An Efficient Technique for Eliminating Hidden Redundant Memory Accesses

[1*]Sanni Kumar
Software Developer,
Wintro Tech Pvt. Ltd,
Delhi, India.
Email: sanniangrish@gmail.com

[2]Pradeep Kumar Singh
Department of Computer Science & Engineering,
M.M.M. University of Technology,
Gorakhpur, India.
Email: topksingh@gmail.com

ABSTRACT: Nowadays, Speed of the processor is much higher than the speed of the memory. To reduce the performance gap between memory and processor require effective compiler optimization techniques. As the number of memory accesses operations increase so reduce the performance of various applications. Loops are the most critical section in all applications and consume most time and power. In this paper, we propose an efficient technique; REMOTOR (Redundant memory accesses eliminator) for tracking the redundant memory accesses and eliminate them outside the loops. We provide a preliminary experimental analysis of our algorithms using the Trimaran 4.0 compiler infrastructure and simulator. The results show that our technique significantly reduces the dynamic load operations.

Keywords: Loop scheduling, memory scheduling, virtual register allocation.

INTRODUCTION

Optimization techniques to reduce memory accesses are often needed to improve the timing performance and power consumption for computational intensive applications. Loops are usually the most critical sections and consume a significant amount of time and power in this type of applications. Therefore, it becomes an important problem to reduce the amount of memory accesses of loops for improving performance. Computationally intensive loop kernels of various 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 detect and eliminate hidden redundant memory accesses for loops of DSP applications. As typical embedded systems have limited number of registers, in our technique, we carefully perform instruction scheduling and register allocation to reduce memory accesses under register constraints.

Our strategy for optimizing memory access is to eliminate the redundancy found in memory interaction when scheduling
memory operations. Such redundancies can be found within loop iterations, possibly over multiple paths. During loop pipelining redundancy is exhibited when values loaded from and/or stored to the memory in one iteration are loaded from and/or stored to the memory in future iterations.
Our main contributions are summarized as follows.

- We study and address the memory access optimization problem for computational intensive applications, which is vital both for enhancing performance and reducing redundant memory accesses. Different from the previous work, we propose a data flow graph model to analyze loop-carried data dependencies among memory operations which perform graph construction and reduction within same loop body with improving the complexity.
- We develop a technique called REMOTOR for reducing hidden redundant memory accesses within the loop using register operations under register constraint. This approach is suitable for computational intensive applications such as DSP, which typically consist of simple loop structures.

Various techniques for reducing memory accesses have been proposed in previous work. Two classical compile-time optimizations, redundant load/store elimination and loop-invariant load/store migration [1]–[3], can reduce the amount of memory traffic by expediting the issue of instructions that use the loaded value. Most of the above optimization techniques only consider removing existing explicit redundant load/store operations.

Scheduling framework, *MARLS* (Memory Access Reduction Loop Scheduling)[15], to reduce memory accesses for DSP applications with loops, proposed. Their basic idea is to replace redundant load operations by register operations that transfer reusable register values

of prior memory accesses across multiple iterations, and schedule the register operations. Algorithm MARLS consists of two steps. In the first step, they build up a graph to describe loop-carried dependencies among memory operations, and obtain the register reservation information from the given schedule. In the second step, they use graph coloring algorithm to assign the register to the memory operations. As the no. of register operations perform it increase the demand of more register which cause more register pressure.

An another machine-independent loop memory access optimization technique proposed , redundant load exploration and migration (REALM)[16], to explore hidden redundant load operations and migrate them outside loops based on loop-carried data dependence analysis. In this paper, their work is closely related to *loop unrolling*. This technique can automatically exploit the loop-carried data dependences of memory operations using a graph, and achieve an optimal solution by removing some possible redundant memory accesses based on the graph. Moreover, their technique outperforms loop unrolling since it introduces code size expansion. They first build up a data-flow graph to describe the inter-iteration data dependencies among memory operations. Then they perform code transformation by exploiting these dependencies with registers to hold the values of redundant loads and migrating these loads outside loops. In this technique they used two different algorithms for Data- flow graph analysis, one for graph construction and another for graph reduction which increase the complexity of REALM[16] technique.

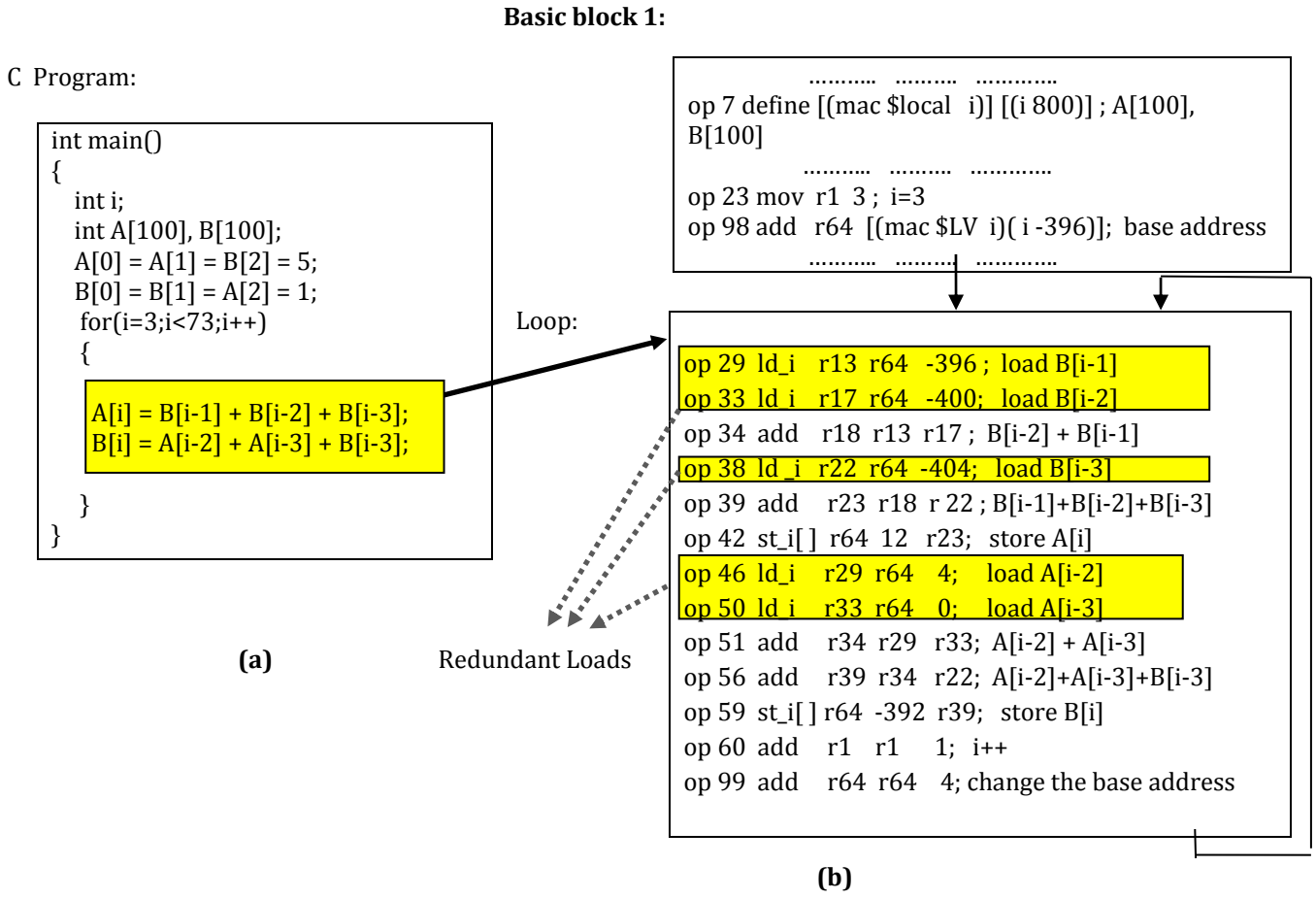


Fig. 1 . Motivational example: (a) A C program: (b) : L-code generate d by IMPACT Compiler

MOTIVATIONAL EXAMPLES

We have taken an example to show our technique. We have used a C program shown in in fig. 1(a). Fig. 2(b) show the Lcode (an intermediate low level code) generated by IMPACT Compiler for this C program. We test it on cycle-accurate VLIW simulator of Trimaran[11].

The operation number 'op' is not in sequence due to various classical optimizations performed by compiler. Two different array A and B are stored in consecutive memory location in fig. 1(b). The base pointer for array reference is set to address of B[0] and assign to register r64 at the end of the basic block 1 *(op98 add[(r64 i)] [(mac $LV i)(i -396*)]). Although classical optimizations remove all redundant operations from t- he intermediate code but hidden redundant load still exist in this. For example, *op38(op 38 ld _i [(r22 i)] [(r64 i) (i -404)]),* is hidden redundant load because it always the load data B[i-3], that is store by *op42(op 42 st_i[ ] [(r64 i) (i -392) (r39 i)]; store B[i]),* B[i], three iteration before, as shown in fig 1(b). As the loop will executed 70 time, there are 7 memory operation and 490 dynamic memory accesses.

REDUNDANT MEMORY ACCESSES ELIMINATOR ALGORITHM

In this section, we proposed the REMOTOR algorithm in Section III-A. This algorithm has three major functions, first function is data-flow graph construction function, which detect and analysis the redundant load/store operation and construct a data-flow graph based on the inter iteration loop-cried data dependencies. In second function, we perform the code replacement; replace the memory operations with register move operations. In this, we perform the register assignments which boost the speed of compilation processes. As the number of register move operation increase, the requirement of more register increase, so in third and final function, we perform register-pressure aware scheduling based on modulo scheduling with slightly modification.

In Section III-B, III-C and III-D, we discuss its all three key functions, and perform complexity analysis in Section III-E.

A. REMOTOR Alogrithm

Require: Intermediate Code generated by compiler after all classical optimizations.
Ensure: Intermediate code with redundant memory accesses across different iterations eliminated.

1. Put all the memory load/store operations into the node set $V = \{ v_1, v_2 \ldots\ldots v_n \}$ for each array, where n is the total number of memory operations.
2. **For** each node set *V* **do**
3. Call function **DFG_Construction(V)** to build up the data-flow graph G = (V, E, d) of node set *V* where *E* is the set of edges between different memory operations and d is delay for any edge (Sec. III-B).
4. Call function **Code_Replacement (V, G)** to replace the memory operations with register move operations based on data flow-graph **G** (Sec. III-C).
5. Call function **RPA_Scheduling (V, G)** to optimize the register assignment and to minimize the register requirement.
6. **end for**

loops. Our basic scheme is to investigate inter-iteration data dependencies among memory operations and replace these memory operations with register operations**.** The registers are used in such a way that we don't need prior memory accesses which are unaffected or needless to be fetched again over multiple loop iterations. Our REMOTOR algorithm is shown in III-A.

Intermediate machine level Lcode generated by IMPACT compiler [2] is the input for this algorithm. This algorithm accomplish in two step. In its first step, it detect the array in the loop and make a set V_X of all memory operations for array X. For our motivational example shown in fig. 1(a), there are two set V_A and V_B for array A and B respectively, shown as follows.

Memory operations set V_A : { op42: stA[i] : r23; op46 : ldA[i-2] : r29 ; op50 :ldA[i-3] : r33}.

Memory operations set V_B : { op29: ldB[i-1]: r13; op33 : ldB[i-2] : r17 ; op38 : ldB[i-3] : r22 ; op59 :stB[i] : r39}.

In second and last step, we carried out optimizations on each memory operations set. In this step, we build up and explain three major functions in section III-B, III-C and III-D.

B. DFG_Construction function

Require: A memory operation s set $V = \{ v_1, v_2 \ldots\ldots v_n \}$.
Ensure: A Reduce data-flow graph $G = (V, E, d)$.

// Get the node set G:

1. Let the memory operation set V be the node set of G.
2. **For** i=1 to N **do**
3. Calculate S = the no. of store operation in V.
4. end for
5. **for** i=1 to N **do**
   // initially set the delay with maximum value or ∞
6. $d_{old} = \infty$;
   //Only access the memory load operation for calculating the delay not store operation
7. **for** j=1 to N – S + 1 **do**
   // Calculate the weight for the each node pair (v_i, v_j):

8. Calculate the weight for the node pair (v_i, v_j) as $\rightarrow$ d($\mathbf{v_i}$ $\mathbf{v_j}$) = distance/base, Where distance = difference between address operand value of $\mathbf{v_i}$ and $\mathbf{v_j}$; and base = value of base pointer;
9. **if** d($\mathbf{v_i} \rightarrow \mathbf{v_j}$) > 0 **then**
10. Add an edge , $v_i \rightarrow v_j$ with the no. of delay d_{old} = min(d($\mathbf{v_i} \rightarrow \mathbf{v_j}$) and $\mathbf{d_{old}}$);
11. **end if ;**
12. **end for**
13. **end for**

Our REMOTOR algorithm eliminates the redundant load in

Delay calculation for dependent memory operations:

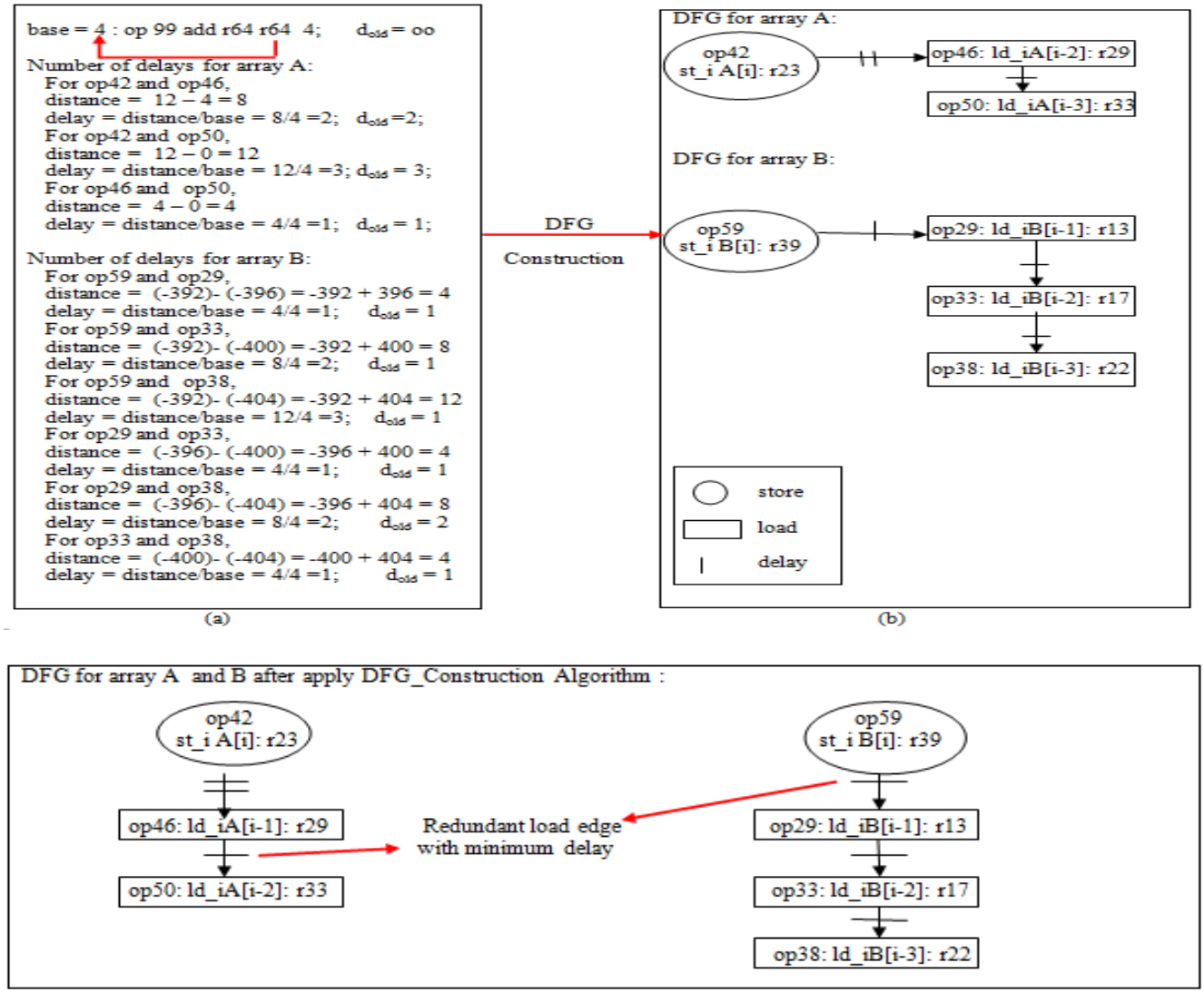


Fig. 2. Data flow-graph construction for array A and B in the motivational example : (a) delay calculation; (b) DFG build using REALM Graph_construction algorithm[18] with complexity *O(MN²)*; (c) DFG build using REMOTOR DFG_Construction algorithm with complexity *O(MN(N-S))*.

DFG_construction algorithm calculates the delay between two nearly dependent memory operations. The input of DFG_construction algorithm is memory operation set of array $V = \{ v_1, v_2 . . . . . v_n \}$, and the output is an edge-weighted directed graph G = (V, E, d), where E is the set of weight directed edges and *d(e)* delay or weight on each edge e ϵ 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 DFG_Construction, we performed two functions simultaneously, first is graph construction, second and last graph reduction.

1) Construction: We first calculate the delay 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, as shown in Fig. 2(a).
- The second part is to acquire the base 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 Fig. 2(a), the base value equals to the third operand of operation in which the base pointer changes.

2) Reduction: we perform both construction and reduction in single algorithm. To reduce graph, for each load, we keep the edge from the closest preceding memory operation, which has the latest produced values. We use two rules shown as follows.

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:

• 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.

In our algorithm, Rule one is applied using an initial variable, d_{old} . d_{old} keep a track of minimum delay. Two applied rule two, we first calculate number of store *S* operation in operation set *V*. Then, using second loop in

14. i = i + 1;
15. **end while**

step 7 of DFG_Construction algorithm we always keep store operation at the top of DFG.

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 Fig. 2(c). According to rule 1, the edge is *(op29 → op38)* deleted as it has more delays than the edge *(op33 op38)* for load *op38*.

C. Code_Replacement function

Require: Optimized data-flow graph G = (V, E d) produced by DFG_Construction(V).

Ensure: Intermediate code with hidden redundant memory operations eliminated.

1. **set** N = number of memory operations in set V;
2. **for** i = 1 to N **do**
3. Associate an coloring variable, Color(v_i), and set Color(v_i) ← Red;
4. **if** ((v_i is load) **&&** (v_i has one incoming edge) **then**
5. **set** Color(v_i) ← yellow;
6. **end if**
7. **end for**
8. **set** i = N
9. **while** (i not equal to 0 && color(v_i) = yallow) **do**
10. let there is an relationship such that "u $\xrightarrow{m}$ v ", where m is delay between parent node u and child node v. Let r_u and r_v the register used for memory access by u and v respectivily. Replace code with following two step:
11. **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 → r_v ;
    - When m > 1 , convert load v to m register operations with the following order: move r_1 r_v , move r_2 → r_1 , ... , move r_u → r_{m-1} in which " r_1, r_2, ...r_{m-1}" are newly generated registers.
12. **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 → r_v, 2nd iteration of v r_1 , mth iteration of v r_{m-1};
13. Color (v_i) ← green;

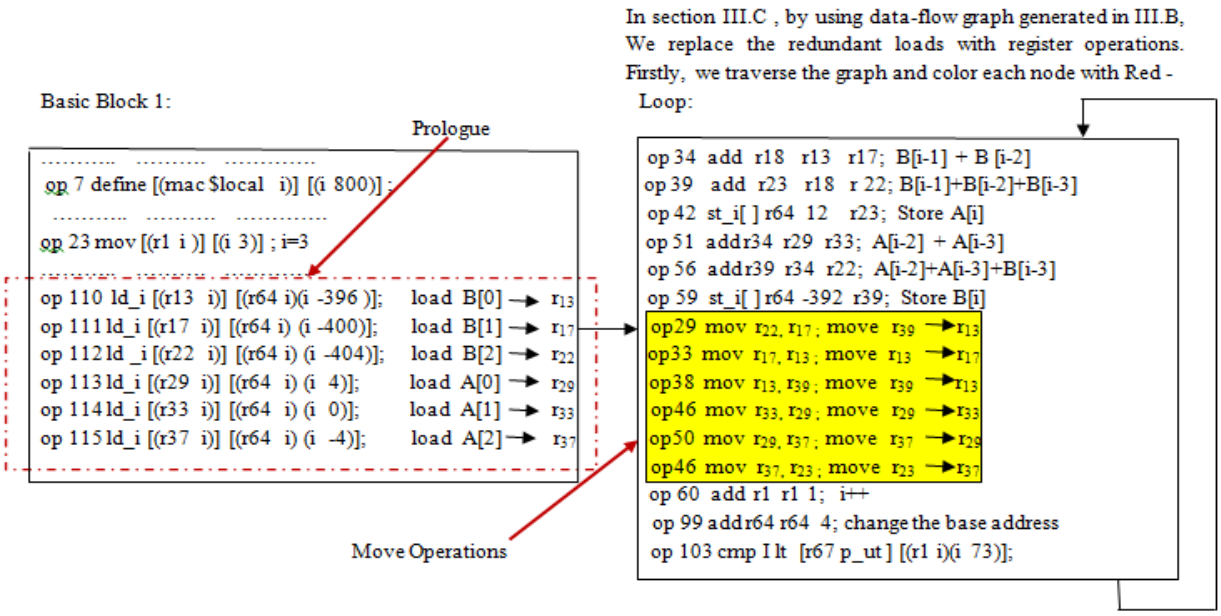


Fig. 3 The optimized intermediate code generated by IMPACT [2] compiler after applying our technique.

Color, to indicate that red node is unprocessed node in DFG. Then we found load node with one incoming edge and color it with yellow color, to indicate that yellow node is redundant load memory operation node. Then, we apply bottom-up approach to replace each redundant load with register operation and color each processed node with green, to indicate that green node replaced by register operation.

Working of our technique is shown in fig. 3. Our basic idea is to replace redundant loads with register move operation. There are six register move operations for five load operations.

There exists a one serious problem that is increase in the register pressure. Increasing in the memory loads operations leads to increasing in number of register move operations. So we require a effective register pressure aware technique to minimize the register pressure.

There are number of method to reduce the register pressure [17]-[20], the most commonly method is modulo scheduling [17].

Our technique can increase the demand of register than available register so here we purposed a modified modulo scheduling with reduce register pressure.

In Section III.D we show our register-pressure aware scheduling algorithm RPA_Scheduling(). In this technique we use data-dependence graph DDG = (N, E, d) where N is the total number of operations such as memory operations, control operations and alu operations. Each edge e = n1 n2with

D. RPA_Scheduling

Require: A register-resource vector R = (r_1, r_2,...., r_m) , where m is the number of register available. A data-dependence graph DDG = (N, E, d) of the input loop, the timing constraint T and Maximum schedule MS.

Insure: The modulo scheduling with reduce register pressure.

1. Find the path in DDG with largest memory load operations, say critical path CP.
2. Let Rx is the number of registers used in CP
3. Rx = 0, MS;
4. **while** (Rx < m and MS > 0) **do**
   // apply modulo scheduling
5. MII() = max (RecII, ResII)
6. II = MII()
7. **while** (the unscheduled operations is not empty &II < TC) **do**
8. MS = MS * N
9. Compute priority for each node in G an put them into the list
10. pick up a node with highest priority and assign it register r from R .
11. calculate timeslot for it;
12. II = II +1; Rx = Rx +1;
13. **end while**

14. MS = MS – 1;
15. **end while**

delay de iterations. R is the register-resource vector with m register for memory operations. The input value for this algorithm beside R and DDG are time constraint T and Maximum Schedule MS. T give the upper bond of schedule length and MS denotes how many schedule attempts we will try to get a legal schedule before giving up the current II.

We first find out the critical path in DDG. A critical path in DDG is a path with largest memory operations. Then we apply our technique along with modified modulo scheduling to replace memory operations with register move operations.

The calculation for Initiation Interval ***(II)*** in step five and six is done as follow.

The initiation interval *II* between two successive iterations

is bounded either by loop-carried dependences in the graph (*RecMII*) or by resource constraints of the architecture (*ResMII*). This lower bound on the *II* is termed the **Minimum**

Initiation Interval (*MII* = max (*RecMII*, *ResMII*)).

This algorithm works in three main steps: computation of *MII*, preordering of the nodes of the DDG using a priority function, and scheduling of the nodes following this order. The ordering function ensures that, when a node is scheduled, the partial scheduling contains at least a predecessor /successor. This function to reduce the lifetime of loop variants and, thus, reduce registers requirements.

III.E Complexity Analysis for REMOTOR

In the REMOTOR 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 REMOTOR algorithm, it takes at most *O (MN)* to obtain the node sets. In function DFG_Construction(), for the node set of an array, it takes at most *O(M.(N-S))* to construct the data flow graph among N nodes where S is the number of store operations. In function Code_Replacement(), we can find the number of children for N-S nodes in *O(N(N-S))* , and it takes at most *O(N-S)* to finish code replacement. Totally, for M arrays, the REMOTOR technique can be finished in *O(M.N.(N-S))*.

In fig. 2(b). the DFG build up using REALM[16] Graph_Construction algorithm[ ], this algorithm use two different algorithm Graph_construction and graph reduction to build up final data flow-graph as shown in fig. 2 (c) with complexity $O(N^2)$, where N is the total number of memory operations. In contrast our DFG_algorithm performed both graph construction and reduction in a single algorithm, DFG_construction() algorithm and takes at most *O(M.(N-S))*. Below us analysis REMOTOR complexity in all three cases:

1. *Best Case:* If S = N-1 means there are only one load operation then Comp(REMOTOR) = *O(M.N).*
2. *Average case:* If S = N/2 means there are equal load and store operation then Comp(REMOTOR) = *O(M.N.N/2)*
3. *Worst Case:* If S = 1 means there are only one store operations then Comp (Remotor) = $O(M.N^2)$.

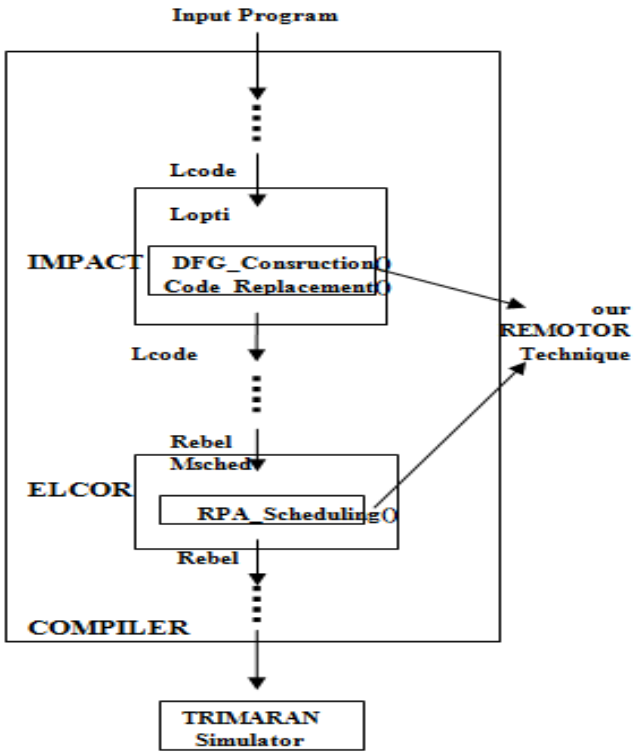


Fig. 4 Implementation and Simulation framework

EXPERIMENS

We have implemented REMOTOR algorithm in two part of Trimaran [12] simulator as shown in fig. 4. Data-flow construction function and code replacement function are implemented on IMPACT compiler of Trimaran. The Last, Register-pressure aware function is implemented on ELCOR compiler of Trimaran.

In this section, we first discuss Trimaran configuration in section IV-A and then introduce our benchmark program

in IV-B. The experimental results and discussion are presented in Section VI-C.

A. Trimaran Configuration

We have conducted the experiments using cyclic accurate VLIW architecture of Trimaran simulator. Configuration for Trimaran is shown in table I.

TABLE: I
Trimaran Configuration

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

The memory system consists of a 32 K four-way associative instruction cache and a 32 K four-way associative data cache; both with 64 byte block size. In the system, there are 32 integer registers and 32 floating point registers.

B. Benchmarks 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 [13] and MiBench [14]. We test both the fixed-point and the floatingpoint versions of benchmarks from DSPstone [13]. The details of benchmarks are shown in Table II.

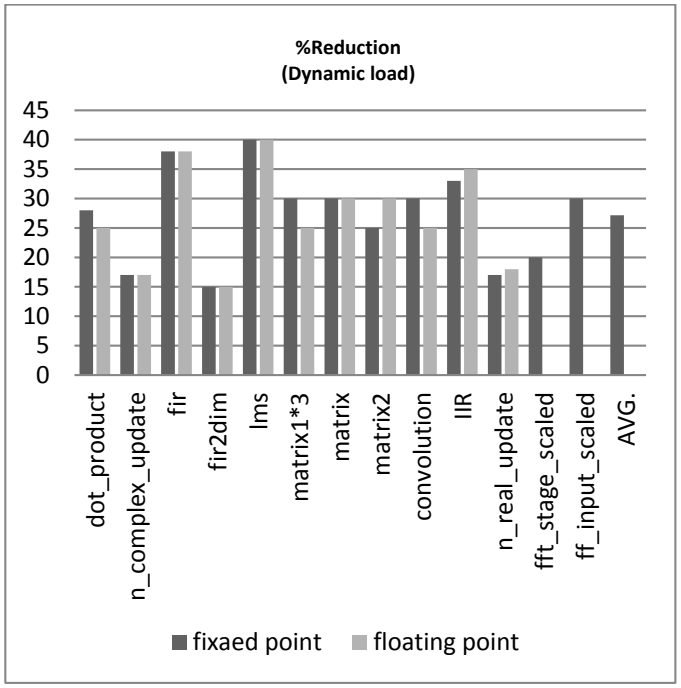
TABLE: II
BENCHMARKS

Benchmarks	Source	Benchmarks Description
Convolution	DSPstone	Convolution
Dot_product	DSPstone	Dot_product
IIR	DSPstone	IIR filter
Fir	DSPstone	finite response filter
Fir2dim	DSPstone	2D finite response filter
Lms	DSPstone	Least mean squre
matrix1*3	DSPstone	matrix1*3
Matrix	DSPstone	Product of two matrix
matrix2	DSPstone	Revised matrix
n_complex_updates	DSPstone	N complex update
n_real_update	DSPstone	N real update
fft_stage_scaled	DSPstone	Integer stage scaling FFT
fft_input_scaled	DSPstone	Integer input scaled FFT
Bfencrypt	Mibench	Blowfish encrypt
Bfdecrupt	Mibench	Blowfish decrypt
Cjpeg	Mibench	JPEG compress
Djpeg	Mibench	PEG decompress
Gsmencode	Mibench	GSM encode
Gsmdecode	Mibench	GSM decode
Rawcaudio	Mibench	ADPCM encode
Rawdaudio	Mibench	ADPCM decode

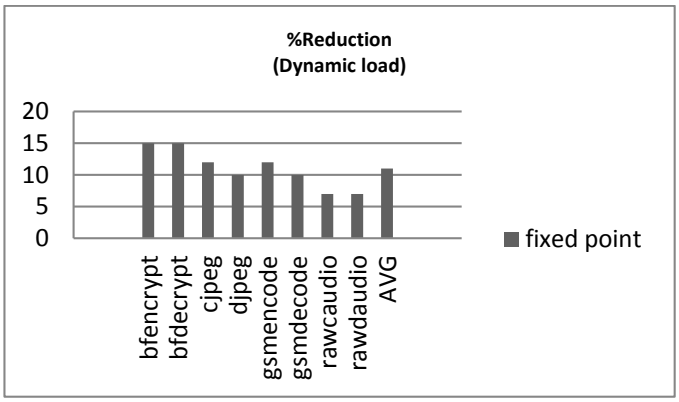
We have generated code for each benchmarks using IMPACT compiler and test it onto Trimaran simulator.

C. Results and Discussions

In this section, we first compare the results our REMOTOR technique with Baseline and after that compare the results of previously proposed REALM technique.



(a)



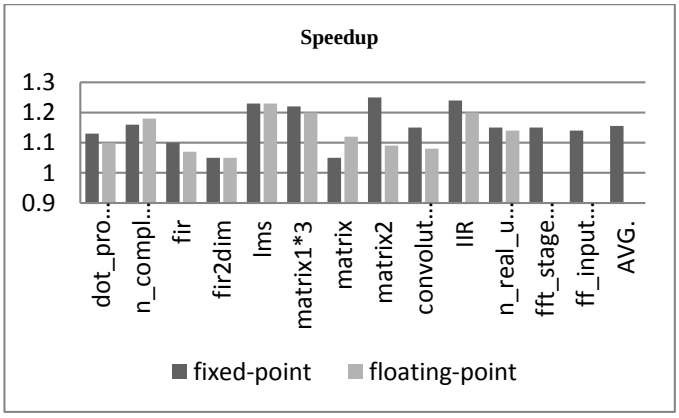
(b)

Fig.5. Reduction in dynamic load operations for benchmarks from: (a) DSPstone; (b) MiBench.

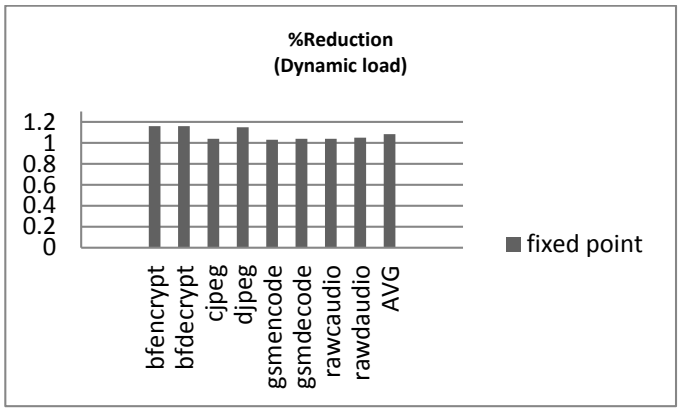
Dynamic Load Reduction. The reduction of load operate-ions in percentages for benchmarks DSPstone[13] and MiBench[14] are shown in fig. 5(a) and fig. 5(b), respectively. The results for the fixed-point and floating-point benchmarks from DSPstone[13] are presented by bars with different colors, and the right-most bar "AVG" is the average result shown in fig. 7(a). For the benchmark MiBench[14] there are only fixed-point benchmarks available so results show using single color bars.

The experimental results show that our REMOTOR algorithm significantly reduces the number of memory access operations using register move operations. On an average, this algorithm achieves 27.15% and 12% reduction for the benchmark from DSPstone and MiBench, respectively when compare with classical optimizations. Next we have compared our technique with previously proposed REALM technique.

Speed up. In fig. 6(a) and 6(b) show that our technique speedup the overall performance by the factor of 1.16 and1.1 for the benchmarks DSPstone and MiBench, respectively.



(a)



(b)

Fig. 6. Performance improvement for the Benchmarks from: (a) DSPstone; (b) MiBench.

For DSPStone (a)

MiBench (b)

Fig. 7. Average performance speedup of REALM[16] and REMOTOR with 16 registers, 32 registers, and 64 registers for the benchmarks from: (a) DSPstone; (b) MiBench.

Comparison between our technique REMOTOR and previously propose method REALM[16] shown in fig 7. The results show that our REMOTOR technique for the benchmarks from DSPstone speedup the performance by the factor of 1.12, 1.19 and 1.25 for 16 registers, 32 registers and 64 register respectively and for the benchmarks from MiBench speedup the performance by the factor of 1.12, 1.19 and 1.25 for 16 registers, 32 registers and 64 register respectively. The results show that our REALM technique achieves larger performance improvements with more register resources.

CONCLUSION AND FUTURE WORK

In this paper, we proposed a machine-independent loop-cried redundant memory accesses optimization technique REMOTOR that is improved version of previously proposed technique, REALM [16]. This technique eliminates the redundant memory operations. Firstly, we analysis the loop-cried data dependencies among the memory operation using data-flow graph with complexity *O(M.N.(N-S))* is much better than the REALM DFG algorithm. We implemented our techniques into IMPAC , ELCOR and Trimaran simulator, and conducted experiments using a set of benchmarks from DSPstone[13] and
MiBench[14] based on the cycle-accurate simulator of Trim-aran. The experimental results showed that our REMOTOR method significantly reduce the number of memory access redundant memory accesses compared with REALM [16] and classical optimizations [1]-[3].

How to unite our techniques, instruction scheduling, and register lifetime analysis together to inefficiently reduce memory accesses under tight register constraints is one of the future work. Second, in embedded systems, energy and thermal are important issues. How to estimate the energy consumption of our techniques, and how to combine our techniques with efficient energy optimization techniques are important problems we require examining in the future.

REFERENCES

1. Aho, M. S. Lam, R. Sethi, and J. Ullman*, Compilers: Principles,Techniques, and Tools*, 2nd ed. Reading, MA: 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-instructionissue processors," in *Proc. 18th Int. Symp. Comput. Arch.*, 1991, pp. 266–275.

3. D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACMComput. 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 Trans. Des. Autom. Electron. Syst. (TODAES)*, vol. 6, no. 2, pp. 149–206, 2001.

5. Y. Ding and Z. Li, "A compiler scheme for reusing intermediate computation results," in *Proc. Ann. IEEE/ACM Int. Symp. Code Generation Opt. (CGO)*, 2004, pp. 279–291.

6. Y. Jun and W. Zhang, "Virtual registers: Reducing register pressure without enlarging the register file," in *Proc. Int. Conf. High Perform. Embed. Arch. Compilers*, 2007, pp. 57–70.

7. D.Kolson, A. Nicolau, and N. Dutt, "Elimination of redundant memory traffic in high-level synthesis," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 15, no. 11, pp. 1354–1363, Nov. 1996.

8. 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 *Proc. IEEE Custom Integr. Circuit Conf.*, Sep. 2005, pp. 239–242.

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

10. J. Seo, T. Kim, and P. R. Panda, "Memory allocation and mapping in high-level synthesis: An integrated approach," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 11, no. 5, pp. 928–938, Oct. 2003.

11. B. R. Rau, "Iterative modulo scheduling: An algorithm for softwarepipeling loops," in *Proc. 27th Ann. Int. Symp. Microarch.*, 1994, pp.63–74.

12. "The Trimaran Compiler Research Infrastructure," [Online].Available: http://www.trimaran.org/

13. V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr, "DSPstone: A DSP-oriented benchmarking

methodology," in *Proc. Int. Conf. Signal Process. Appl. Technol.*, 1994, pp. 715–720.

14. 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 *Proc. IEEE*

15. *Meng Wang, Duo Liu, Yi Wang and Zili Shao, "*Loop Scheduling with Memory Access Reduction under Register Constraints for DSP Application*" in IEEE Trans in 2009.*

16. Meng Wang, Zili and Jingling Xue ,*"*On Reducing Hidden Redundant Memory Accesses for DSP Applications *"*in *IEEE Tra.on VLSI sys, vol. 19, no. 6, June 2011.*

17. N.J. Warter and N. Partamian, "Modulo Scheduling with Multiple Initiation Intervals," *Proc. 28th Int'l Symp. Microarchitecture*, pp. 111-118, Nov. 1995.

18. Josep Llosa, Mateo Valero and Antonio Gonzalez, " Modulo Scheduling with reduce register pressure" in *IEEE TRANSACTIONS ON COMPUTERS, VOL. 47, NO. 6, JUNE 1998.*

19. Rami Beidas, Wai Sum Mong and Jianwen Zhu, "Register Pressure Aware Scheduling for High Level Synthesis" *in IEEE Tran. In 2011.*

20. T. C. Wilson, N. Mukherjee, M. K. Garg, and D. K. Banerji, "An ILPSolution for Optimum Scheduling, Module and Register Allocation, and Operation Binding in Datapath Synthesis". *VLSI Design*, 3(1):21–36.