loop structures.
- We propose a practical algorithm called RPMS_CP_REALM (register-pressure-aware modulo scheduling with critical-path-based REALM) to minimize register pressure and improve performance by combining REALM with modulo scheduling [37] that is widely used in DSP applications as a back-end software pipelining technique.

We implemented our techniques into IMPACT [2] and Trimaran [38]. To the best of our knowledge, this is the first work to implement *the memory access reduction with loop-carried data reuse* in real world compilers. This function is not implemented in various open-source compilers such as IMPACT [2], Trimaran [38], GCC, Open64, SUIF [39], and the TI DSP compiler (Code Composer Studio V3.3). In particular, REALM is implemented in IMPACT [2], and RPMS_CP_REALM, which combines REALM with modulo scheduling, is implemented in Trimaran [38] as modulo scheduling is part of the Trimaran suite.

We conducted experiments using a set of benchmarks from DSPstone [40] and MiBench [41] on the cycle-accurate VLIW simulator of Trimaran [38]. The experimental results show that our REALM technique achieves significant memory access reduction and performance improvement with little code size expansion compared with classical optimizations [1]–[4].

- For DSPstone [40], with the configuration of 32 registers, our REALM technique contributes to 22.52% reduction in the number of memory accesses, 12.61% improvement on ILP (instruction level parallelism) and a speedup of 1.13 on overall time performance with 1.43% increase in code size on average. For MiBench [41], with the same configuration, our REALM technique achieves an average of 8.3% reduction in the number of memory accesses, 4.43% improvement on ILP (instruction level parallelism) and a speedup of 1.06 on overall time performance with 0.77% increase in code size.
- RPMS_CP_REALM, our practical algorithm that combines REALM and modulo scheduling considering register pressure, can improve performance compared with the REALM technique. For DSPstone [40], with the configurations of 16, 32 and 64 registers, our RPMS_CP_REALM technique contributes to 2%, 4% and 6% improvement on average performance, respectively. For MiBench [41], with the configurations of 16, 32 and 64 registers, our RPMS_CP_REALM technique leads to 2%, 3% and 4% improvement on average performance, respectively. Based on the experimental results, we conclude that our RPMS_CP_REALM technique is the best with limited register resources.

The rest of this paper is organized as follows. The background is introduced in Section II. Motivational examples are shown in Section III. The REALM technique is presented in Section IV. The RPMS_CP_REALM technique is proposed in Section V. The experimental results and analysis are provided in Section VI. The conclusion is given in Section VII.

II. Background

DSP applications are generally characterized as computationally intensive with a large data set, loop-dominant control flow behavior, and accumulation-based operations [42]. Inside loop kernels of DSP applications, the control-flow structure is simple, and repetitive memory accesses to array elements have great impact on overall performance. Among all of the DSP benchmarks from DSPstone [40], almost every loop is in the form of a simple for-loop with a single-entry-single-exit and a single loop back edge. Except loop-back branches, there are no other conditional or unconditional branches inside these loops. Moreover, array-based and pointer-based computations are widely used in the loop kernel, and the array indexes in each loop are affine expressions of the loop index that is incremented unconditionally.

In loop kernels of DSP applications, one important characteristic is that the same memory location is repeatedly accessed by different memory operations across multiple loop iterations. By analyzing the inter-iteration relations among these memory operations, we can detect hidden redundant load operations in the innermost loop, and replace them by register operations as shown in Section III.

III. Motivational Examples

In order to show how our approach works, we present an example in this section. We use the IMPACT compiler [2] to generate intermediate code for this example and test it on the cycle-accurate VLIW (Very Long Instruction Word) simulator of Trimaran [38]. The C source code of the example is shown in Figure 1(a).

The intermediate code generated by the IMPACT compiler [2] with classical optimizations is presented in Figure 1(b). The code is in the form of Lcode which is a low-level machine-independent intermediate representation. Note that the "op" numbers represent the identifiers of operations, and they are not in sequence due to different optimization stages of the compiler. As shown in Figure 1(b), two different integer arrays A and C are stored in the memory with consecutive locations. The base pointer for array references in the loop is initialized using the address of C[2] and assigned to register r37 at the end of basic block 1 (Instruction: op59 add r37, mac $LV, -792). In the loop segment, the second array reference for C[i-1] in statement S2 in Figure 1(a) has been removed by performing classical optimizations. However, hidden redundant loads still exist in the intermediate code by analyzing inter-iteration data dependencies among memory operations. For example, as shown in Figure 1(b), op30 (Instruction: op30 ld_i r20, r37, 392; load A[i-2]) is redundant since it always loads data

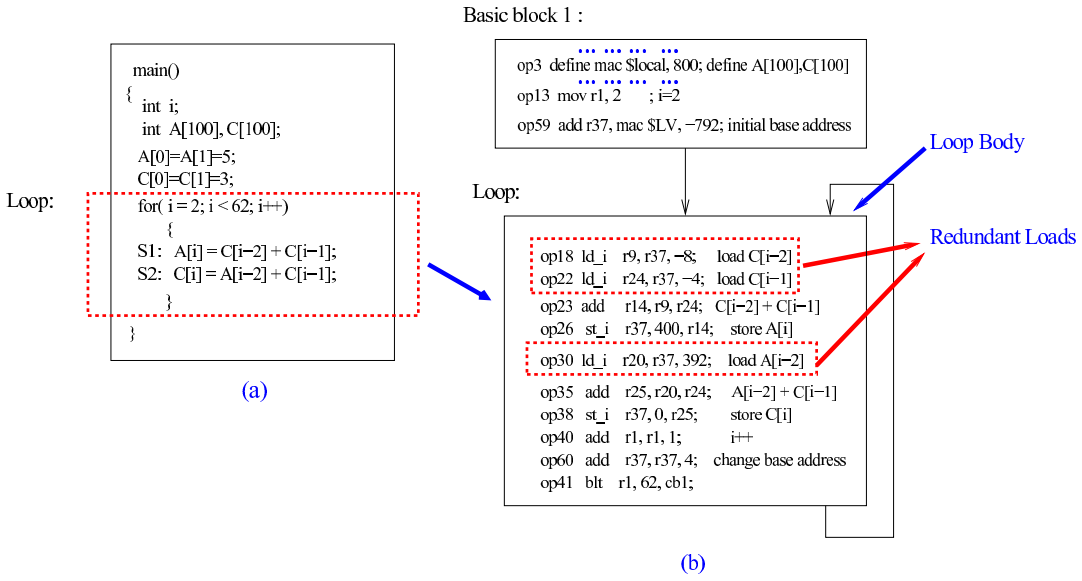

Fig. 1. The motivational example: (a) C source code; (b) The original intermediate code generated by the IMPACT compiler [2] after applying classical optimizations [1]–[4].

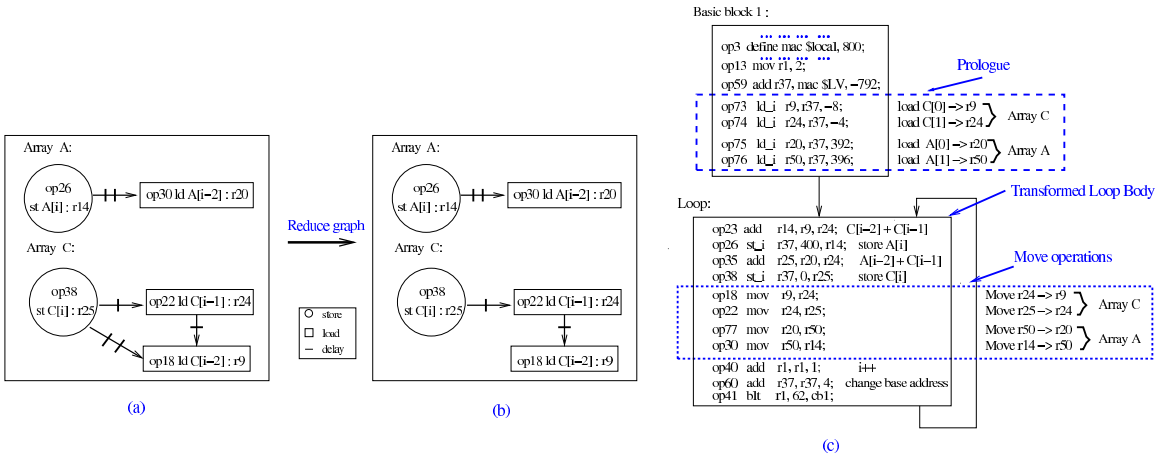

Fig. 2. The motivational example: (a) Data-flow graph; (b) the reduced data-flow graph; (c) The optimized code generated by the IMPACT [2] compiler after applying our technique.

from the memory location in which op26 (Instruction: op26 st_i r37, 400, r14; store A[i]) writes to before two iterations in the loop. All in all, there are 5 memory operations and 300 dynamic memory accesses in this example as the loop will be executed 60 times.

Motivated by this, we have developed the loop optimization technique, REALM, that further detects and eliminates redundant load operations across iterations. As shown in Figure 2(a) and Figure 2(b), we first build up a data-flow graph to describe the inter-iteration dependencies among memory operations for each array. For example, in the data-flow graph of array A as shown in Figure 2(a), the edge between the store op26 and the load op30 with two delays denotes that op30 always loads data from the memory location which was written by op26 two iterations ago. Thus, this redundant load op30 can be eliminated by exploiting register r14 which holds the value loaded from the memory location by op26 across two iterations. The data-flow graph is constructed using address-related operands of load/store operations. We reduce the graph in order to keep the correctness of computation and determine a definite code replacement pattern to eliminate the detected redundant load as shown in Figure 2(b).

Based on the constructed data-flow graph in Figure 2(b), our technique explores hidden redundant load operations and performs code transformation to eliminate them. The resultant code is presented in Figure 2(c). As shown in Figure 2(c), after our optimization, the redundant load operations are converted to register operations which are placed at the end of the loop body before the loop-back branch. With our technique, several iterations of the eliminated load operations are promoted into the prologue in order to provide the initial values of the registers used to replace the load in the loop. As shown in Figure 2,

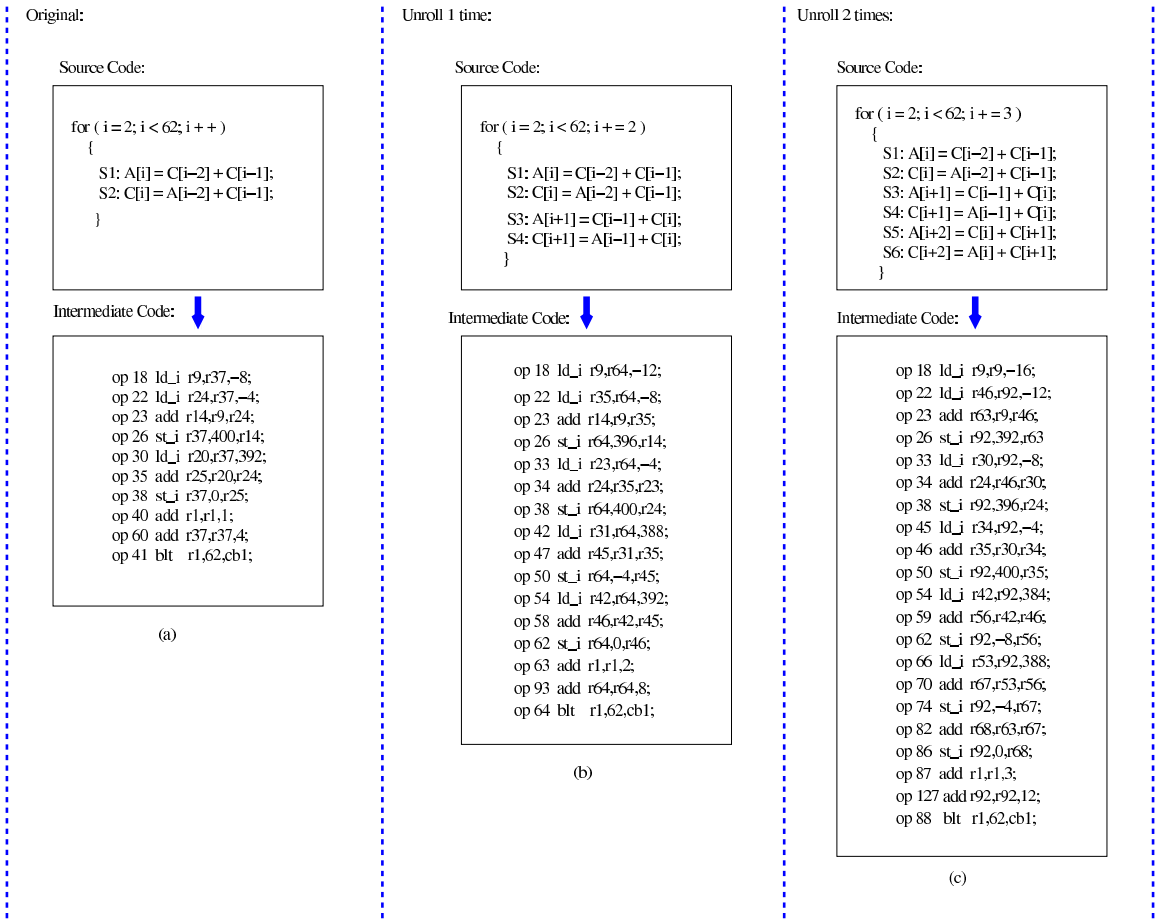

Fig. 3. The C code and intermediate code of the loop with the different unrolling factors: (a) 0 (the original loop); (b) 1; (c) 2.

by promoting load operations into the prologue, it provides more opportunities for compilers to do further optimizations for the basic block that contains the prologue. However, the operations promoted into the prologue will increase the code size and cause the performance overhead. To avoid big code size expansion, in our implementation, we restrict the maximum number of register operations and promoted load operations for reducing hidden redundant memory accesses. For this example, all of the three hidden redundant load operations, op18, op22, and op30 are completely removed by our technique. As the loop body runs for 60 times for this example, 180 dynamic memory accesses are eliminated. It shows that our approach can effectively reduce the number of memory accesses.

Next, we compare our technique against loop unrolling, which is widely used to optimize the loop performance at the expense of code size. The source and intermediate code generated by unrolling the original loop of Figure 3(a) with factor of 1 and 2 are shown in Figure 3(b) and (c), respectively. In the unrolled loop, loop-carried data dependences are changed to intra-iteration data dependences, and as a result, a few load operations are removed by the classical optimization techniques. However, as shown in Figure 3, the load operations cannot be fully removed with the unrolling factors of 1 and 2, and the best unrolling factor is hard to be decided. For example, there are 6 loads and 6 stores in Figure 3-(c) with the unrolling factor of 2, and there are 240 dynamic memory accesses as the unrolled loop will be executed 20 times. In other words, it only reduces the total dynamic memory accesses of the example by 60 which shows that our technique outperforms loop unrolling. This example also shows that our technique results in a much smaller code size expansion compared with that of loop unrolling.

Besides the significant memory accesses reduction in this example, our technique reduces the schedule length of the loop body as well. The schedule of the original loop in Figure 1(b) and that of the transformed loop in Figure 2(c) are shown in Table I(a) and Table I(b), respectively. The schedules are generated on the Trimaran [38] simulator, a VLIW based simulator that has multiple function units and can process several instructions simultaneously. The configurations of the simulator are as follows: 2 integer ALUs, 2 memory units and 1 branch unit (the detailed configurations are presented in Section VI). The reason of the reduction in schedule length is that data dependencies in the loop body are changed due to the elimination of redundant loads which are formerly on the critical path. And the register operations used to replace hidden redundant loads can be put into the empty slots with multiple function units of the VLIW architecture. From this example, we can see that our technique can effectively reduce

(a)

Time	Integer ALU		Memory Units		Branch Unit
	FU1	FU2	FU1	FU2	
0	op40		op18	op22	
1			op30		
2	op23				
3	op35		op26		
4	op60		op38		op41

(b)

Time	Integer ALU		Memory Units		Branch Unit
	FU1	FU2	FU1	FU2	
0	op23	op35			
1	op40	op18	op26	op38	
2	op22	op77			
3	op60	op30			op41
4					

TABLE I
THE SCHEDULES (A) FOR THE ORIGINAL LOOP IN FIGURE 1(B); (B) FOR THE OPTIMIZED LOOP IN FIGURE 2(C).

memory accesses and schedule length. Next, we will present our proposed technique.

IV. THE REDUNDANT LOAD EXPLORATION & MIGRATION ALGORITHM

In this section, we first propose the REALM (**RE**dund**A**nt **L**oad Exploration & **M**igration) algorithm in Section IV-A, and discuss its two key functions in Section IV-B and Section IV-C, respectively. Then we perform complexity analysis in Section IV-D.

A. The REALM Algorithm

Algorithm IV.1 Algorithm REALM.

Require: Intermediate code after applying all classical optimizations [1]–[4].

Ensure: Intermediate code with hidden redundant loads across different iterations eliminated.

1: Identify different arrays in the loop. For each array, put all of the load/store operations into the node set $V=\{v_1,v_2...v_N\}$ with their original order in the intermediate code, where N is the total number;
2: **for** each node set V **do**
3: Call function **Graph_Construction(V)** to build up the data-flow graph $G=\langle V, E, d\rangle$ of node set V in order to determine the inter-iteration dependencies among memory operations (discussed in Section IV-B).
4: Call function **Code_Transformation(V,G)** to eliminate hidden redundant loads of set V based on the data-flow graph G (discussed in Section IV-C).
5: **end for**

The REALM algorithm is designed to reduce hidden redundant memory accesses in loops of DSP applications. Our basic idea is to explore loop-carried data dependencies to replace hidden redundant loads with register operations. The registers are used in such a way that we do not need prior memory accesses which are unchanged or unnecessary to be fetched again over multiple loop iterations. The REALM algorithm is shown in Algorithm IV.1.

The input of our algorithm is the intermediate code after classical optimizations. In this paper, we select Lcode, the low-level intermediate code of IMPACT compiler [2], as the input. We choose IMPACT because it is an open-source compiler infrastructure with full support from the open-source community. Note that our technique is general enough and can be applied in different compilers.

The REALM algorithm consists of two steps. The first step is to obtain the memory operation sets for different arrays, and the second step is to perform optimizations on each set.

In step one, we first identify different arrays. In the following, we introduce our implementation. In IMPACT, the explicit information of array references is maintained in the high-level intermediate code, Hcode. The Hcode of statements S1 and S2 of the motivational example in Figure 1(a) is shown as follows:

S1: (**assign** (**var** *P_p_A*) (**var** *P_i*)) (**add** (**var** *P_q_C*) **sub** (**var** *P_i*) (2)) ((**var** *P_q_C*)) **sub** (**var** *P_j*) (1))

S2: (**assign** (**var** *P_q_C*) (**var** *P_i*)) (**add** (**var** *P_p_A*) **sub** (**var** *P_i*) (2)) ((**var** *P_q_C*)) **sub** (**var** *P_i*) (1))

In this code, arrays A and C are represented using pointer references "**var** P_p_A" and "**var** P_q_C", respectively. In our technique, we keep such information as the annotation of the operation data structure during the stage of intermediate code generation. For pointer-based code which is widely used in DSP applications, we adopt the alias analysis techniques to identify which array is pointed by which pointer, and pass the information to Lcode. Alias analysis is the preliminary step in our work based on points-to analysis algorithms [43]. We obtain the alias information to identify array accesses via pointers, and fed them into the annotations. Thus, we can identify memory operations of the same array by comparing the annotations of them. Based on this information, we put all memory operations of the same array into a node set $V=\{v_1,v_2...v_N\}$ with their original order in the intermediate code. Two memory operation sets V_A and V_C for arrays A and C of the motivational example in Figure 1 are shown as follows:

Memory operation set V_A *of array A :* {*op26 : st A[i] : r14, op30 : ld A[i-2] : r20*}

Memory operation set V_C *of array C :* {*op18 : ld C[i-2] : r9, op22 : ld C[i-1] : r24, op38 : st C[i] : r25*}.

In step two, we perform optimizations on each memory operation set. We first call function Graph_Construction() to build up the data-flow graph of the node set that describes the inter-iteration dependencies among memory operations. Then, function Code_Transformation() is used to perform code transformation on the intermediate code based on the data-flow graph. The details of these two key functions are shown in Section IV-B and Section IV-C below.

B. Function Graph_Construction()

Graph_Construction() is used to build a data-flow graph for each memory operation set with loop-carried data dependence

Algorithm IV.2 Function Graph_Construction().

Require: A memory operation set V={v_1,v_1...v_N}.
Ensure: A data-flow graph G=⟨V, E, d⟩.
// Get the node set of G**:**
1: Let the memory operation set V be the node set of G.
// Step 1: Data-flow graph Construction: (N **is the number of nodes in** V)
2: **for** i = 1 to N **do**
3: **for** j = 1 to N **do**
// Calculate the weight for each node pair (v_i, v_j)**:**
4: Calculate the weight for the node pair (v_i, v_j) as d($v_i \rightarrow v_j$)= distance / step, in which step is the value of base pointer for array references changing in every iteration and distance = address operand value of v_i - address operand value of v_j.
5: **if** d($v_i \rightarrow v_j$) > 0 && v_j is a load node **then**
6: Add an edge, $v_i \rightarrow v_j$, with the number of delays d($v_i \rightarrow v_j$), into the edge set E.
7: **end if**
8: **end for**
9: **end for**
// Step2: Data-flow graph Reduction:
10: Reduce the data-flow graph G obtained from Step 1 (Data-flow graph Construction) by calling function **Graph_Reduction(G)**.

analysis. Our basic idea is that inter-iteration dependencies among memory operations remain invariant if two memory operations access the same memory location among different iterations. Such relation can be exploited to eliminate the unnecessary memory accesses. And, register values which have been loaded from memory or newly generated to be stored can be reused in the next iterations without loading them from memory again. Therefore, we construct the data-flow graph for each memory operation set to describe the inter-iteration dependencies among load/store operations using Graph_Construction() as shown in Algorithm IV.2.

The input of Graph_Construction() is the memory operation set of an array, and the output is a weighted data-flow graph G. Data-flow graph G=⟨V, E, d⟩ is an edge-weighted directed graph, where V={v_1,v_1...v_N} is the node set including all memory operations of the same array, E is the edge set, and d(e) is a function to represent the number of delays for any edge $e \in$ E. Edges with delays represent inter-iteration data dependency while edges without delays represent intra-iteration data dependency. In this paper, the inter-iteration dependency between two memory operations denotes that the source node and the destination node operate on the same memory location among different iterations. The number of delays represents the number of iterations involved.

In Graph_Construction(), we first get the node set of data-flow graph G using the input memory operation set. Then, two steps, data-flow graph construction and data-flow graph reduction, are performed to build up the data-flow graph as shown below.

1) Data-Flow Graph Construction: We first calculate the weight for each node pair (v_i, v_j) in the first step of data-flow graph construction. It involves two parts of computation.

- The first part is the memory access distance calculation between two nodes v_i and v_j. In the intermediate code, memory operations consist of two operands: one is for memory address calculation and the other is to specify the register that the operation will use to load or store data. We obtain the memory access distance between two nodes by comparing the differences of their address-related operands. For example, the distance between op22 and op18 equals to 4 as shown in Figure 4(a).
- The second part is to acquire the step value of the base pointer for array references changes in every iteration. We obtain this value directly from the operands of the corresponding base pointer calculation operations for each array in the loop. For example, as shown in Figure 4(a), the step value equals to the third operand of operation op60 in which the base pointer r37 changes.

After finishing all of the required computations, we calculate the number of delays for each node pair using the memory distance between two nodes to divide the step value of that array (distance / step). Thus, we can determine across how many iterations the source node and the destination node will operate on the same memory location.

After the weight calculation, we add an edge, $v_i \rightarrow v_j$, with the number of delays d($v_i \rightarrow v_j$) into the edge set E when the weight between them is greater than zero. The positive value denotes that node v_j operates on the memory location where node v_i has operated on several iterations before. Thus, node v_j can be replaced by exploiting the register value of node v_i which is loaded from memory or stored in several iterations before.

Note that we focus on eliminating redundant loads by exploiting reusable register values across different iterations. So we only add one edge into the edge set when the destination node is a load node and the source node is either a store node or a load node. Therefore, the data-flow graph has the following properties:

- The store nodes can only have outgoing edges;
- There are no edges between two store nodes.

An example of data-flow graph construction is shown in Figure 4. For the memory operation set V_C of array C, after calculating the number of delays for each edge in Figure 4(b), we build up the data-flow graph shown in Figure 4(c). For example, the edge (op22 $\rightarrow$ op18) with one delay denotes that op18 always loads data from the same memory location as op22 writes to in the previous iteration. Thus, op18 can be replaced with the register that holds the value of op22 one iteration before.

2) Data-Flow Graph Reduction: After the graph is built in the first step of Graph_Construction(), in order to determine definite code replacement patterns to eliminate redundancy, we call function Graph_Reduction() to delete redundant edges for all loads with more than one incoming edges. Graph_Reduction() is shown in Algorithm IV.3.

In Graph_Reduction(), for each load, we keep the edge from the closest preceding memory operation, which has the latest produced values. We use two rules as shown below.

Rule 1:

- For each node, if it is a load node and has more than one incoming edges, keep the incoming edges with the minimum delays and delete all other incoming edges.

Rule 2:

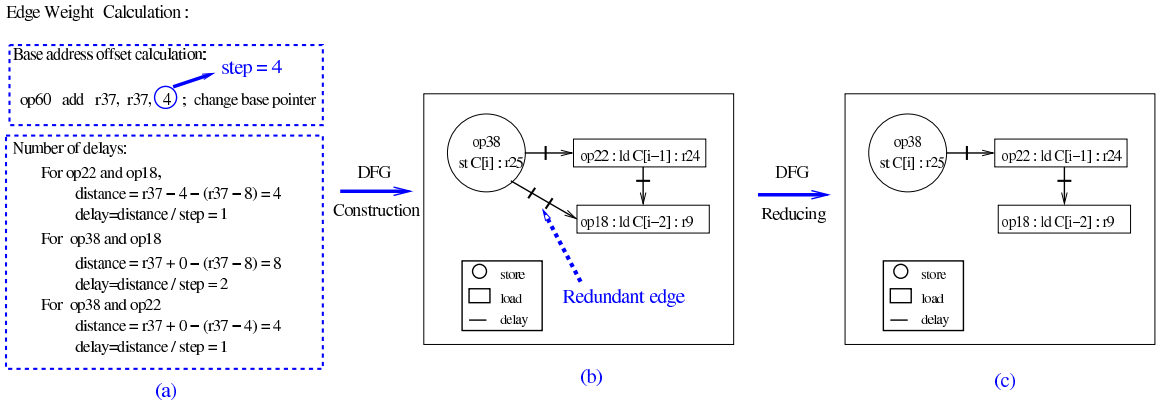

Fig. 4. Data-flow graph construction and reduction of array C in the motivational example: (a) edge weight calculation; (b) data-flow graph construction; (c) data-flow graph reduction.

Algorithm IV.3 Function Graph_Reduction().

Require: Data-flow graph G=⟨V, E, d⟩.
Ensure: The reduced data-flow graph G=⟨V, E, d⟩.

```
1:  for i = 1 to N do
2:    if ((v_i is load) && (v_i has more than one incoming edges)) then
3:      Keep the incoming edges with the minimum delays and delete all other incoming edges for v_i;
4:      if (v_i still has more than one incoming edges) then
5:        for each incoming edge of v_i do
6:          if (the source node is store) then
7:            Keep this edge and delete all other edges; Break;
8:          end if
9:        end for
10:     end if
11:   end if
12: end for
```

- After applying rule 1, if a load node still has more than one incoming edges, check the types of the source nodes of all incoming edges. If the source node of one edge is a store node, keep this edge and delete all other edges.

The reason for choosing **Rule 1** is that we can use the least registers to replace redundant loads by keeping the edges with the minimum delays. **Rule 2** needs to be applied because a store node updates the data of the specific memory location so we should use its register value to replace the redundant load. An example of how to reduce the data-flow graph is shown in Figure 4(c). According to rule 1, the edge (op38 → op18) is deleted as it has more delays than the edge (op22 → op18) for load op18.

After applying the above rules, we have the following properties for the reduced data-flow graph. The properties and their proofs are shown as follows.

Property IV.1. *For each load node in the reduced data-flow graph, there is only one incoming edge.*

Proof: After we reduce the data-flow graph according to **Rule 1,** only the incoming edges with the same weight remain. Among these edges, there are only two types of edges, each with either load or store as its source node. In other words, if there is more than one load (or store), these loads (or stores) must load (or store) data from (to) the same memory location. Therefore, these explicit redundant loads (stores) should have been eliminated by classical optimizations [1]–[4]. If they have not been, the classical redundant-memory-reduction optimization techniques will be applied to remove them. Then by applying **Rule 2,** only the incoming edge with store as its source node will be kept. So the property is proved. ■

Property IV.2. *There is no cycle in the reduced data-flow graph.*

Proof: Suppose there is a cycle in the reduced data-flow graph, and v_i and v_j are two different nodes in this cycle. As there is a path from v_i to v_j, v_i will operate on the same memory location several iterations earlier than v_j. Thus, when we calculate the weight for the node pair, (v_i, v_j), we can have the following result: $d(v_i \rightarrow v_j) > 0$. However, as there is also a path from v_j to v_i, the weight result for the node pair will become $d(v_i \rightarrow v_j) < 0$. Contradiction. ■

C. Function Code_Transformation()

Based on the data-flow graph constructed in Section IV-B, function Code_Transformation() is used to perform code transformation on the original intermediate code as shown in Algorithm IV.4.

In Code_Transformation(), we traverse the data-flow graph and eliminate redundant loads by replacing them with register operations. We use a bottom-up method to perform code replacement for each redundant load node and a node is only processed after all of its child nodes have been eliminated by our technique.

Our basic idea of code replacement is to replace redundant loads with register operations. Each redundant load is removed from the loop through two steps. First, we use register operations to replace a load and put them at the end of the loop before the loop-back branch. New registers are used to be operands of these register operations which shift the register value from the source node to the destination node across multiple loop iterations. Second, we put several iterations of the redundant load into the prologue. The purpose of promoting load into the prologue is to initialize the register

Algorithm IV.4 Function Code_Transformation().

Require: Intermediate code after performing classical optimizations, the memory operation set V={v_1,v_2...v_N} and the reduced data-flow graph G=⟨V, E, d⟩.

Ensure: Intermediate code with hidden redundant load operations eliminated.

1: **for** each node $v_i \in V$ (i=1 to N) **do**
2: Associate a boolean variable, $Mark(v_i)$, and set $Mark(v_i) \leftarrow False$;
3: Associate an integer variable $Dep(v_i)$, and set $Dep(v_i) \leftarrow$ The number of children of v_i;
4: **if** ((v_i is load) && (v_i has one incoming edge)) **then**
5: set $Mark(v_i) \leftarrow True$;
6: **end if**
7: **end for**
8: **while** there exists a node $v \in V$ whose ($Dep(v) == 0$ && $Mark(v) == True$) **do**
9: Let u be the parent node of v for edge "$u \rightarrow v$" with m delays. u uses r_u to load/store data from memory and v uses r_v to load data. Generate code with the following two steps:
10: **Step 1:** In the loop body, replace redundant load v with m register move operations and put them at the end of the loop body before the loop-back branch.
- When $m = 1$, convert load v to: move $r_u \rightarrow r_v$;
- When $m > 1$, convert load v to m register operations with the following order: move $r_1 \rightarrow r_v$, move $r_2 \rightarrow r_1$, ... , move $r_u \rightarrow r_{m-1}$ in which "$r_1, r_2, ..., r_{m-1}$" are newly generated registers.

11: **Step 2:** Promote the first m iterations of v into prologue which is at the end of the previous block of the loop with the following order: 1st iteration of $v \rightarrow r_v$, 2nd iteration of $v \rightarrow r_1$, ... , mth iteration of $v \rightarrow r_{m-1}$;
12: Set $Mark(v) \leftarrow False$ and calculate $Dep(u) \leftarrow Dep(u) - 1$ for v's parent u.
13: **end while**

values that will be used in the loop. In the intermediate code, the prologue is put to the end of the previous basic block of the loop. For each redundant load, both the number of iterations to be promoted into prologue and the number of move operations to be generated in the new loop body are determined by the number of delays of its incoming edge.

D. Complexity Analysis

In the REALM technique, let M be the number of arrays and N be the number of load/store operations for each array in the loop. In the first step of the REALM algorithm, it takes at most $O(MN)$ to obtain the node sets. In function Graph_Construction(), for the node set of an array, it takes at most $O(N^2)$ to construct the data flow graph among N nodes, and it takes at most $O(N^2)$ to traverse the graph and delete the redundant edges. In function Code_Transformation(), we can find the number of children for N nodes in $O(N^2)$, and it takes at most $O(N)$ to finish code replacement. Totally, for M arrays, the REALM technique can be finished in $O(MN^2)$.

V. A Practical Register-Pressure-Aware REALM Technique

In practice, the REALM technique requires a large number of registers that may degrade the performance. Modulo scheduling [37], a back-end software pipelining technique that is widely used in DSP applications, also increases register pressure. In this section, we combine software pipelining with the REALM technique, and propose a practical algorithm called RPMS_CP_REALM (register-pressure-aware modulo scheduling algorithm with critical-path-based REALM) to minimize register pressure and improve performance.

Software pipelining technique has been proposed for exploiting the instruction level parallelism of loops by overlapping the execution of successive iterations. It imposes high register requirements as the lifetime of loop variables may cross the boundary of iterations. Gao et al. [44] proposed a software pipelining technique to improve performance while minimizing register requirements. Llosa et al. [45] presented a heuristic approach for resource-constrained software pipelining with reduced register pressure. The scheduling part of RPMS_CP_REALM technique is based on the heuristic approach in [45]. In the following, we first introduce the background of modulo scheduling. Then we present the RPMS_CP_REALM algorithm.

A. Background of Modulo Scheduling

The objective of modulo scheduling [37] is to compute a schedule for one iteration of the loop such that when this same schedule is repeated at regular intervals, no intra- or inter-iteration dependence is violated, and no resource usage conflict arises between operations of either the same or distinct iterations. This constant interval between the start of successive iterations is termed the initiation interval (II).

The minimum initiation interval (MII) is a lower bound on the smallest possible value of II for which a modulo schedule exists. The MII must be equal to or greater than both the resource-constrained MII (ResMII) and the recurrence-constrained MII (RecMII). The candidate II is initially set to the MII and increased until a legal modulo schedule is found. The modulo scheduling algorithm iteratively finds a legal schedule before giving up the current II. To determine an order for the nodes to be scheduled, a height-based priority function is used to define a topological sort of the nodes, which takes into account both intra- and inter-iteration data dependencies. The priority for each node is calculated using the longest path from the node to the leaf node of the data dependence graph. With this function, a node will have a higher priority than its successors.

B. RPMS_CP_REALM Algorithm

The RPMS_CP_REALM algorithm is shown in Algorithm V.1. In the input, TC is the given upper bound of schedule length, and BudgetRatio denotes how many schedule attempts we will try to get a legal schedule before giving up the current II.

The first step is to perform the critical-path-based REALM technique for eliminating hidden redundant load operations along the critical path with the minimum cost. Our basic idea is to reduce memory accesses along the critical path while minimizing the number of registers used to eliminate them. In our technique, each redundant load along the critical path is associated with a cost, which is calculated using the number

Algorithm V.1 Algorithm RPMS_CP_REALM.

Require: Data Dependence Graph DDG=⟨V_G, E_G, d_G⟩ of the input loop, the timing constraint TC, BudgetRatio.
Ensure: The modulo schedule with reduced register pressure.
// A practical critical-path-based REALM technique:
Find CP, a critical path of the DDG.
Perform REALM to detect hidden redundant load operations of the DDG.
while there exists hidden redundant load operations along CP **do**
  Calculate the cost of each redundant load along CP.
  Use REALM to replace the redundant load operation with the minimum cost.
  DDG = The changed Data Dependence Graph.
  CP = Find the critical path of DDG.
end while
// register-pressure-aware modulo scheduling:
Initialize the value of II to the Minimum Initiation Interval II := MII().
while II < TC **do**
  Budget := BudgetRatio * NumberofOperations;
  Compute priorities for each node in DDG and put them into the list.
  while (the list of unscheduled operations is not empty) & (Budget > 0) **do**
    Pick up the node with highest priority and find valid time slot for it;
    Budget:= Budget - 1;
  end while
  II := II+1;
end while

of registers required to replace it. The redundant load with the minimum cost denotes that this load requires the least number of registers to be replaced with. When the critical path contains no redundant load operations, this step finishes and the generated data dependence graph becomes the input of modulo scheduling.

The second step is to perform the register-pressure-aware modulo scheduling algorithm to minimize register lifetime. In the scheduling algorithm, we order the node list and schedule nodes using as early/late as possible schemes depending on their previously scheduled predecessors/successors in the partial schedule. The register lifetime is thus reduced compared with the conventional top-down scheduling approach [37]. Different from the technique in [37], we first put all nodes along the critical path of DDG into the node list with their height-based priority, and then we put other nodes into the list with their height-based order. When an operation is to be scheduled, it is scheduled using different schemes depending on its predecessors and successors in the partial schedule. If an operation has only predecessors in the partial schedule, then it is scheduled as early as possible (ASAP). If an operation has only successors, then it is scheduled as late as possible (ALAP). For other cases, we employ the same schedule schemes as in [45]. As the critical-path-based REALM will take at most O(N) to finish assuming that there are N load operations, the complexity of algorithm RPMS_CP_REALM is bounded by modulo scheduling [37].

VI. Experiments

We have implemented our technique into the IMPACT compiler [2] and conducted experiments using a set of benchmarks from DSPstone [40] and MiBench [41] on the cycle-accurate VLIW simulator of Trimaran [38]. In this section, we first discuss our implementation and simulation environment in section VI-A, and then introduce our benchmark programs in section VI-B. The experimental results and discussion are presented in section VI-C.

A. The Implementation and Simulation Platform

Our experimental platform is shown in Figure 5. The back-end of the IMPACT infrastructure has a machine-independent optimization component called Lopti [46] which performs classical optimizations. Our optimization technique is applied on Lcode, a low-level machine-independent intermediate code. We have implemented the REALM algorithm into IMPACT for code generation. Major modifications are performed to integrate our technique into the loop optimization module of IMPACT.

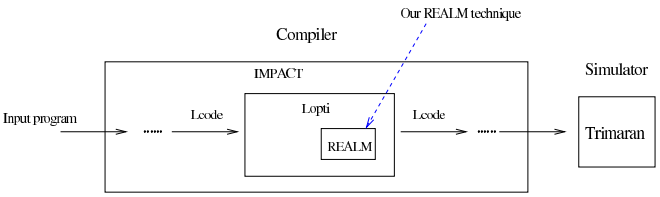

Fig. 5. The implementation and simulation framework.

Parameter	Configuration
Functional Units	2 integer ALU, 2 floating point ALU, 2 load-store units, 1 branch unit, 5 issue slots
Instruction Latency	1 cycle for integer ALU, 1 cycle for floating point ALU, 2 cycles for load in cache, 1 cycle for store, 1 cycle for branch
Register file	32 integer registers, 32 floating point registers

TABLE II
The configurations of Trimaran.

To compare our technique with classical optimizations [1]–[4], we use the Trimaran [38] infrastructure as our test platform. The configurations for the Trimaran simulator are shown in Table II. The memory system consists of a 32K 4-way associative instruction cache and a 32 K 4-way associative data cache, both with 64 byte block size. In the system, there are 32 integer registers and 32 floating point registers.

B. Benchmark Programs

To evaluate the effectiveness of our algorithm, we choose a suite of 21 benchmarks which are the only ones with loops and hidden redundant load operations from DSPstone [40] and MiBench [41]. We test both the fixed-point and the floating-point versions of benchmarks from DSPstone [40]. (Two benchmarks, fft_stage_scaled and fft_input_scaled only

Benchmark	Source	Benchmark Description	Benchmark	Source	Benchmark Description
Convolution	DSPstone	Convolution	fft_stage_scaled	DSPstone	integer stage scaling FFT
dot_product	DSPstone	dot product	fft_input_scaled	DSPstone	integer input scaled FFT
IIR	DSPstone	IIR filter	bfencrypt	Mibench	blowfish encrypt
fir	DSPstone	finite response filter	bfdecrypt	Mibench	blowfish decrypt
fir2dim	DSPstone	2D finite response filter	cjpeg	Mibench	JPEG compress
lms	DSPstone	least mean square	djpeg	Mibench	JPEG decompress
matrix1*3	DSPstone	matrix 1*3	gsmencode	Mibench	GSM encode
matrix	DSPstone	product of two metrices	gsmdecode	Mibench	GSM decode
matrix2	DSPstone	revised matrix	rawcaudio	Mibench	ADPCM encode
n_complex_updates	DSPstone	n complex updates	rawdaudio	Mibench	ADPCM decode
n_real_updates	DSPstone	n real updates			

TABLE III
THE BENCHMARKS.

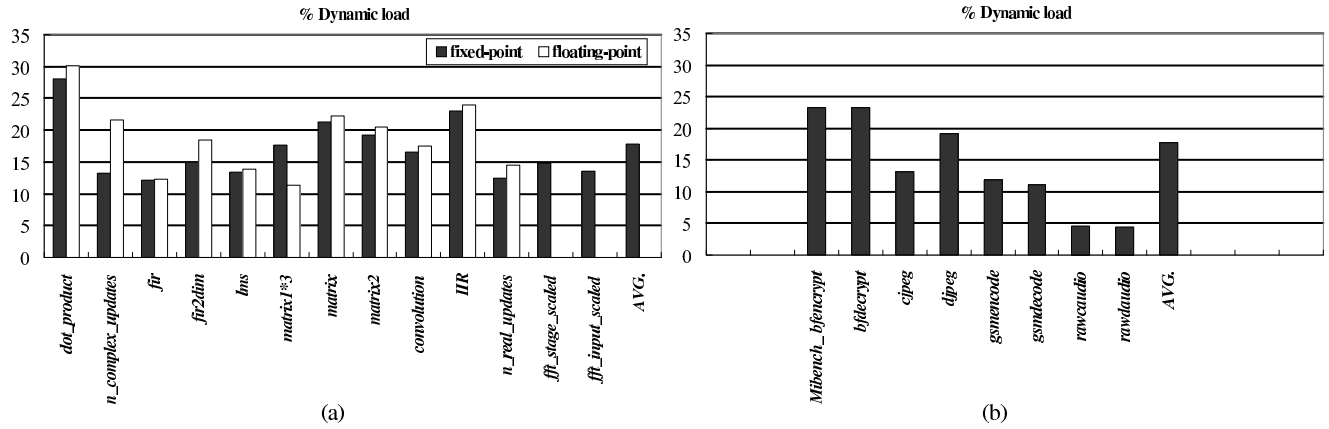

Fig. 6. The percentages of dynamic load operations over the total number of dynamic operations for the benchmarks from (a) DSPstone; (b) MiBench.

have the fixed-point version.) The details of benchmarks are show in Table III.

For each benchmark, we first generate code using the IMPACT compiler, and test the code on the simulator of Trimaran. The percentages of the number of dynamic load operations over the total number of dynamic operations for benchmarks from DSPstone and MiBench are shown in Figures 6 (a) and (b), respectively. It can be observed that, a large fraction of the dynamic operations, on average, 17.79% and 13.86% are dynamic load operations for DSPstone and MiBench, respectively. This shows that further memory access optimizations are necessary for improving performance.

C. Results and Discussions

In this section, we first present the results obtained by our REALM technique, and compare them with the baseline scheme of the IMPACT compiler [2] in section VI-C1. Then we compare our RPMS_CP_REALM with REALM in terms of average performance improvement with various register resources in section VI-C2.

1) REALM vs. Baseline: In this section, we compare our REALM approach with the baseline scheme of IMPACT. In the experiments, we set up the maximum number of delays adopted to determine the code replacement pattern as 4 to avoid big code expansion. With this constraint, for one redundant load operation, our technique will use at most four registers to replace it. In the following, we present and analyze the results in terms of memory access reduction, ILP improvement, overall performance improvement and code size expansion.

Memory Access Reduction. The percentages of memory access reduction for benchmarks from DSP stone and MiBench are shown in Figures 7 (a) and (c), respectively. In Figure 7 (a), the results for fixed-point and floating-point benchmarks from DSPstone [40] are presented in bars with different colors, and the right-most bar "AVG." is the average result.

Our REALM algorithm reduces memory accesses by exploring hidden redundant loads with loop-carried data dependence analysis and eliminating them with register operations. Moreover, more redundant load operations in the prologue can be further eliminated by performing classical optimizations with the output of our algorithm. The experimental results show that our algorithm significantly reduces the number of memory accesses. Compared with classical optimizations, on average, our algorithm achieves 22.52% and 8.3% reduction for the benchmarks from DSPstone and MiBench, respectively.

ILP Improvement. Our technique improves ILP for each benchmark. In our experiments, ILP refers to the average number of issued operations per cycle. As shown in Figures 7(b) and (d), on average, the results show that our technique achieves 12.61% and 4.43% improvement for the benchmarks from DSP stone and MiBench, respectively. The reason below is that our technique replaces redundant load operations with register operations. As a result, the data dependence graph is changed and these operations can be put into the available

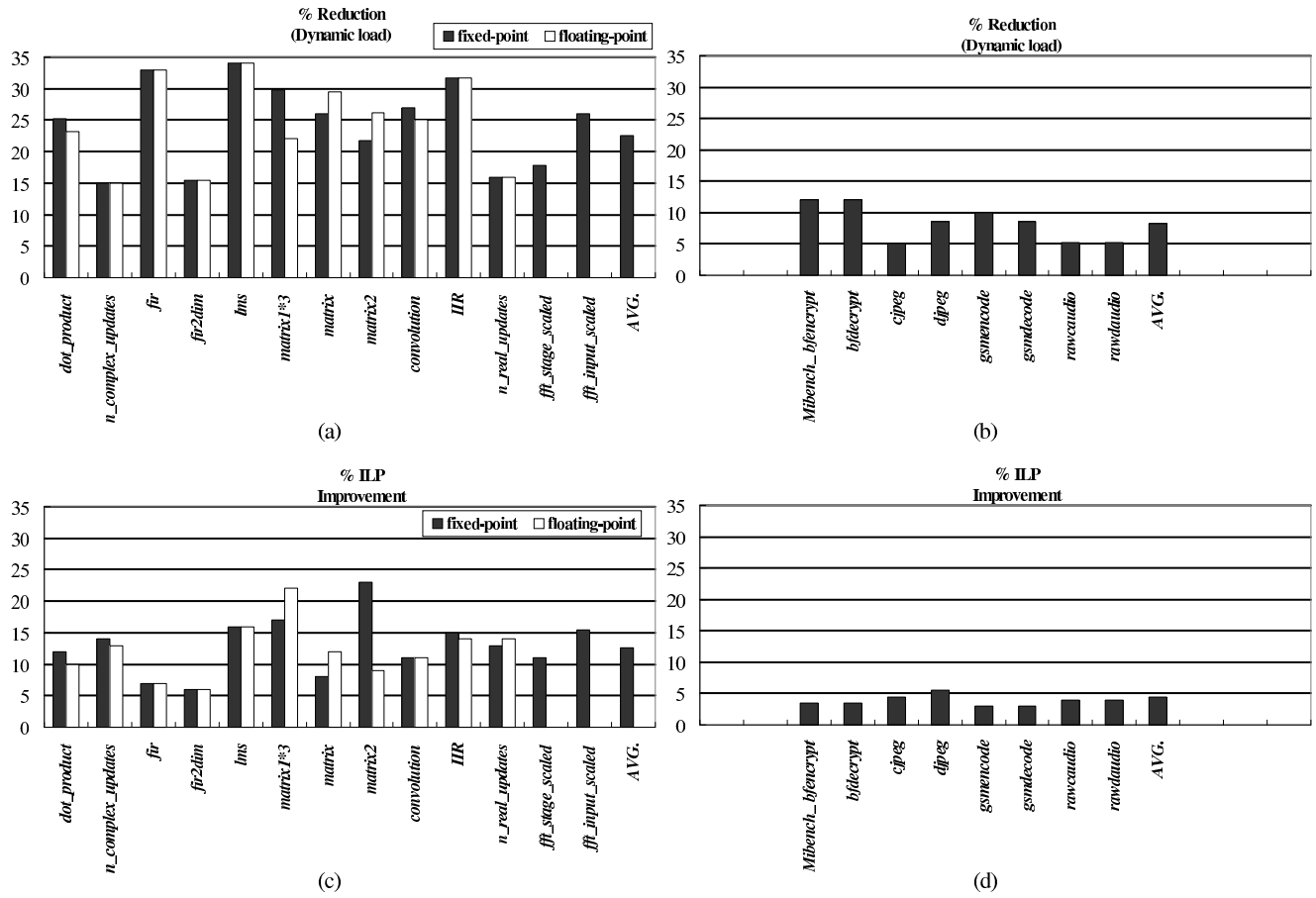

Fig. 7. The reduction in the number of dynamic load operations for the benchmarks from (a) DSPstone; (b) MiBench. The improvement of ILP for the benchmarks from (c) DSPstone; (d) MiBench.

empty slots of the multiple functional units on the VLIW architecture. Thus, the number of executed operations per cycle is increased.

Overall Performance Improvement. The overall performance improvement for benchmarks from DSPstone and MiBench are shown in Figures 8 (a) and (c), respectively. The speedup numbers are normalized based on the result of the original code generated by IMPACT. On average, the results show that our algorithm achieves a speedup of 1.13 and 1.06 for benchmarks of DSPstone and MiBench, respectively.

From the results of the benchmarks in DSPstone, we can observe that our technique leads to 13% performance improvement. The percentage of dynamic memory operations over total operations is 17.79% (shown in section VI-B), and our technique can reduce 22.52% of dynamic memory operations. Thus, we can get the percentage of reduction in dynamic memory operations over total operations using the product of the above two values, which is 4%. In the following, we present the reasons for this 13% performance improvement caused by only 4% reduction in memory operations. First, our technique eliminates load operations within loops that are the most time consuming part of DSP applications. Second, load operations are usually on the critical path of the execution for loop kernels of DSP applications. After performing our algorithm, the redundant load operations across different iterations are replaced by register operations with less execution time. Thus, the data dependencies in the loop are changed accordingly. As fewer redundant load operations remain in the loop, more operations which depend on them previously can be scheduled earlier in the new loop. This leads to the reduction of the schedule length of the loop.

The results also show that the performance gain for benchmarks in MiBench (6%) is smaller than that of DSPstone. The reason is that we eliminate about 1% of dynamic operations which is relatively small compared with the big code size of benchmarks in MiBench.

Code Size Expansion. The percentages of code size expansion for DSPstone and MiBench are shown in Figures 8 (b) and (d), respectively. On average, the results show that our technique leads to 1.43% and 0.77% expansion for the code size of benchmarks in DSPstone and MiBench, respectively.

The reason of the expansion is that our technique may use more than one register operation to replace one redundant load operation. However, in our technique, the code size expansion is controlled by the maximum number of delays that determines the maximum number of register operations used to replace one redundant load operation. In the experiment, the maximum number of delays is set as 4. Therefore, the code size expansion is very small. With such small code size expansion, our technique is suitable for embedded systems.

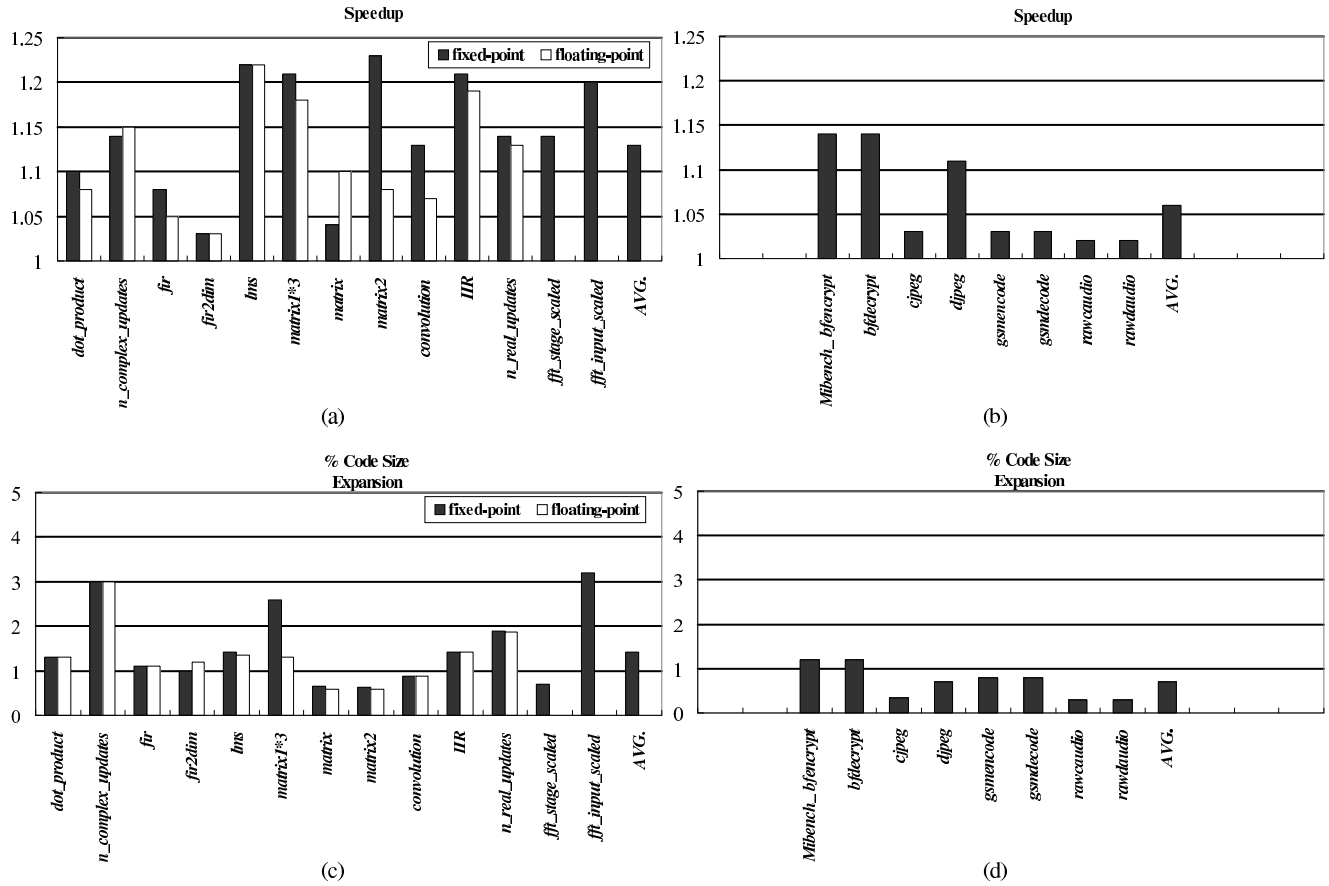

Fig. 8. Performance improvement due to the REALM algorithm for the benchmarks from (a) DSPstone; (b) MiBench. The code size expansion of the benchmarks from (c) DSPstone; (d) MiBench.

2) RPMS_CP_REALM vs. REALM: In this section, we compare the average performance speedup results achieved by the baseline scheme of IMPACT, our REALM technique with conventional modulo scheduling (*REALM_MS*) and our *RPMS_CP_REALM* technique using the configurations of 16, 32 and 64 registers. The speedup numbers are normalized based on the results of the baseline scheme in IMPACT. The results for the benchmarks of DSPstone and MiBench are shown in Figures 9 (a) and (b), respectively.

The results show that our REALM technique achieves bigger performance improvements with more register resources. And our practical RPMS_CP_REALM technique, that combines software pipelining and REALM, performs better than using REALM as it reduces register requirements. Compared with the REALM technique, for DSPstone, with the configurations of 16, 32 and 64 registers, our RPMS_CP_REALM technique contributes to an average of 2%, 4% and 6% performance improvement, respectively; For MiBench, with the configurations of 16, 32 and 64 registers, on average, our RPMS_CP_REALM technique achieves 2%, 3% and 4% performance improvement, respectively. Based on the experimental results, we conclude that our RPMS_CP_REALM technique is the best with limited register resources.

The reasons of the performance improvement achieved by RPMS_CP_REALM over REALM are as follows. First, the critical path of the loop is changed as redundant load operations along critical path with minimum cost are replaced. Second, with lower increment of register replace requirements and reduced register lifetime, the extra code spilling that may degrade performance during register allocation is alleviated.

VII. Conclusion and Future Work

In this paper, we proposed the machine-independent loop optimization technique REALM to eliminate redundant load operations of loops for DSP applications. In our approach, we built the data-flow graph by exploiting the loop-carried dependencies among memory operations. Based on the constructed graph, we performed code transformation to eliminate redundant loads. Finally, we proposed a practical algorithm to reduce register pressure and improve performance by combining software pipelining and REALM. We implemented our techniques into IMPACT [2] and Trimaran [38], and conducted experiments using a set of benchmarks from DSPstone [40] and MiBench [41] based on the cycle-accurate simulator of Trimaran [38]. The experimental results showed that our technique significantly reduces the number of memory accesses compared with classical optimizations [1]–[4].

There are several directions for future work. First, registers are critical resources in embedded systems. How to combine our techniques, instruction scheduling, and register life-time analysis together to effectively reduce memory accesses under tight register constraints is one of the future work. Second,

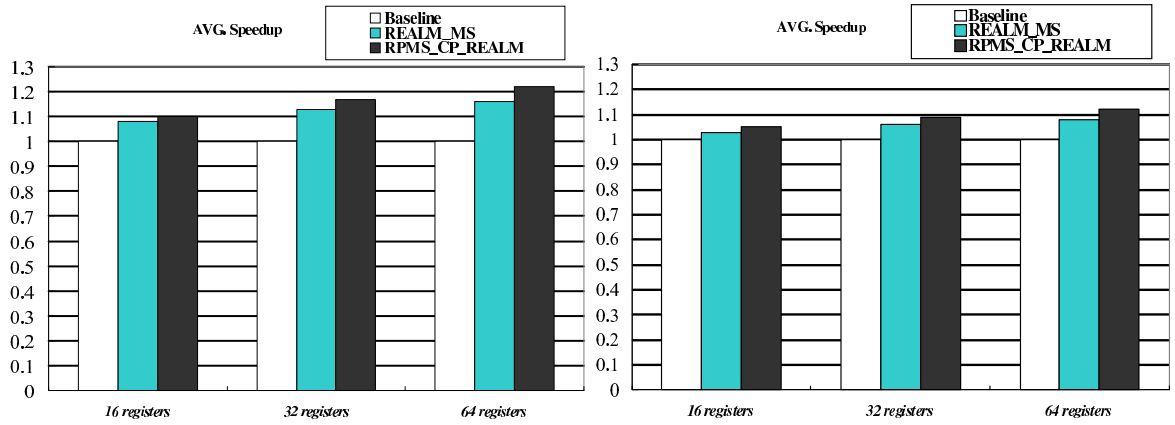

Fig. 9. Average performance speedup of REALM and RPMS_CP_REALM with 16 registers, 32 registers and 64 registers for the benchmarks from (a) DSPstone; (b) MiBench.

our techniques currently work well for DSP applications with simple control flow. How to extend our approaches to general-purpose applications with complicated control branches is another important problem of the future work. Third, in embedded systems, energy and thermal are important issues. How to evaluate the energy consumption of our techniques, and how to combine our techniques with efficient energy optimization techniques are important problems we need to investigate in the future.

REFERENCES

[1] A. Aho, M. S. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools — Second Edition*. Addison-Wesley, 2007.

[2] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "Impact: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, 1991, pp. 266–275.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 345–420, 1994.

[4] P. R. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 6, no. 2, pp. 149–206, 2001.

[5] Y. Song, R. Xu, C. Wang, and Z. Li, "Data locality enhancement by memory reduction," in *Proceedings of the 15th ACM International Conference on Supercomputing*. ACM Press, 2001, pp. 50–64.

[6] Y. Ding and Z. Li, "A compiler scheme for reusing intermediate computation results," in *Proceedings of the 2004 Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'04)*, 2004, pp. 279–291.

[7] L. Liu, Z. Li, and A. H. Sameh, "Analyzing memory access intensity in parallel programs on multicore," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008, pp. 359–367.

[8] M. Kandemir, "A compiler-based approach for improving intra-iteration data reuse," in *Proceedings of the conference on Design, automation and test in Europe*, 2002, pp. 984–990.

[9] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "Data reuse analysis technique for software-controlled memory hierarchies," in *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, 2004, pp. 202–207.

[10] J. Yan and W. Zhang, "Exploiting virtual registers to reduce pressure on real registers," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 4, January 2008.

[11] Y. Jun and W. Zhang, "Virtual registers: Reducing register pressure without enlarging the register file," in *Proceedings of the 2007 International Conference on High Performance Embedded Architectures & Compilers*, 2007, pp. 57–70.

[12] S. Leventhal, L. Yuan, N. K.Bambha, S. S.Bhattacharyya, and G. Qu, "DSP address optimization using evolutionary algorithms," in *Proceedings of the 2005 workshop on Software and compilers for embedded systems*, 2005, pp. 91–98.

[13] M. Ko, C. Shen, and S. S. Bhattacharyya, "Memory-constrained block processing for DSP software optimization," *Journal of Signal Processing Systems*, vol. 50, no. 2, pp. 163–177, February 2008.

[14] P. Salmela, R. Gu, S. S. Bhattacharyya, and J. Takala, "Efficient parallel memory organization for turbo decoders," in *Proceedings of the European Signal Processing Conference*, 2007, pp. 831–835.

[15] R. Leupers and P. Marwedel, "Time-constrained code compaction for DSPs," in *Proceedings of the 8th international symposium on System synthesis*, 1995, pp. 54–59.

[16] D. Kolson, A. Nicolau, and N. Dutt, "Elimination of redundant memory traffic in high-level synthesis," *IEEE Transactions on Computer-aided Design*, vol. 15, no. 11, pp. 1354–1363, 1996.

[17] C. Huang, S. Ravi, A. Raghunathan, and N. K. Jha, "Generation of heterogeneous distributed architectures for memory-intensive applications through high-level synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 11, pp. 1191–1204, 2007.

[18] K. S. Khouri, G. Lakshminarayana, and N. K. Jha, "Memory binding for performance optimization of control-flow intensive behavioral descriptions," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 5, pp. 513–524, 2005.

[19] C. Huang, S. Ravi, A. Raghunathan, and N. K. Jha, "Eliminating memory bottlenecks for a JPEG encoder through distributed logic-memory architecture and computation-unit integrated memory," in *Proceedings of the IEEE Custom Integrated Circuit Conference*, Sept. 2005, pp. 239–242.

[20] Z. Wang, T. W. O'Neil, and E. H.-M. Sha, "Optimal loop scheduling for hiding memory latency based on two level partitioning and prefetching," *IEEE Transactions on Signal Processing*, vol. 49, no. 11, pp. 2853–2864, 2001.

[21] Q. Wang, N. Passos, and E. H.-M. Sha, "Optimal loop scheduling for hiding memory latency based on two level partitioning and prefetching," *IEEE Transactions on Circuits and Systems II - Analog and Signal Processing*, vol. 44, no. 9, pp. 741–753, 1997.

[22] Z. Wang, E. H.-M. Sha, and Y. Wang, "Partitioning and scheduling DSP applications with maximal memory access hiding," *EURASIP Journal on Applied Signal Processing*, vol. 2002, no. 1, January 2002.

[23] J. Seo, T. Kim, and P. R. Panda, "Memory allocation and mapping in high-level synthesis: An integrated approach," *IEEE Transactions on VLSI Systems (T-VLSI)*, vol. 11, no. 5, pp. 928–938, October 2003.

[24] P. R. Panda, N. Dutt, A. Nicolau, F. Catthoor, A. Vandecappelle, E. Brockmeyer, C. Kulkarni, and E. de Greef, "Data memory organization and optimizations in application-specific systems," *IEEE Design and Test of Computers*, vol. 18, no. 3, pp. 56–68, 2001.

[25] Y. Song and Z. Li, "Applying array contraction to a sequence of doall loops," in *Proceedings of the 2004 International Conference on Parallel Processing*, 2004, pp. 46–53.

[26] Z. Wang, E. H.-M. Sha, and X. S. Hu, "Combined partitioning and data padding for scheduling multiple loop nests," in *Proceedings of the 2001*

International conference on Compilers, architecture, and synthesis for embedded systems, 2001, pp. 67–75.
[27] R. Bodik, R. Gupta, and M. L. Soffa, "Load-resue analysis: Design and evaluation," in *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999, pp. 64–76.
[28] J. Xue and Q. Cai, "A lifetime optimal algorithm for speculative PRE," *ACM Transactions on Architecture and Code Optimization*, vol. 3, no. 2, pp. 115–155, 2006.
[29] J. Xue and J. Knoop, "A fresh look at pre as a maximum flow problem," in *Proceedings of the 2006 International Conference on Compiler Construction (CC'06)*, 2006, pp. 139–154.
[30] Q. Cai and J. Xue, "Optimal and efficient speculation-based partial redundancy elimination," in *Proceedings of the 1st Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'03)*, 2003, pp. 91–102.
[31] D. Callahan, S. Carr, and K. Kennedy, "Improving register allocation for subscripted variables," in *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1990, pp. 53–65.
[32] V. Sarkar and G. R. Gao, "Optimization of array accesses by collective loop transformations," in *Proceedings of the 5th international conference on Supercomputing*, 1991, pp. 194–205.
[33] G. R. Gao, "A maximally pipelined tridiagonal linear equation solver," *Journal of Parallel and Distributed Computing*, vol. 3, no. 2, pp. 215–235, June 1986.
[34] L. E. L. A. P. Brown and K. K. Parhi, "Unfolding and retiming for high-level DSP synthesis," in *Proceedings of International Symposium on Circuits and Systems*, 1991, pp. 2351–2354.
[35] L.-F. Chao and E. H.-M. Sha, "Scheduling data-flow graphs via retiming and unfolding," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 12, pp. 1259–1267, Dec. 1997.
[36] S. Kurra, N. K. Singh, and P. R. Panda, "The impact of loop unrolling on controller delay in high level synthesis," in *Proceedings of the conference on Design, automation and test in Europe*, 2007, pp. 391–396.
[37] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipeling loops," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994, pp. 63–74.
[38] *The Trimaran Compiler Research Infrastructure*. http://www.trimaran.org/.
[39] *The SUIF Compiler Group. SUIF: An infrastructure for research on parallelizing and optimizing compilers*. http://suif.stanford.edu.
[40] V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," in *Proceedings of the 1994 International Conference on Signal Processing Applications and Technology*, 1994.
[41] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE International Workshop on Workload Characterization*, 2001, pp. 3–14.
[42] E. A. Lee, "Programmable DSPs: A brief overview," *IEEE Micro*, vol. 10, pp. 14–16, October 1990.
[43] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the Symposium on Principles of Programming Languages*, 1996, pp. 32–41.
[44] R. Govindarajan, E. R. Altman, and G. R. Gao, "Minimal register requirements under resource-constrained software pipelining," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994, pp. 85–94.
[45] J. Llosa, M. Valero, E. Ayguadé, and A. González, "Modulo scheduling with reduced register pressure," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 625–638, June 1998.
[46] S. A. Mahlke, "Design and implementation of a portable global code optimizer," *Master thesis*, vol. Dept. of Computer Science, University of Illinois, 1992.

Meng Wang received the B.E. and M.S. degrees in computer science from Xidian University, Xi'an, China, in 2003 and 2006, respectively. He has been a PhD candidate with the Department of Computing, Hong Kong Polytechnic University, since 2006. His research interests include embedded systems, compiler optimization, and real-time systems.

Zili Shao received the B.E. degree in electronic mechanics from the University of Electronic Science and Technology of China, Sichuan, China, in 1995, and the M.S. and Ph.D. degrees from the Department of Computer Science, University of Texas at Dallas, in 2003 and 2005, respectively. He has been an Assistant Professor with the Department of Computing, Hong Kong Polytechnic University, since 2005. His research interests include embedded systems, high-level synthesis, compiler optimization, and hardware/software co-design.

Jingling Xue received the BSc and MSc degrees in Computer Science from Tsinghua University, China in 1984 and 1987, respectively. He received the PhD degree in Computer Science from Edinburgh, United Kingdom, in 1992. He is currently a Professor in the School of Computer Science and Engineering at the University of New South Wales. His current research interests include programming languages, compiler optimizations, program analysis, high-performance computing and embedded systems. He is a senior member of IEEE and a member of ACM.