SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2186

Loop scheduling with memory access reduction subject to register constraints for DSP applications[‡]

Yi Wang[1], Zhiping Jia[2], Renhai Chen[1], Meng Wang[1], Duo Liu[1] and Zili Shao[1,*,†]

[1]*Department of Computing, The Hong Kong Polytechnic University, Kowloon, Hong Kong*
[2]*School of Computer Science and Technology, Shandong University, Jinan, China*

SUMMARY

Memory accesses introduce big-time overhead and power consumption because of the performance gap between processors and main memory. This paper describes and evaluates a technique, loop scheduling with memory access reduction (LSMAR), that replaces hidden redundant load operations with register operations in loop kernels and performs partial scheduling for newly generated register operations subject to register constraints. By exploiting data dependence of memory access operations, the LSMAR technique can effectively reduce the number of memory accesses of loop kernels, thereby improving timing performance. The technique has been implemented into the Trimaran compiler and evaluated using a set of benchmarks from DSPstone and MiBench on the cycle-accurate simulator of the Trimaran infrastructure. The experimental results show that when the LSMAR technique is applied, the number of memory accesses can be reduced by 18.47% on average over the benchmarks when it is not applied. The measurements also indicate that the optimizations only lead to an average 1.41% increase in code size. With such small code size expansion, the technique is more suitable for embedded systems compared with prior work. Copyright © 2013 John Wiley & Sons, Ltd.

Received 14 May 2012; Revised 8 January 2013; Accepted 10 January 2013

KEY WORDS: DSP applications; loop optimization; memory optimization; instruction scheduling

1. INTRODUCTION

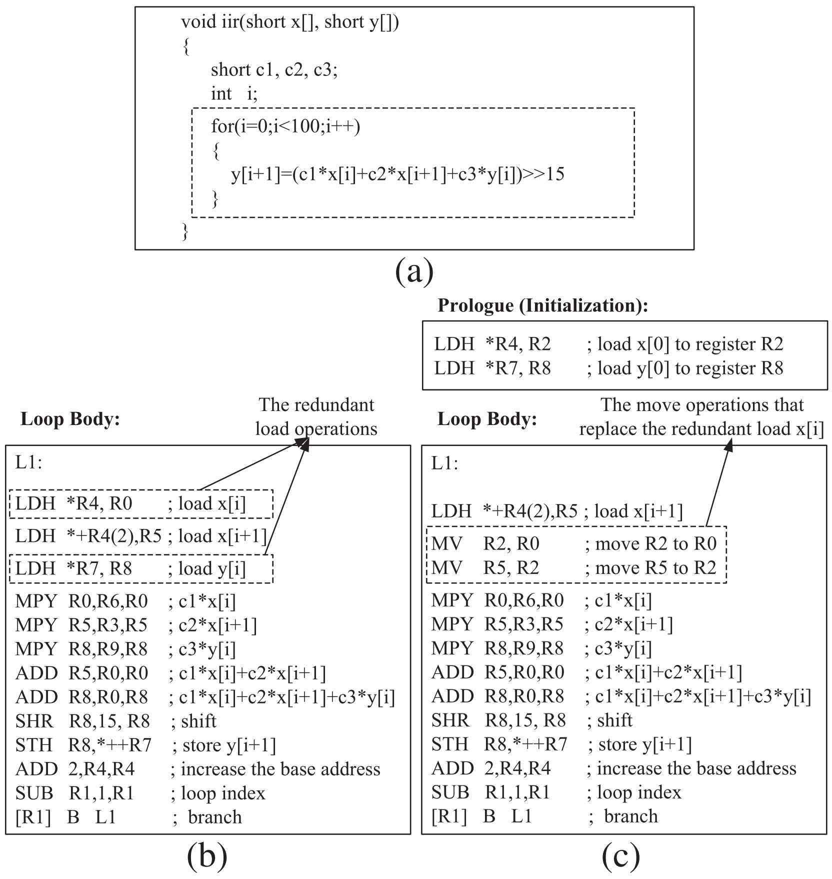
Reducing memory accesses is very important for digital signal processing (DSP) applications in embedded systems as this can effectively improve timing performance and reduce power consumption. Loops are usually the most critical sections and consume a significant amount of time and power in DSP applications. Therefore, it becomes an important problem to reduce memory accesses for loops in DSP applications. In this paper, we focus on developing effective scheduling techniques with memory access reduction on DSP processors as they have been widely used in embedded systems.

Given a DSP application, the infinite impulse response filter from [1], Figure 1(a) shows the C source code of the infinite impulse response filter, and Figure 1(b) shows the corresponding assembly code of its loop kernel. This assembly code is a revised version of the TI C6000 assembly code generated by the TI DSP CCS v4 [2]. For the sake of illustration, the names of the registers in the revised code are revised from the original TI code, so they can be mapped to the very-long-instruction word (VLIW) processor described in Section 3.3 that has 16 physical registers ($R_0, R_1, \ldots, R_{15}$). In the code, it can be observed that three data $x[i]$, $x[i+1]$, and $y[i]$ are loaded

*Correspondence to: Zili Shao, Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong.

†E-mail: cszlshao@comp.polyu.edu.hk

‡A preliminary version of this work appeared in the Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS 2009).



```
void iir(short x[], short y[])
{
    short c1, c2, c3;
    int   i;
    for(i=0;i<100;i++)
    {
       y[i+1]=(c1*x[i]+c2*x[i+1]+c3*y[i])>>15
    }
}
```

(a)

Loop Body:

```
L1:
LDH  *R4, R0      ; load x[i]
LDH  *+R4(2),R5 ; load x[i+1]
LDH  *R7, R8      ; load y[i]
MPY  R0,R6,R0    ; c1*x[i]
MPY  R5,R3,R5    ; c2*x[i+1]
MPY  R8,R9,R8    ; c3*y[i]
ADD  R5,R0,R0    ; c1*x[i]+c2*x[i+1]
ADD  R8,R0,R8    ; c1*x[i]+c2*x[i+1]+c3*y[i]
SHR   R8,15, R8   ; shift
STH   R8,*++R7    ; store y[i+1]
ADD  2,R4,R4       ; increase the base address
SUB   R1,1,R1      ; loop index
[R1]   B   L1         ;  branch
```

(b)

Prologue (Initialization):

```
LDH  *R4, R2      ; load x[0] to register R2
LDH  *R7, R8      ; load y[0] to register R8
```

Loop Body:

```
L1:
LDH  *+R4(2),R5 ; load x[i+1]
MV    R2, R0        ; move R2 to R0
MV    R5, R2        ; move R5 to R2
MPY  R0,R6,R0    ; c1*x[i]
MPY  R5,R3,R5    ; c2*x[i+1]
MPY  R8,R9,R8    ; c3*y[i]
ADD  R5,R0,R0    ; c1*x[i]+c2*x[i+1]
ADD  R8,R0,R8    ; c1*x[i]+c2*x[i+1]+c3*y[i]
SHR   R8,15, R8   ; shift
STH   R8,*++R7    ; store y[i+1]
ADD  2,R4,R4       ; increase the base address
SUB   R1,1,R1      ; loop index
[R1]   B   L1         ;  branch
```

(c)

The redundant load operations
The move operations that replace the redundant load x[i]

Figure 1. (a) The C source code of the infinite impulse response filter [1]. (b) The assembly code of the loop kernel with redundant load operations. (c) The assembly code of the loop kernel by replacing redundant load operations with register operations.

from the main memory by three load operations, and the result $y[i+1]$ is stored after the finish of the corresponding arithmetic operations in each iteration. However, two load operations, *LDH *R4, R0* and *LDH *R7, R8*, used to load $x[i]$ and $y[i]$, respectively, are redundant if two consecutive iterations are examined together. For example, considering iterations 1 and 2, in iteration 1, $x[2]$ is loaded and $y[2]$ is generated, whereas $x[2]$ and $y[2]$ are loaded again in iteration 2. The problem is caused because the aforementioned redundant load operations are not explicit in the loop kernel. These redundant load operations cannot be eliminated or migrated by the classical compiler optimization techniques such as the redundant load/store elimination and loop-invariant load/store migration techniques [3]. In this paper, such redundant load operations are called *hidden redundant load operations*. Figure 1(c) shows the code in which the hidden redundant load operations are replaced with the register operations. The goals of this work are to identify and replace hidden redundant load operations with register operations, to assign physical registers to the newly generated register operations, and to put them into available clock cycles in a schedule (i.e., the free time slots that have not been utilized in the initial task schedule).

This paper presents loop scheduling with memory access reduction (LSMAR) to solve the problem. In our technique, hidden redundant load operations can be effectively identified. We can also guarantee that, if a redundant load operation is replaced with register operations, physical registers can be assigned to the newly generated register operations without register spilling. This is very important as register pressure is increased by the replacement and register spilling may introduce more memory operations than before. LSMAR consists of two phases.

- In the first phase, a *memory access graph* (MAG) is constructed to describe loop-carried dependence among memory operations, and all hidden redundant load operations are identified.

- In the second phase, our technique iteratively replaces each hidden redundant load operation with register operations, assigns a physical register to the newly generated register operations, and schedules them into available clock cycles if possible. In this phase, on the basis of the MAG, each redundant load is first replaced by register operations with virtual register operands. To allocate available physical registers to the operands, a *register-matching graph* (RMG) is built up to describe the time relations among available slacks of physical registers. Then the register allocation problem is solved by finding a *fixed-length simple path* between two specified vertices in the RMG. Finally, on the basis of the results of the register allocation, partial scheduling is performed, so register operations can be put into available clock cycles in a schedule.

We implement our technique LSMAR into the Trimaran compiler [4]. To the best of our knowledge, this is the first work to reduce hidden redundant memory accesses with loop scheduling for DSP applications. We conduct experiments on a set of benchmarks from DSPstone [5] and MiBench [6] on the cycle-accurate VLIW simulator of Trimaran [4]. Experimental results show that our technique achieves significant memory access reduction with little code size expansion compared with the baseline scheme of Trimaran. Experimental results also indicate that our LSMAR technique leads to a negligible increase of code size.

The rest of this paper is organized as follows. Section 2 describes related work. The basic concepts and models are introduced in Section 3. Section 4 presents the LSMAR technique. The experiments are provided in Section 5. Section 6 presents the conclusion and future work.

2. RELATED WORK

Many memory-related issues have been addressed for DSP applications on various memory architectures [7–18]. In [9], Spiral, a program generator and optimizer for linear transforms such as discrete Fourier transform [19], is extended to generate efficient transform code for shared memory platforms including multicore systems. Taking memory constraints into consideration, Ko *et al.* [11] developed a task-level vectorization technique to efficiently process long streams of input data. Zhang *et al.* [14] proposed an architecture to create virtual registers that do not have physical storage locations in the register file, and they proposed a region-based register allocation algorithm to exploit the virtual registers for the short-lived variables. Their experimental results indicated that virtual registers are very effective at reducing register spills. Different from the preceding techniques, in this paper, we focus on eliminating redundant memory accesses for DSP applications.

Various approaches have been investigated to reduce memory accesses [3, 20–31]. Two compile-time optimizations, redundant load/store elimination and loop-invariant load/store migration, can reduce explicit redundant memory accesses [3]. Xue *et al.* [20] presented the first lifetime optimal algorithm that performs speculative partial redundant elimination by combining code motion and control speculation. Li *et al.* [22] introduced the notion of memory access intensity to facilitate quantitative analysis of a program's memory behavior on multicore architectures. The preceding approaches only consider removing explicit redundant memory accesses within one iteration of a loop. Our technique can explore and eliminate hidden redundant load operations across multiple loop iterations on the basis of the loop-carried dependence analysis.

Many data-reuse techniques have been developed for locality optimization and parallelization in both compiler optimization [32–39] and high-level synthesis [40, 41]. Hu *et al.* [34] proposed a compiler-directed cache polymorphism for optimizing data locality of array-based embedded applications. Weinhardt *et al.* [40] developed methods to synthesize shift registers and small on-chip random access memories to reduce the number of memory accesses. Dutt *et al.* [41] proposed a local scheduler transformation that optimizes the accessing of a secondary memory, thereby reducing memory traffic. Our technique is a good supplement of the aforementioned techniques, and it can combine with the said techniques to further improve the performance.

Many loop transformation techniques, such as loop partitioning [42] and array contraction [43–45], have been proposed to reduce memory accesses. In [44, 45], a technique is proposed to facilitate loop fusion by performing array contraction as much as possible to reduce the memory–register traffic of loops. Wang *et al.* [46] proposed an optimal loop scheduling technique for hiding memory latency on the basis of a two-level partitioning and prefetching scheme. Different from the preceding techniques, our technique replaces redundant memory operations with register operations. In addition, our technique can be integrated with the said techniques by providing more opportunities for loop transformations.

With register pressure issues taken into consideration, much research has been performed on register-aware loop scheduling [47–50]. Gao *et al.* [48] proposed a software pipelining technique to improve performance while minimizing register requirements. Chen *et al.* [47] presented a framework for scheduling DSP applications with multidimensional loops subject to register constraints and other resource constraints. In our technique, we integrate register allocation and instruction scheduling to reduce memory accesses under register constraints.

Our work is closely related to the work in [51–54]. In [51], a technique called dynamic memory disambiguation is proposed to improve instruction-level parallelism by disambiguating memory references in loops at execution time. By applying loop unrolling with dynamic memory disambiguation, a loop will be unrolled a few times according to an unrolling factor, and redundant load/store operations can be revealed, disambiguated, and eliminated. With loop unrolling, however, it is hard to determine the optimal unrolling factor, and the code size expansion after unrolling is undesirable for embedded systems. At source code level, a technique called scalar replacement [52–54] can identify repeatedly accessed array elements and replace them with scalar variables. However, these techniques focus on solving the problem at the source code level. This transformation may cause register spilling, which is not considered in the previous work.

Wang *et al.* proposed a theoretical framework called redundant load exploration and migration (REALM) to explore hidden redundant load operations and migrate them outside loops on the basis of loop-carried data dependence analysis [55]. However, register allocation and instruction scheduling are not considered in REALM, which are the two most important phases for the problem of loop scheduling with memory access reduction. In this paper, we focus on solving this critical problem. We propose efficient register allocation and partial instruction scheduling algorithms, and we implement them on the basis of the Trimaran compiler.

3. MODELS AND PROBLEM STATEMENT

In this section, we present models and define the problem. The target VLIW architecture is introduced in Section 3.1. Loops and graph model are defined in Section 3.2. The static schedule and register usage map model are introduced in Section 3.3. Finally, in Section 3.4, we formulate the problem.

3.1. *The target very-long-instruction-word architecture*

As shown in Figure 2, we use a VLIW architecture similar to TI C6000 in this paper. This architecture has several functional units (FUs) that can process multiple operations within the same clock

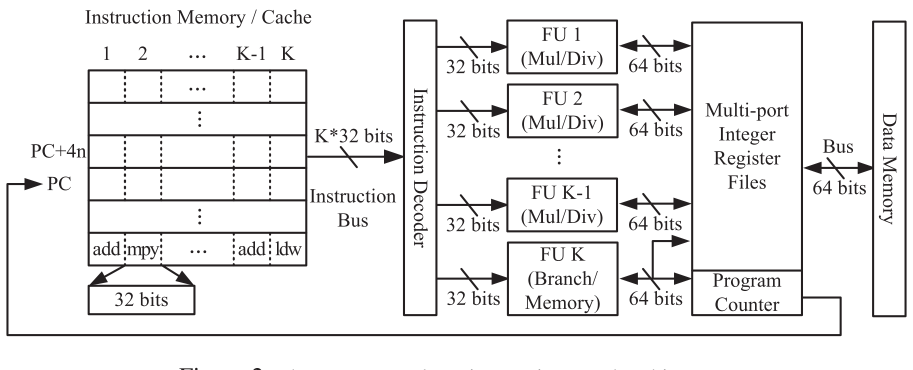


Figure 2. The target very-long-instruction word architecture.

cycle. All FUs are fully pipelined. In this VLIW architecture, a long instruction word consists of K instructions, and each instruction is 32 bits long. In each clock cycle, a long instruction word is fetched to the instruction decoder through a $32 \times K$-bit instruction bus and correspondingly executed by K FUs. There are different types of FUs, including the integer arithmetic logic unit (ALU), floating-point ALU, memory unit, and branch unit. In the experiments, this VLIW architecture is applied by extending the VLIW simulator of Trimaran [4].

3.2. Loops and directed acyclic graphs

In this paper, loops are modeled as a *directed acyclic graph* (DAG).

Definition 3.1
A DAG $G = \langle V, E, t \rangle$ is a node-weighted directed graph, in which V is a set of nodes and each node denotes an operation in the loop; $E = \{(a,b) | a, b \in V\}$ is the edge set, where (a,b) denotes that there is one edge from a to b and b is dependent on a; $t(a)$ represents the latency of node a for each node $a \in V$.

An example is given in Figure 3. In Figure 3(a), each operation in the loop body in Figure 1(b) is assigned a node name, and Figure 3(b) shows the corresponding DAG that models the loop body in Figure 1(b). In the DAG, each edge represents the data dependence between two nodes. For example, node D is a multiplication operation, and it requires the data loaded by node A. So there is one edge from A to D in the DAG.

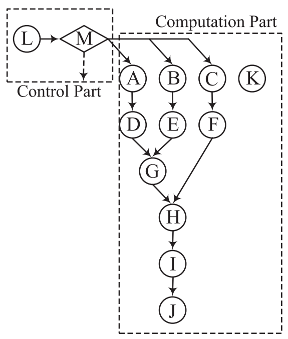
3.3. Static schedules and register usage maps

A static schedule is a repeated pattern of an execution of a loop on a given VLIW architecture. In our work, a schedule implies both control step assignment and allocation. A static schedule can be obtained from the DAG of a loop, and all dependence relations of a DAG must be obeyed. Figure 3(c) shows an example.

Given a VLIW processor with seven FUs (FU_1, FU_2, ..., FU_7) and 16 physical registers ($R_1, R_2, \ldots, R_{15}$) based on the target VLIW architecture in Section 3.1, let FUs FU_1 and FU_2 be integer ALUs, FU_3 and FU_4 be floating-point ALUs, FU_5 and FU_6 be memory units, and FU_7 be a branch unit. We assume that the latency of a memory load operation or a multiplication operation is two cycles, and the latency of other instructions is one cycle. For the DAG in Figure 3(b), the static schedule generated by list scheduling is shown in Figure 3(c). As all FUs are fully pipelined in our VLIW architecture, nodes can be released to an FU in each cycle. If there is one edge between two nodes, the child node needs to wait for a number of cycles, which are equal to the latency of its

Node	Instruction
A	LDH *R4, R0
B	LDH *+R4(2),R5
C	LDH *R7, R8
D	MPY R0,R6,R0
E	MPY R5,R3,R5
F	MPY R8,R9,R8
G	ADD R5,R0,R0
H	ADD R8,R0,R8
I	SHR R8,15, R8
J	STH R8,*++R7
K	ADD 2,R4,R4
L	SUB R1,1,R1
M	[R1] B L1

(a)



(b)

Time	IALU		FALU		Memory Units		BRANCH
	FU1	FU2	FU3	FU4	FU5	FU6	FU7
0					A	B	
1					C		
2	D	E					
3	F						
4	G						
5	H						
6	I	L					
7	K				J		M

(c)

Figure 3. (a) The operations in the loop body in Figure 1(b) and their node names. (b) The directed acyclic graph that models the loop body in Figure 1(b). (c) The schedule generated by list scheduling.

parent node after its parent node has been released. For example, in the DAG, D is dependent on A and the latency of A is two cycles, so D is scheduled at 2.

To record register reservation in a schedule, we associated a life segment set with each physical register, and it contains a set of life segments related to a register that can be obtained by liveness analysis [56]. In this paper, a life segment is represented by $< node, start, end >$ where *node* is the node that generates data, *start* is the starting (release) time of the data, and *end* is the ending (finishing) time of the data. For example, from the loop body and schedule in Figure 3(a, c), the life segment set of register R_0 can be obtained as $\{< A, 0, 3 >, < D, 3, 4 >, < G, 4, 5 >\}$. Putting the life segment set of each register together, we can generate a register usage map.

The register usage map of the schedule in Figure 3(c) is shown in Figure 4. In Figure 4, each column in the map represents a physical register, and each row represents one schedule step. Each bold line in the map represents the life segment of a value that is kept in a physical register. The character beside the beginning of each line denotes the computation node that generates the value. The life segment of a value starts when the value is generated and terminates when the value is consumed by the last computation node that requires this value. Note that the overlapping life segments of the

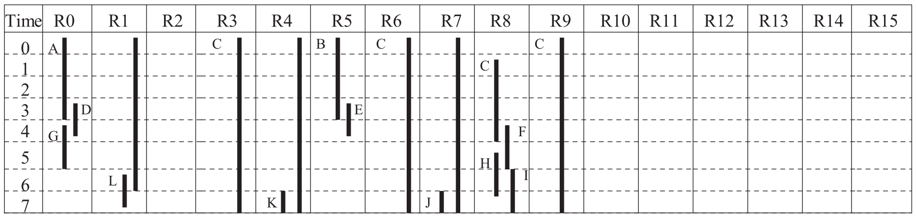
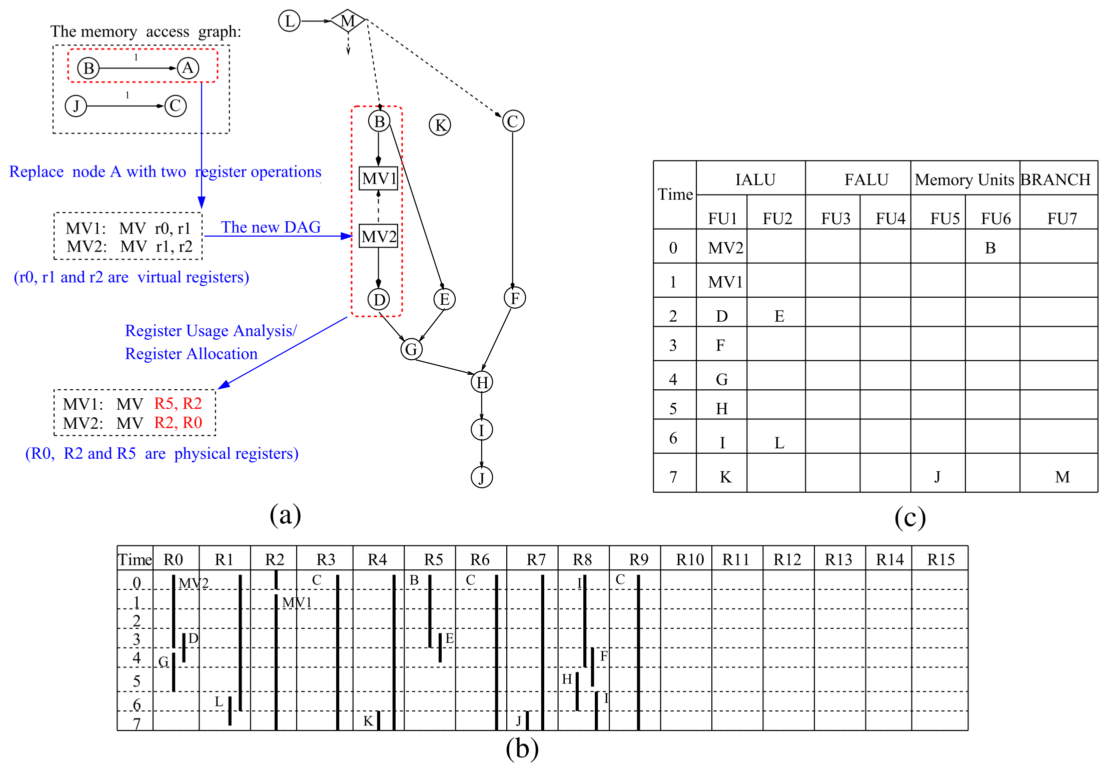


Figure 4. The register usage map of the schedule in Figure 3(c).



Time	IALU		FALU		Memory Units		BRANCH
	FU1	FU2	FU3	FU4	FU5	FU6	FU7
0	MV2					B	
1	MV1						
2	D	E					
3	F						
4	G						
5	H						
6	I	L					
7	K				J		M

Figure 5. (a) The brief process to replace the hidden redundant load operation (node A) with register operations ($MV1$ and $MV2$) and perform register usage analysis/register allocation. (b) The corresponding register usage map after register allocation. (c) The final schedule.

same register are legal as a register can be used as a source operand and a destination operand at the same schedule step in our VLIW architecture.

3.4. Problem statement

The problem to be solved in this paper is defined as follows:

Given a loop, the DAG of the loop, a VLIW processor with N_{reg} registers, and an initial schedule with the corresponding register usage map, the objective is to find a new loop and a schedule such that hidden redundant load operations can be replaced with register operations, and newly generated register operations can be assigned with physical registers and then scheduled into available cycles.

To solve this problem, we not only identify and replace hidden redundant load operations with register operations but also perform register allocation and partial instruction scheduling for the newly generated register operations. An example is given in Figure 5. Given a loop in Figure 1(b), its DAG and schedule in Figure 3, and the corresponding register usage map in Figure 4, Figure 5(a) shows the process that does the following: (i) identify the redundant load operation from the MAG (discussed in Section 4.2); (ii) generate the register operations with virtual registers ($MV1$ and $MV2$) that replace the redundant load operation (node A) (discussed in Section 4.3.1); and (iii) perform register allocation for register operations (discussed in Section 4.3.2). The final register usage map and final schedule are shown in Figure 5(b and c), respectively.

4. LOOP SCHEDULING WITH MEMORY ACCESS REDUCTION

In this section, we present the LSMAR algorithm. First, an overview of the LSMAR algorithm is given in Section 4.1. Then how to identify hidden redundant load operations is described in Section 4.2. Section 4.3 shows how to replace redundant load operations with register operations and how to perform register allocation and partial scheduling. Finally, we analyze the properties and complexity of the LSMAR algorithm.

4.1. The loop scheduling with memory access reduction algorithm

Algorithm 4.1 Loop scheduling with memory access reduction

```
Require: The DAG $G$ of the loop, an initial schedule $S$, the initial register usage map, and the configurations
    of a target processor.
Ensure: A new schedule $S$ in which redundant load operations are reduced.
    // Phase 1. Identify hidden redundant load operations (Section 4.2).
 1: Build up the MAG (Algorithm 4.2) to describe the loop-carried dependence among memory operations;
 2: Put all hidden redundant load operations into set $R$, and associate a cost with each load in the set;
    // Phase 2. Register allocation and partial instruction scheduling (Section 4.3).
 3: while set $R$ is not empty do
 4:     Let $v$ be the redundant load operation with the minimum cost; generate a register move operation
        chain with virtual register operands to replace $v$ based on the MAG;
 5:     Call function SCH_REG_ALLOC() (Algorithm 4.3) to perform register allocation and partial
        instruction scheduling;
 6:     Remove the operation $v$ from set $R$.
 7: end while
```

The LSMAR algorithm is designed to identify and replace hidden redundant load operations with register operations, assign physical registers for the register operations, and schedule them subject to register constraints. An overview of our LSMAR algorithm is shown in Algorithm 4.1. The inputs of LSMAR include the DAG of the loop, an initial schedule, the corresponding register usage map, and the configurations of a target processor. It mainly consists of two phases.

- The *first phase* is to build up the MAG and identify all hidden redundant load operations.
- The *second phase* is to iteratively perform register allocation and instruction scheduling for newly generated register operations used to replace a redundant load operation.

The details of the two phases are presented in Sections 4.2 and 4.3, respectively.

4.2. *The identification of hidden redundant load operations*

In this paper, we build up a graph, called MAG, to describe data dependence relations among memory operations. The MAG is defined as follows.

Definition 4.1
An MAG, $MAG = \langle MV, ME, w \rangle$, is an edge-weighted directed graph, where $\{MV = mv_1, mv_2, \ldots, mv_N\}$ is the node set including all memory operations; $ME = \{(mv_i, mv_j) | m_i, m_j \in MV\}$ is the edge set, and (mv_i, mv_j) represents the dependence from mv_i to mv_j; $w(mv_i, mv_j)$ is a function to represent the edge weight of (mv_i, mv_j) for an edge $(mv_i, mv_j) \in ME$.

Algorithm 4.2 Build up MAG.

```
Require: DAG $G = \langle V, E, t \rangle$.
Ensure: The $MAG = \langle MV, ME, w \rangle$.
    // MAG construction:
 1: Let the memory operations of $G$ be the nodes of the node set $MV$.
 2: for each node pair $(mv_i, mv_j)$ that operates on the same array do
 3:     Calculate the edge weight, $w(mv_i, mv_j) = distance/step$, where $step$ is the increment value of the
        base pointer of the array in each iteration and $distance$ is obtained by subtracting the address operand
        value of $mv_i$ from the address operand value of $mv_j$.
 4:     Add the edge $(mv_i, mv_j)$ into $ME$ if the edge weight $w(mv_i, mv_j)$ is greater than 0. $mv_j$ is a load
        operation and $mv_i$ is either a store operation or a load operation.
 5: end for
    // Memory access graph simplification:
 6: for each node $mv_i$ in $MV$ do
 7:     if $mv_i$ is a load operation and has more than one incoming edge then
 8:         Keep the edge with the minimum weight and delete all other edges.
 9:         if $mv_i$ has more than one incoming edge then
10:             Keep the edge whose source node is a store operation and delete all other edges.
11:         end if
12:     end if
13: end for
```

Algorithm 4.2 shows how to construct a MAG. In Algorithm 4.2, all memory operations of the DAG G are firstly put into the node set MV of the $MAG\langle MV, ME, w \rangle$. Then, for each pair of nodes (mv_i, mv_j), mv_i, $mv_j \in MV$ that operates on the same array, the weight $w(mv_i, mv_j)$ is calculated by $distance/step$. If the weight $w(mv_i, mv_j)$ is greater than 0 and mv_i is a load operation, we add an edge $(mv_i, mv_j) \in ME$ in the MAG and let the weight $w(mv_i, mv_j)$ be the edge weight of the edge $(mv_i, mv_j) \in ME$. The edge weight with positive value denotes that node mv_j operates on the memory location where node mv_i has been operated $w(mv_i, mv_j)$ iterations before. Thus, node mv_j can be replaced by exploiting the register value of node mv_i. Note that we focus on eliminating redundant load operations. We add one edge into the edge set when the destination node is a load operation and the source node is either a store operation or a load operation. Therefore, in the MAG, we can observe the following: (i) the store nodes can only have outgoing edge(s); and (ii) there are no edges between two store nodes.

After the MAG is constructed, to determine definite replacement patterns, the graph is simplified by deleting redundant edges for every load operation with more than one incoming edge. The rules used here are as follows: (i) for each load node that has more than one incoming edge, it keeps edges with the minimum weight; and (ii) after the first rule is applied, if a load still has more than one incoming edge, it keeps the incoming edge with store operation as its source. The purpose of designing the first rule is that we can use the least number of registers to replace redundant load operations by keeping the edges with the minimum weight. The second rule determines that a store node updates the data of the specific memory location, and it must use the updated value to replace the redundant load. With the preceding rules applied, the MAG has the following properties.

Property 4.1
For each load node in the MAG, there is only one incoming edge.

Proof
After the MAG is reduced according to the first rule, only the incoming edges with the same weight remain. Along these edges, there are at most two edges in which one source node is a load node and another is a store node. If there is more than one load (or store) node, these load (or store) nodes must load (or store) data from (to) the same memory location. Therefore, these explicit redundant loads (stores) should have been eliminated by the previous work in [3]. Then, with the second rule applied, only the incoming edge with store nodes as its source node will be kept. So the property is proven. □

Property 4.2
The $MAG = \langle MV, ME, w \rangle$ can identify all hidden redundant load operations.

Proof
Algorithm 4.2 constructs the MAG. In this $MAG = \langle MV, ME, w \rangle$, store nodes only have outgoing nodes, and there are no edges between two store nodes. For each pair of nodes mv_i and mv_j that has a data dependence relationship, an edge will be added into the MAG only if the following three conditions are satisfied: (1) mv_j is a load operation; (2) mv_i is either a store operation or a load operation; and (3) the weight $w(mv_i, mv_j)$ is greater than 0. Because Algorithm 4.2 iteratively checks each pair of nodes, they have covered all load operations. Therefore, redundant load operations can be identified by checking the number of incoming edges of each node in the MAG. □

Figure 6 shows an illustrative example about how to obtain the MAG for the DAG in Figure 3(b). The value of *step* of each array is calculated using the increment value of the base pointer of the array in each iteration. For example, for array x, its base pointer is changed by node K, and the step is equal to 2. After the edge weight calculation, we add these edges and obtain the graph.

From the MAG obtained by Algorithm 4.2, all hidden redundant load operations can be identified. In the MAG, if a node has an incoming edge, it is a redundant load operation. For simplicity, each redundant load operation is put into a set R and associates with a cost that is the edge weight of its incoming edge in the MAG. For example, for the memory graph shown in Figure 6, set R has two nodes, A and C, and the costs of both nodes are 1.

4.3. Register allocation and partial instruction scheduling

This section presents the second phase of our proposed LSMAR algorithm. In this phase, from the MAG constructed in the first phase, we iteratively eliminate each hidden redundant load operation if possible. To process each redundant load operation, the following two stages are applied.

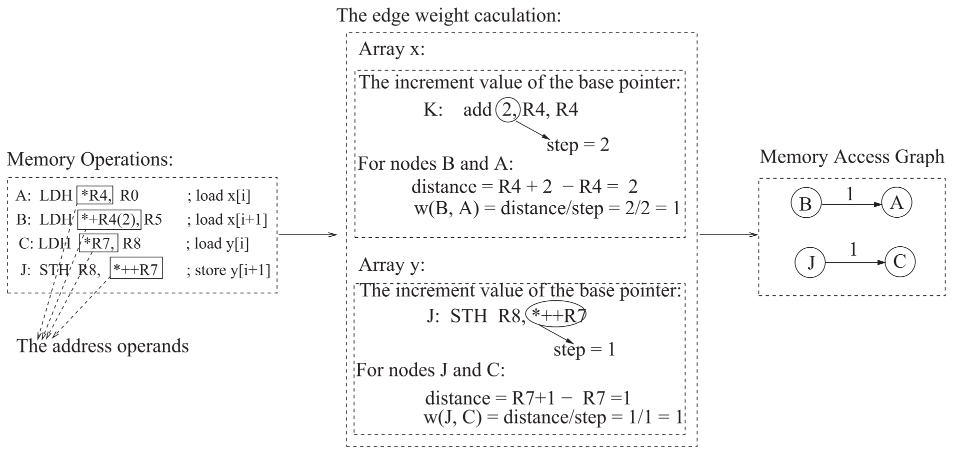

The edge weight caculation:
Array x:
The increment value of the base pointer:
K: add 2, R4, R4
step = 2
For nodes B and A:
distance = R4 + 2 − R4 = 2
w(B, A) = distance/step = 2/2 = 1
Array y:
The increment value of the base pointer:
J: STH R8, *++R7
step = 1
For nodes J and C:
distance = R7+1 − R7 =1
w(J, C) = distance/step = 1/1 = 1
Memory Operations:
A: LDH *R4, R0 ; load x[i]
B: LDH *+R4(2), R5 ; load x[i+1]
C: LDH *R7, R8 ; load y[i]
J: STH R8, *++R7 ; store y[i+1]
The address operands
Memory Access Graph
B
A
J
C
1
1

Figure 6. Construct the memory access graph for the given directed acyclic graph in Figure 3(b).

$$\boxed{\begin{array}{rl} MV_{k+1}: & r_k \rightarrow r_{k+1}; \\ MV_k: & r_{k-1} \rightarrow r_k; \\ \dots\dots\dots & \dots\dots; \\ MV_2: & r_1 \rightarrow r_2; \\ MV_1: & r_0 \rightarrow r_1; \end{array}}$$

Figure 7. The register operation chain.

- In the first stage, *a register operation chain* is generated to replace a redundant load operation on the basis of the MAG. The operands of this chain are virtual registers without physical register allocation.
- In the second stage, algorithm *SCH_REG_ALLOC*() (Algorithm 4.3) is invoked to perform register allocation and partial instruction scheduling for the register operation chain subject to register constraints.

In the following, the details of these two stages are presented in Sections 4.3.1 and 4.3.2, respectively.

4.3.1. Generating a register operation chain. In this section, we show how to replace a redundant load with register operations. Let v be the target redundant load operation that is the node with the minimum cost in R obtained from phase 1. Actually, the cost is equal to the number of registers required to replace the redundant load, so the minimum number of registers is needed to replace this redundant load operation. Let (u, v) be the edge from u to v, and its edge weight is k in the MAG. Moreover, we assume that the parent node u uses the physical register phy_u as its data operand to load/store data and node v uses physical register phy_v to hold the value of the loaded data.

According to the MAG model, the edge (u, v) denotes that nodes u and v access the same memory location across k iterations. In other words, the register value of phy_u in the ith iteration is the same as the value of phy_v in the $(i + k)$th iteration. Thus, the redundant load v can be replaced by a set of register operations if they can directly transfer the value from phy_u to phy_v across k iterations. The pattern for generating a register operation chain is shown in Figure 7.

The generated chain consists of $(k + 1)$ register operations, $MV_1, \dots, MV_{k+1}$, and $(k + 2)$ virtual register operands, $r_0, \dots, r_{k+1}$. Each register operation is used to transfer the reusable value of node u to the next iteration, and each operand (except r_0 and r_{k+1}) is used to keep the desired value for one iteration.

Our LSMAR algorithm uses MV_1 to get the reusable value of node u in each loop iteration and uses MV_{k+1} to define the register value consumed by v's children k iterations later. Thus, the physical register phy_u is allocated to the operand r_0, and phy_v is allocated to r_{k+1}. Using this chain, we can transfer the value that is loaded or stored by node u across k iterations. In such a way, node v can be eliminated as the data required by its children (kept in phy_v) is defined by the move operation MV_{k+1}.

After the register operation chain related to u and v is generated, Algorithm $SCH\_REG\_ALLOC()$ is called to perform register allocation and scheduling.

4.3.2. Algorithm SCH_REG_ALLOC(). Algorithm *SCH_REG_ALLOC*() is designed to schedule the register operation chain, and it solves two interrelated problems. One is how to allocate physical registers to the k intermediate virtual register operands, $r_1, \dots, r_k$. The other one is how to schedule the $(k + 1)$ register operations. Algorithm *SCH_REG_ALLOC*() is shown in Algorithm 4.3, and its four steps are presented as follows.

Step 1: register usage analysis

We first analyze the data dependence of the register operations and then give the schedule boundaries of the register operation chain. Finally, the register requirements of the chain for register allocation are discussed.

```
Algorithm 4.3 Algorithm SCH_REG_ALLOC().
Require: $G$, $S$, the register usage map $r_{map}$, the $MAG$, nodes $u$ and $v$, and the register operation chain $MV_1, \ldots, MV_{k+1}$.
Ensure: The changed $G$, $S$, and $r_{map}$.
1: Perform register usage analysis for the register operation chain, and obtain the register requirements for its virtual register operands.
2: Build up the RMG to find candidate physical registers that fulfill the register requirements of the chain.
3: Find all simple paths of length $k+1$ between the source and sink nodes in the RMG using a path-finding method.
4: Repeatedly performing allocation and scheduling on the register operations on the basis of the paths found by the preceding step until a legal schedule is found or every path has been checked.
```

By transferring a value with the register operation chain, we introduce some new dependence. As shown in the MAG in Figure 6(a), there is an edge from B to A, which means A is redundant and can be generated by B and the register operation chain. So the original DAG in Figure 3 is changed to the new DAG in Figure 5. In general, given nodes u and v, the new dependence introduced by the register operation chain is illustrated in Figure 8.

In Figure 8, the dashed edges are added into the original DAG. The number over each edge denotes the number of iterations of the data dependence, and the symbol below each edge represents the register (either physical or virtual) that holds the value generated by the source node and consumed by the destination node. As shown in Figure 8(c), there is an inter-iteration data dependence between every two consecutive register operations. The sum of the edge weights associated with the edges of the chain equals to k. By exploiting this chain, we can transfer the desired value from the physical register phy_u to phy_v across k iterations.

When we schedule register operations, inter-iteration data dependence is modeled as intra-iteration read–write data dependence as shown in Figure 8(d). In Figure 8(d), a read–write dependence is represented by a dash edge. For example, '$MV_1 \overset{0}{\underset{r_1}{\dashleftarrow}} MV_2$' means there is a read–write dependence between MV_2 and MV_1; therefore, operation MV_2 must be scheduled no later than MV_1. Following the read–write dependence, the chain should be scheduled with the order of

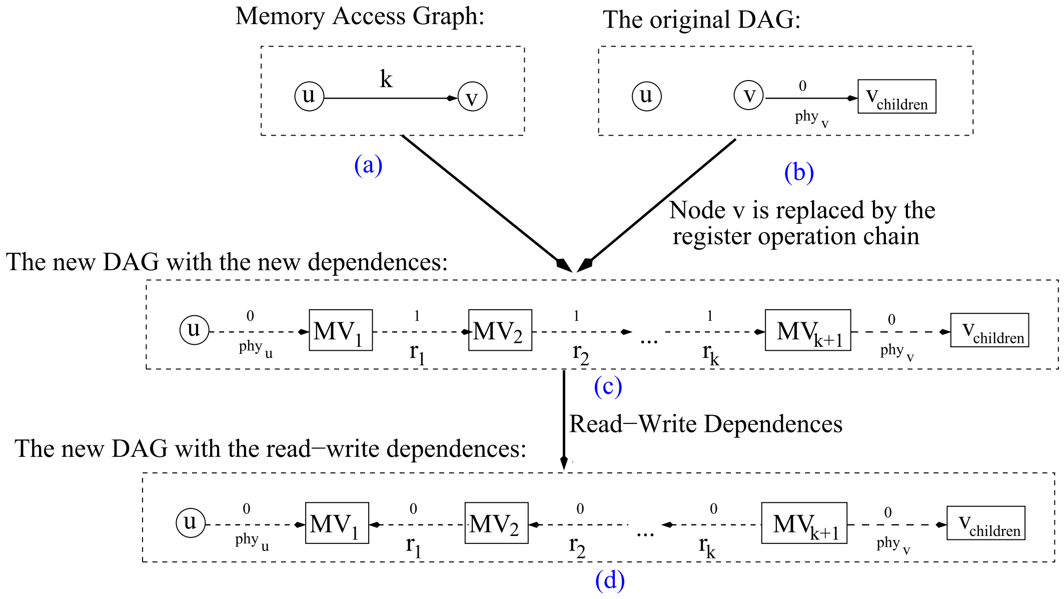


Figure 8. (a) The edge $u \to v$ in the memory access graph. (b) u and v in the original DAG. (c) The new dependence generated by replacing v with the register operation chain. (d) The new dependence by adding the read–write data dependence.

'$MV_{k+1}, MV_k, \ldots, MV_1$'. In addition, the schedule range of the chain is bounded by the schedule times of the first operation, MV_1, and the last operation, MV_{k+1}. In the DAG, there is an edge from u to MV_1 whose edge weight is 1, which means that MV_1 consumes the data generated by node u in an iteration. Thus, we schedule MV_1 into the live range of the value generated by u. Similarly, MV_{k+1} has to be scheduled into the live range of the value generated by node v. The schedule boundaries of the chain are illustrated in Figure 9.

From Figure 9, it can be observed that for each register operation, MV_i, of the chain, its two operands r_{i+1} and r_i must have an intersection of their lifetime. Also, each virtual register operand r_i is used to keep the desired value for one iteration, which means that the lifetime of r_i $(i \neq 0, k+1)$ must span two iterations. With the preceding analysis, we obtain the lifetime requirements of the virtual register operands, $r_1, \ldots, r_k$, as shown in Figure 10.

In Figure 10, the beginning of the lifetime for r_i is represented as '$r_i.st$', and the end of the lifetime for it is represented as '$r_i.et$'. The *first set of requirements* is obtained on the basis of the fact that a register operation is only legal when there exists an overlap between the lifetime of its source and destination operands. In other words, an overlap is required between the end time of the virtual register r_{i-1} and the start time of r_i. The *second set of requirements* shows that the lifetime of each virtual register operand is required to span two iterations.

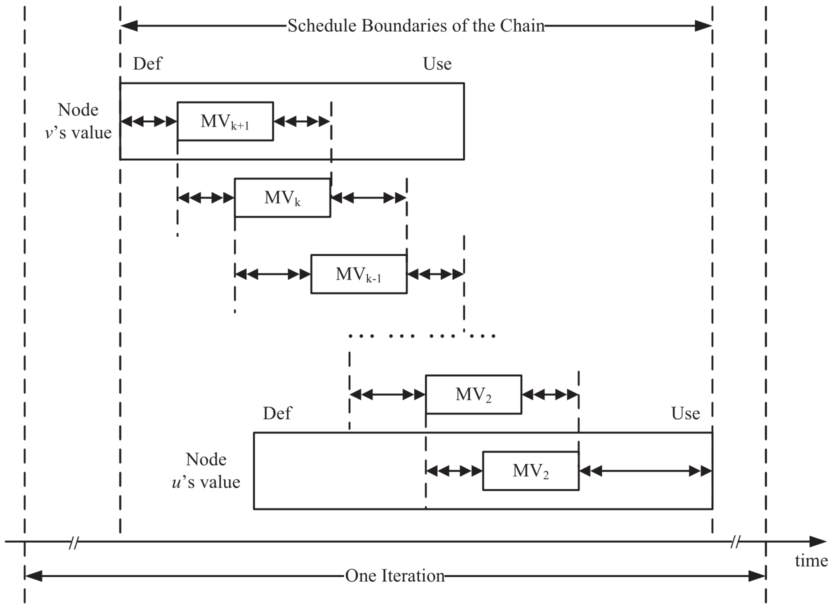


Figure 9. The schedule boundaries of the $(k+1)$ register operations in the register operation chain.

$$\left.\begin{array}{rcl} phy_u.et & \geqslant & r_1.st \\ r_1.et & \geqslant & r_2.st \\ \ldots\ldots & \ldots\ldots & \ldots\ldots \\ r_{k-1}.et & \geqslant & r_k.st \\ r_k.et & \geqslant & phy_v.st \end{array}\right\} \textbf{Requirement 1}$$

$$\left.\begin{array}{rcl} r_1.st & \geqslant & r_1.et \\ r_2.st & \geqslant & r_2.et \\ \ldots\ldots & \ldots\ldots & \ldots\ldots \\ r_{k-1}.st & \geqslant & r_{k-1}.et \\ r_k.st & \geqslant & r_k.et \end{array}\right\} \textbf{Requirement 2}$$

Figure 10. The lifetime requirements for the virtual register operands of the chain ($\boldsymbol{r}_0$ has been assigned to phy_u, and $\boldsymbol{r}_{k+1}$ has been allocated to phy_v).

Algorithm 4.4 Constructing an RMG.

```
Require: The register usage map, the redundant load $v$, and its predecessor $u$ of the MAG.
Ensure: An RMG, $RMG = \langle RV, RE \rangle$.
    // Select physical registers to be the nodes:
 1: Let $S$ be the lifetime of the value generated by node $u$ (kept in $phy_u$) within one iteration.
 2: Let $T$ be the lifetime of the value generated by node $v$ (kept in $phy_v$) within one iteration.
 3: Let $S$, $T$, and all register slacks whose idle intervals span two iterations (requirement 2) be the nodes.
    // Add edges among the selected candidates on the basis of requirement 1:
 4: for each node $s_i \in RV - S - T$ do
 5:     if $S.et \geqslant s_i.st$ then add an edge from nodes $S$ to $s_i$.
 6:     if $s_i.et \geqslant T.st$ then add an edge from nodes $s_i$ to $T$.
 7:     for each node $s_j \in RV - S - T$ $(j \neq i)$ do
 8:         if $s_i.et \geqslant s_j.st$ then add an edge from nodes $s_i$ to $s_j$.
 9:     end for
10: end for
```

Step 2: constructing an RMG

In this section, we build up an *RMG* to find available physical registers that fulfill the register requirement of the chain. Our basic idea is to match the lifetime of the virtual register operands with the idle (available) time of the physical registers. We use *register slack* to refer to the idle time of a physical register. Each register slack is associated with a start time *st*, an end time *et*, and the physical register to which the slack belongs. The RMG model is defined as follows.

Definition 4.2
An RMG, $RMG = \langle RV, RE \rangle$, is a cyclic directed graph, where *RV* is the set of physical register slacks whose idle intervals span two iterations and *RE* is the set of connection edges that are added according to the first set of register requirements of the chain.

Algorithm 4.4 presents how to generate an RMG. In Algorithm 4.4, we first obtain the set of nodes on the basis of the input register usage map (lines 1–3). S and T are the source and sink nodes, respectively. The *source* node S is the time interval during which the value generated by node u is live in the physical register phy_u. The *sink* node T is the time interval during which the value generated by node v is live in the physical register phy_v before reaching v's earliest child. Besides S and T, a register slack is selected only if its idle interval spans two iterations. Then we add edges among the nodes according to requirement 1 (lines 4–12). An edge is added from s_i to s_j if the end time of s_i is greater than or equal to the start time of s_j, which means that there is an overlap between the time intervals of nodes s_i and s_j. The edge direction shows that s_i can be allocated to the source operand and s_j can be allocated to the destination operand.

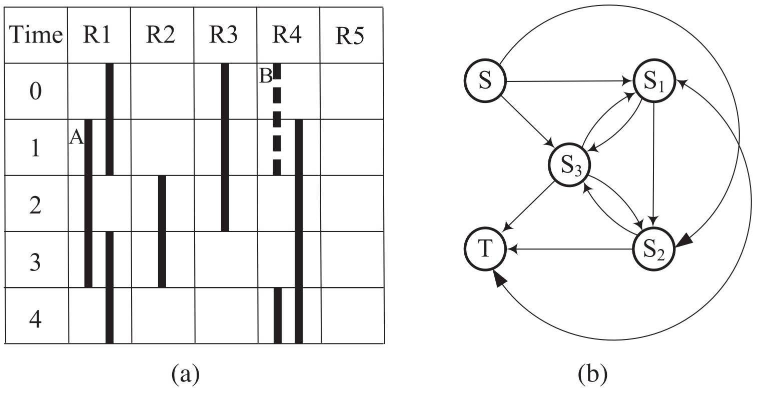


Figure 11. An example of a register-matching graph.

An example is given in Figure 11. In this example, we assume that there is an edge, '$A \xrightarrow{2} B$', in the MAG. With our technique, we can eliminate the load operation B by exploiting A's value across two iterations. According to the register usage map shown in Figure 11(a), we obtain the nodes as follows. The source S is the time range, [1,3], of A's value residing in physical register R_1 within one iteration, and the sink T is the time range, [0,1], of B's value residing in physical register R_4 within one iteration. There are three register slacks that satisfy requirement 2. They are s_1:[R_2,3,2], s_2:[R_3,2,0], and s_3:[R_5,any,any] of the unassigned physical register R_5.

The RMG of this example is shown in Figure 11(b). In this graph, S has only outgoing edges, whereas T has only incoming edges, and there is no edge between S and T because they are time intervals within one iteration and cannot be connected without using cross-iteration register slacks. As node S_3 is the slack of an unassigned register, it has incoming edges from every node except T and has outgoing edges to every node except S. All other edges are added according to requirement 1. For example, as the equation '$S_1.et = 2 = S_2.st$' holds, we add an edge from S_1 to S_2.

Step 3: finding a simple path

From the RMG, the register allocation problem is solved by finding a simple path p from S to T with length $(k + 1)$ in the RMG. The *length* of path p is the number of edges along p, and a path is *simple* if no vertex is traversed more than once. The RMG is constructed on the basis of the lifetime requirements for the virtual register operands of the chain in Figure 10, and each edge of the graph represents an available physical register matching requirement 2.

Therefore, the intermediate nodes of p are register slacks that satisfy the register requirements of the chain, and we can allocate them to the virtual register operands. The length $(k + 1)$ denotes that we should have k intermediate candidate register slacks between S and T, and the simple path avoids the case of assigning a register slack more than once. By using the k physical registers whose idle intervals are connected, we can transfer the desired value across k iterations. Therefore, we can perform register allocation on the basis of a simple path from S to T with length $(k + 1)$. Algorithm 4.5 describes how to find simple paths with length $(k + 1)$.

Algorithm 4.5 The simple path-finding algorithm.

```
Require: An RMG with $n$ vertices $v_1, v_2, \ldots, v_n$, two specified vertices $v_1$ (the source node $S$) and $v_n$ (the
    sink node $T$) of $G$, and $k$ (the cost associated with $u$ and $v$).
Ensure: All simple paths from $v_1$ to $v_n$ with length $(k + 1)$ in RMG.
  1: Construct the $n \times n$ matrix $\alpha$ of $G$ in which $\alpha(i, j) = v_j$ ($v_j$ is the name of node $v_j$) if there is an edge
     from nodes $v_i$ to $v_j$ in RMG; otherwise, $\alpha(i, j) = 0$.
  2: Raise the matrix $\alpha$ to the power of $(k + 1)$, and obtain its entry $\alpha^{k+1}(1, n)$, which is the sum of all of
     the products of nodes containing $(k + 1)$ edges from nodes $v_1$ to $v_n$.
  3: Remove all products (non-simple paths) that contain node $v_i$ or any repeated node from the entry
     $\alpha^{k+1}(1, n)$.
  4: Get all simple paths from $v_1$ to $v_n$ with length $(k + 1)$ by concatenating $v_1$ with each of the remaining
     node products in entry $\alpha^{k+1}(1, n)$.
```

The *fixed-length simple path-finding problem* is NP-complete [57]. Our algorithm is based on the following graph property.

Property: Given an $n \times n$ adjacency matrix A of a graph G, $A^k(i, j)$ is the number of (simple and nonsimple) paths from i to j, containing k edges.

In Algorithm 4.5, to enumerate all paths with length $(k + 1)$ in an RMG, we use a modified adjacency matrix α in which $\alpha(i, j) = v_j$ if there is an edge from nodes v_i to v_j; otherwise, $\alpha(i, j) = 0$. Here, v_j is the name of the jth node of the RMG. When we calculate the product of two entries, $\alpha(i,k)$ and $\alpha(k, j)$, the result is 0 if either of them is 0, and it is the concatenation $(v_k v_j)$ otherwise. From the preceding graph property, the sum of the node products in the entry $\alpha^{k+1}(i, j)$ is the sum of the paths with a length $(k + 1)$ from nodes v_i to v_j. We remove the node products that contain node v_i or any repeated node from the entry $\alpha^{k+1}(i, j)$, as these products represent non-simple paths. The simple paths are then obtained by concatenating v_i with each of the remaining

$$A: \begin{array}{c|ccccc} & S & S_1 & S_2 & S_3 & T \\ \hline S & 0 & S_1 & S_2 & S_3 & 0 \\ S_1 & 0 & 0 & S_2 & S_3 & T \\ S_2 & 0 & 0 & 0 & S_3 & T \\ S_3 & 0 & S_1 & S_2 & 0 & T \\ T & 0 & 0 & 0 & 0 & 0 \end{array}$$

(a)

$$A^2: \begin{array}{c|ccccc} & S & S_1 & S_2 & S_3 & T \\ \hline S & 0 & S_3S_1 & S_1S_2+S_3S_2 & S_1S_3+S_2S_3 & S_1T+S_2T+S_3T \\ S_1 & 0 & S_3S_1 & S_3S_2 & S_2S_3 & S_2T+S_3T \\ S_2 & 0 & S_3S_1 & S_3S_2 & 0 & S_3T \\ S_3 & 0 & 0 & S_1S_2 & S_1S_3+S_2S_3 & S_1T+S_2T \\ T & 0 & 0 & 0 & 0 & 0 \end{array}$$

(b)

$$A^3: \begin{array}{c|ccccc} & S & S_1 & S_2 & S_3 & T \\ \hline S & 0 & S_1S_3S_1+S_2S_3S_1 & S_1S_3S_2+S_2S_3S_2+S_3S_1S_2 & S_1S_2S_3+S_3S_1S_3+S_3S_2S_3 & \boxed{S_1S_2T+S_1S_3T+S_2S_3T+S_3S_1T+S_3S_2T} \\ S_1 & 0 & S_2S_3S_1 & S_2S_3S_2+S_2S_1S_2 & S_3S_1S_3+S_3S_2S_3 & S_2S_3T+S_3S_1T+S_3S_2T \\ S_2 & 0 & 0 & S_3S_1S_2 & S_3S_1S_3+S_3S_2S_3 & S_3S_1T+S_3S_2T \\ S_3 & 0 & S_1S_3S_1+S_2S_3S_1 & S_1S_3S_2+S_2S_3S_2 & S_1S_2S_3 & S_1S_2T+S_1S_3T+S_2S_3T \\ T & 0 & 0 & 0 & 0 & 0 \end{array}$$

The set of paths from S to T with length 3:
$\{S\rightarrow S_1\rightarrow S_2\rightarrow T,\ S\rightarrow S_1\rightarrow S_3\rightarrow T,\ S\rightarrow S_2\rightarrow S_3\rightarrow T,$
$S\rightarrow S_3\rightarrow S_1\rightarrow T,\ S\rightarrow S_3\rightarrow S_2\rightarrow T\}$

(c)

Figure 12. (a) The modified adjacency matrix A that represents the register-matching graph in Figure 11(b). (b) A^2. (c) A^3 and the entry $A^3(S,T)$ that contains all paths from S to T with a length of 3.

node products in entry $\alpha^{k+1}(i,j)$. An example is given in Figure 12 to illustrate our path-finding algorithm.

In this example, the modified adjacency matrix A (shown in Figure 12(a)) represents the RMG in Figure 11(b). For example, the entry $A(S, S1)$ is $S1$ as there is an edge from S to $S1$. When we raise the matrix A to power 3, the entry $A^3(S,T)$ contains all paths from S to T of length 3, and the paths are represented by the sum of the node products. For this example, we can get five paths with a length of 3 from S to T. They are '$P_1 : S \rightarrow S_1 \rightarrow S_2 \rightarrow T$', '$P_2 : S \rightarrow S_1 \rightarrow S_3 \rightarrow T$', '$P_3 : S \rightarrow S_2 \rightarrow S_3 \rightarrow T$', '$P_4 : S \rightarrow S_3 \rightarrow S_1 \rightarrow T$', and '$P_5 : S \rightarrow S_3 \rightarrow S_2 \rightarrow T$'. All of them are simple paths, and the intermediate nodes of each path can be used to allocate registers.

The time complexity of the simple path-finding algorithm is $O(RN^{k+1})$, where *RN* is the number of nodes (register slacks) in the RMG and k is a variable to control the elimination of hidden redundant load operations. For nodes u (the source node) and v (the sink node) in the *RMG*, there will be a cost $c(u,v)$ associated with nodes u and v. The cost $c(u,v)$ denotes that there are $c(u,v)$ intermediate candidate register slacks between nodes u and v. Then register allocation can be performed on the basis of the simple path from nodes u to v with length $c(u,v)$.

In our technique, k is a predefined threshold value for $c(u,v)$ to control the expansion of code sizes. If $c(u,v)$ is less than or equal to k, we will replace the redundant load operations with register operations. Otherwise, we will not eliminate the redundant load operation. In such a way, the time complexity of the proposed simple path-finding algorithm can be significantly reduced. In practice, k is fixed as a small constant. For example, in our experiments, k is set as 4, which means if a redundant load operation whose cost is greater than 4, we will not replace it with register operations. In addition, the number of nodes in the RMG, *RN*, is bounded by the number of physical registers for the target architecture. Therefore, in practice, the method is polynomial-time solvable as RN^{k+1} becomes a constant.

Step 4: instruction scheduling

We perform instruction scheduling on the basis of the results of register allocation. If multiple simple paths exist, we sort the simple paths in an increasing order in terms of the number of nodes (register slacks) that belong to unassigned registers in the path. Following this increasing order, one simple path is picked, and register allocation is performed for the register operation chain accordingly. In this way, we can use the register slacks of the assigned registers first and leave more unassigned registers for replacing other redundant load operations. For example, as shown in Figure 12, we have five simple paths with a length of 3 from S to T. In our technique, when we perform register allocation, we first choose path P_1 as it includes the least number of register slacks that belong to unassigned registers among the paths.

After register allocation has been finished for the register operation chain, each register operation is scheduled into the earliest available cycle in the time interval of its source and destination register operands. The register operations in the chain are scheduled following the order of '$MV_{k+1}, MV_k, \ldots, MV_1$', so the read–write data dependence can be obeyed. Once we obtain a legal schedule of the chain, the input DAG, schedule, and register usage map are changed accordingly. We also migrate the redundant load operations into the prologue to initialize the intermediate registers of the chain. If we cannot obtain a legal schedule for this register allocation, we pick up the next simple path and repeat the aforementioned scheduling steps until a legal schedule is found or all simple paths have been checked.

4.4. *Discussion and analysis*

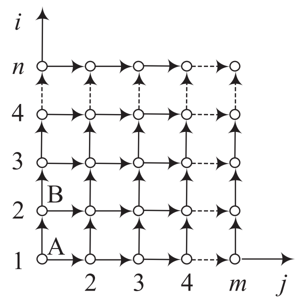
For DSP applications with nested loops, our technique can be applied to explore hidden redundant load operations in the innermost loop. It may not generate good results to replace redundant load operations by exploiting data dependence across different dimensions. For example, given the two-dimensional loop program in Figure 13(a), to transfer a reuse value from A to B shown in its iteration space in Figure 13(b), the value needs to be kept across m iterations. To achieve this, a large number of physical registers may be needed; therefore, by doing this, we may not get any benefits. Thus, we focus on reducing memory accesses in the innermost loops for DSP applications.

During the compilation of DSP applications, the compiler must decide how to allocate variables to a finite set of registers. One register can only be used by one variable at one moment. During register allocation, register spilling may be caused when there are not enough registers to hold all variables. In our technique, hidden redundant load operations are iteratively replaced with register operations. We build up an RMG to describe the time relations among available slacks of physical registers. Register operations will be allocated only when there are available clock cycles. Therefore, our technique will not cause register spilling.

The time complexity of the LSMAR algorithm is analyzed as follows. As shown in Algorithm 4.1, the LSMAR algorithm has two phases. In phase 1, for algorithm MAG(), it takes at most $O(|V|^2)$ to finish, as we need to process each pair of memory operations and the number of memory operations is bounded by $|V|$ ($|V|$ is the number of nodes in the DAG). For step 2, it takes at most $O(|V|)$ to finish as $|MV|$ is the subset of $|V|$. In phase 2, as mentioned earlier, to replace a redundant load operation in the MAG with edge weight k, it takes $O(RN^{k+1})$ for the simple path-finding algorithm to finish, and at most $O(RN^{k+1})$ paths are generated (RN is the number of register slacks in the register

```
for i = 1 to n do
    for j = 1 to m do
        A[i, j] = A[i, j-1]+5;
        B[i, j] = B[i-1, j]*2;
    endfor
endfor
```

(a)



(b)

Figure 13. (a) A nested loop. (b) Its iteration space.

usage map and is bounded by the number of physical registers). For each simple path, suppose that the schedule length is L, then it takes at most $O(k \times L)$ to schedule $(k+1)$ register operations. L is at most $O(|V| \times \max\_latency)$ ($\max\_latency$ is the maximum latency among all nodes), which is the case when all nodes are scheduled sequentially. Therefore, the complexity of the LSMAR algorithm is $O(|V|^2 \times RN^{k+1} \times k \times \max\_latency)$ that is obtained from $O(|V| \times RN^{k+1} \times k \times |V| \times \max\_latency)$. RN and $\max\_latency$ are constants, and k is set as a small constant in practice; therefore, in this case, the complexity of the LSMAR algorithm is $O(|V|^2)$.

5. EXPERIMENTS

We implement our technique into the Trimaran compiler [4] and conduct experiments using a set of benchmarks from DSPstone [5] and MiBench [6] on the cycle-accurate VLIW simulator of Trimaran. In this section, we first discuss our implementation and simulation environment in Section 5.1 and then introduce our benchmark programs in Section 5.2. The experimental results and discussion are presented in Section 5.3.

5.1. *Experimental setup*

We implement the LSMAR algorithm into Trimaran for code generation. Major modifications are performed to integrate our technique into the instruction scheduling and register allocation modules of Trimaran. Basically, our technique is applied on the code generated by the cycle scheduler and register allocator of Trimaran [4]. Modulo scheduling [58], the widely used software pipelining technique, is applied on the code generated by our LSMAR technique and the baseline scheme of Trimaran [4].

To compare our technique with the baseline scheme of the compiler, we use the VLIW simulator of Trimaran as our test platform, which has the same architecture as our system model presented in Section 3.1. The configurations for the VLIW architecture are shown in Table I. The memory system consists of a 32K four-way associative instruction cache and a 32K four-way associative data cache, both with a block size of 64 B. In the system, there is a register file, which will be configured as 16, 32, and 64 registers in the experiments.

5.2. *Benchmark programs*

In the experiments, 21 benchmarks are selected from DSPstone [5] and MiBench [6]. These benchmarks contain loops or hidden redundant load operations. The details of these benchmarks are shown in Table II. For each benchmark, the code is first generated using Trimaran and tested on the simulator with the VLIW configurations in Table I. Both the fixed-point and floating-point versions of the benchmarks from DSPstone are evaluated. However, two benchmarks, *fft_stage_scaled* and *fft_input_scaled*, have only fixed-point versions. For each benchmark from Mibench, the set of large data is used as the input in the experiments.

In Table II, the columns 'Total', 'Load', 'IALU', and 'FALU' represent the number of total dynamic operations, dynamic load operations, dynamic integer operations, and dynamic floating-point operations, respectively. In this paper, memory access operations are divided into dynamic

Table I. The very-long-instruction-word configurations.

Parameter	Configuration
Functional units	2 integer ALUs, 2 floating-point ALUs, 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, 7 cycles for load in level 2 cache, 35 cycles for load in main memory, 1 cycle for store in cache, 7 cycles for store in level 2 cache, 35 cycles for store in main memory, 1 cycle for branch
Register file	16/32/64 registers

ALU, arithmetic logic unit.

Table II. The descriptions and characteristics of the benchmarks.

		Number of dynamic instructions					
Benchmarks	Input	Total	Load	IALU	FALU	Load/total (%)	IALU/total (%)
DSPstone							
Convolution	—	305	51	115	38	16.72	37.70
Convolution-fix	—	271	36	165	0	13.28	60.89
dot_product	—	757	109	125	203	14.40	16.51
dot_product-fix	—	1,020	300	515	0	29.41	50.49
IIR	—	597	84	331	31	14.07	55.44
IIR-fix	—	478	39	295	0	8.16	61.72
fir	—	364	32	111	50	8.85	30.58
fir-fix	—	438	54	243	0	12.33	55.48
fir2dim	—	3,377	622	1,391	326	18.42	41.19
fir2dim-fix	—	2,658	451	1,606	0	16.97	60.42
lms	—	647	92	315	36	14.22	48.69
lms-fix	—	657	87	373	0	13.24	56.77
matrix 1×3	—	263	28	131	27	10.65	49.81
matrix 1×3-fix	—	152	28	91	0	18.42	59.87
matrix	—	13,960	3,003	6,210	1002	21.51	44.48
matrix-fix	—	14,558	3,003	7,810	0	20.63	53.65
matrix2	—	13,270	2,713	5,910	1202	20.44	49.54
matrix2-fix	—	13,868	2,713	7,710	0	19.60	55.60
n_complex_updates	—	1,498	327	718	65	21.83	47.93
n_complex_updates-fix	—	1,291	172	735	0	13.32	56.93
n_real_updates	—	788	155	400	18	19.67	50.76
n_real_updates-fix	—	590	53	332	0	8.98	56.27
fft_stage_scaled	—	470,447	67,525	276,993	8192	14.35	58.88
fft_input_scaled	—	347,467	47,045	215,463	8192	13.54	62.01
Average						16.52	50.90
Mibench							
blowfishencrypt	Large	1.12M	0.26M	0.64M	0	23.31	57.06
blowfishdecrypt	Large	1.12M	0.26M	0.64M	0	23.32	57.07
cjpeg	Large	51.18M	6.92M	25.04M	0	13.52	48.92
djpeg	Large	29.88M	5.87M	18.30M	0	19.64	61.24
gsmencode	Large	387.51M	74.70M	220.87M	0	19.28	57.00
gsmdecode	Large	780.26M	40.45M	334.09M	0	5.18	42.82
rawcaudio	Large	885.43M	39.98M	312.61M	0	4.52	35.31
rawdaudio	Large	744.68M	33.33M	265.04M	0	4.48	35.59
Average						14.16	49.38

FALU, floating-point arithmetic logic unit; IALU, integer arithmetic logic unit; IIR, infinite impulse response.

operations and static operations. Dynamic operations refer to the operations at run time when executing the application with the input, whereas static operations refer to the operations at compile time. For dynamic operations, the performance of each benchmark is affected by the number and types of dynamic operations. 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 column 'Load/total (%)' of Table II. It can be observed that a large fraction of the dynamic operations, on average 16.52% and 14.16%, are dynamic load operations for DSPstone and MiBench, respectively. This shows that further memory access optimization is necessary to improve DSP performance and memory power.

5.3. Results and discussion

In the experiments, we obtain the results of reducing memory access, speeding up performance, and expanding code size with various register constraints on the code generated by our LSMAR algorithm. We compare these results with that of the baseline scheme generated by Trimaran [4]. The modulo scheduling module is enabled in both the baseline scheme and our approach. In the experiments, as our register allocation results are based on the path-finding method (in Section 6.2.3),

Table III. The number of dynamic load operations under various register constraints for benchmarks.

	Number of dynamic load operations								
	Number of registers = 16			Number of registers = 32			Number of registers = 64		
Benchmark	Base [4]	LSMAR	IMP (%)	Base [4]	LSMAR	IMP (%)	Base [4]	LSMAR	IMP (%)
DSPstone									
Conv.	51	33	35.29	51	33	35.29	51	33	35.29
Conv.-fix	36	26	27.78	36	26	27.78	36	26	27.78
product	109	78	28.44	109	78	28.44	109	78	28.44
product-fix	300	215	28.33	300	215	28.33	300	215	28.33
IIR	84	65	22.62	84	65	22.62	84	65	22.62
IIR-fix	39	27	30.77	39	27	30.77	39	27	30.77
fir	38	26	31.58	38	26	31.58	32	22	31.25
fir-fix	57	42	26.32	57	42	26.32	54	38	29.63
fir2dim	622	518	16.72	622	518	16.72	622	518	16.72
fir2dim-fix	468	384	17.95	471	384	18.47	451	384	14.86
lms	92	61	33.70	97	65	32.99	92	61	33.70
lms-fix	104	71	31.73	104	71	31.73	87	60	31.03
mat.1x3	28	20	28.57	28	20	28.57	28	20	28.57
mat.1x3-fix	28	20	28.57	28	20	28.57	28	20	28.57
mat.	3,003	2,487	17.18	3,003	2,487	17.18	3,003	2,487	17.18
mat.-fix	3,003	2,487	17.18	3,003	2,487	17.18	3,003	2,487	17.18
mat.2	2,713	2,202	18.84	2,713	2,202	18.84	2,713	2,202	18.84
mat.2-fix	2,718	2,202	18.98	2,718	2,202	18.98	2,718	2,202	18.98
n_complex	332	261	21.39	338	261	22.78	327	261	20.18
n_complex-fix	173	132	23.70	183	132	27.87	172	132	23.26
n_real	155	133	14.19	155	133	14.19	155	133	14.19
n_real-fix	53	41	22.64	53	41	22.64	53	41	22.64
fft_stage-fix	73,607	60,262	18.13	67,531	54,396	19.45	67,525	53,858	20.24
fft_input-fix	53,127	40,137	24.45	50,061	37,611	24.87	47,045	35,166	25.25
Average improvement (%)			24.38			24.67			24.70
Mibench									
bfencrypt	0.30M	0.26M	13.88	0.28M	0.24M	14.12	0.26M	0.22M	14.66
bfdecrypt	0.30M	0.26M	13.88	0.28M	0.24M	14.12	0.26M	0.22M	14.66
cjpeg	7.81M	7.26M	6.98	7.17M	6.64M	7.33	6.92M	6.38M	7.84
djpeg	6.81M	6.18M	9.23	6.67M	6.04	9.45	5.87M	5.28M	10.12
gsmencode	80.48M	71.19M	11.54	75.11M	65.59M	12.67	74.70M	65.00M	12.98
gsmdecode	48.93M	43.44M	11.22	41.42M	36.50M	11.88	40.45M	35.50M	12.23
rawcaudio	40.00M	36.67M	8.33	39.98M	36.59M	8.47	39.98M	36.40M	8.95
rawdaudio	33.33M	30.18M	9.44	33.33M	30.18M	9.44	33.33M	30.18M	9.44
Average improvement (%)			10.56			10.94			11.36

IIR, infinite impulse response; IMP, improvement; LSMAR, loop scheduling with memory access reduction.

we set up 4 to be the maximum number of delays adopted for eliminating redundant load operations. By doing this, we can avoid a big expansion of the code, as the number of newly generated register operations is limited accordingly.

In Tables III to V, we give the results for all benchmarks tested with 16, 32, and 64 registers, respectively. In these tables, the column ‘Base’ refers to the results generated by the baseline scheme of Trimaran [4], and the column ‘LSMAR’ represents the results obtained by our LSMAR technique. Column ‘IMP (%)’ represents the improvement obtained by our LSMAR algorithm compared with the baseline scheme of Trimaran. In the following, we present and analyze the results in terms of reducing memory access, improving overall performance, and expanding code size in Sections 5.3.1–5.3.3, respectively.

5.3.1. Reducing memory access. The percentages of reduction in the number of dynamic load operations for benchmarks DSPstone and MiBench, when we compare our algorithm with the baseline scheme of Trimaran, are shown in Table III. From experimental results, it can be observed that our

Table IV. The improvement in performance under various register constraints for benchmarks.

	Number of execution cycles								
	Number of registers = 16			Number of registers = 32			Number of registers = 64		
Benchmark	Base [4]	LSMAR	IMP (%)	Base [4]	LSMAR	IMP (%)	Base [4]	LSMAR	IMP (%)
				DSPstone					
Conv.	461	402	12.80	461	402	12.80	423	367	13.13
Conv.-fix	482	448	7.13	482	448	7.13	458	423	7.65
product	1,262	1,134	10.12	1,262	1,134	10.12	1,254	1,126	10.24
product-fix	2,235	2,046	8.44	2,235	1,868	8.44	2,210	2,022	8.51
IIR	968	770	20.44	861	672	21.92	861	672	21.92
IIR-fix	876	717	18.18	794	647	18.50	794	647	18.50
fir	627	581	7.27	562	520	7.43	562	520	7.43
fir-fix	860	818	4.87	683	649	4.95	683	649	4.95
fir2dim	4,172	4,044	3.08	4,053	3,927	3.12	3,861	3,724	3.55
fir2dim-fix	3,247	3,318	3.19	3,192	3,086	3.33	3,204	3,095	3.41
lms	992	767	22.66	982	757	22.89	954	732	23.30
lms-fix	1,060	833	21.45	1,012	792	21.78	971	757	21.99
mat.1x3	520	410	21.12	520	410	21.12	520	410	21.12
mat.1x3-fix	444	363	18.22	444	363	18.22	444	336	18.22
mat.	10,611	9,973	6.01	10,611	9,973	6.01	10,611	9,973	6.01
mat.-fix	13,384	12,066	9.85	13,384	12,066	9.85	13,384	12,066	9.85
mat.2	9,770	7,525	22.98	9,770	7,525	22.98	9,770	7,525	22.98
mat.2-fix	9,672	8,930	7.67	9,672	8,930	7.67	9,672	8,930	7.67
n_complex	1,831	1,571	14.20	1,425	1,220	14.37	1,403	1,196	14.76
n_complex-fix	1,722	1,470	14.66	1,651	1,404	14.98	1,518	1,285	15.34
n_real	881	758	13.98	796	684	14.06	796	684	14.06
n_real-fix	663	574	13.48	646	558	13.55	646	558	13.55
fft_stage-fix	332,363	279,617	15.87	323,355	270,260	16.42	315,305	262,392	16.78
fft_input-fix	261,398	208,935	20.07	250,547	195,477	21.98	241,299	187,393	22.34
Average improvement (%)			13.24			13.48			13.65

Table IV. *Continued.*

	Number of execution cycles								
	Number of registers = 16			Number of registers = 32			Number of registers = 64		
Benchmark	Base [4]	LSMAR	IMP (%)	Base [4]	LSMAR	IMP (%)	Base [4]	LSMAR	IMP (%)
Mibench									
bfencrypt	1.28M	1.09M	14.66	1.21M	1.03M	14.87	1.13M	0.96M	15.15
bfdecrypt	1.28M	1.09M	14.66	1.21M	1.03M	14.87	1.13M	0.96M	15.15
cjpeg	20.72M	19.36M	6.56	20.65M	19.09M	7.13	20.61M	19.08M	7.42
djpeg	21.26M	19.44M	8.56	20.77M	18.72M	9.88	18.79M	16.83M	10.45
gsmencode	383.20M	361.13M	5.76	375.60	352.65M	6.11	371.80M	346.30M	6.85
gsmdecode	588.36M	554.35M	5.78	566.96M	531.07M	6.33	546.47M	507.56M	7.12
rawcaudio	714.43M	678.85M	4.98	711.00M	673.89M	5.22	711.00M	672.25M	5.45
rawdaudio	588.68M	558.95M	5.05	588.68M	556.95M	5.39	585.31M	552.12M	5.67
Average improvement (%)			8.26			8.73			9.16

IIR, infinite impulse response; IMP, improvement; LSMAR, loop scheduling with memory access reduction.

scheduling algorithm can effectively reduce the number of memory accesses under various register constraints. For DSPstone, with the configurations of 16, 32, and 64 registers, our LSMAR technique contributes to average reductions of 24.38%, 24.67%, and 24.70% in the number of dynamic load operations, respectively. For MiBench, with the configurations of 16, 32, and 64 registers, our LSMAR technique achieves average reductions of 10.56%, 10.94%, and 11.36% in memory access, respectively.

5.3.2. Improving performance. The overall improvement in performance for benchmarks from DSPstone and MiBench is shown in Table IV. On average, for DSPstone, with the configurations of 16, 32, and 64 registers, our LSMAR technique contributes to a respective 13.24%, 13.48%, and 13.65% improvement in performance. For MiBench, with the configurations of 16, 32, and 64 registers, on average, our LSMAR technique achieves a performance improvement of 8.26%, 8.73%, and 9.16% , respectively. The reasons for the improvement are shown as follows.

First, our technique eliminates load operations within loops, which are the most time-consuming part of DSP applications. With the reduction in the number of memory accesses, the number of cache misses is reduced accordingly. Second, load operations are usually on the critical path of the execution of the loop kernels of DSP applications. After performing our algorithm, we replace the redundant load operations across different iterations by register operations with less execution time. Thus, the data dependence in the loop is changed accordingly. As fewer redundant load operations remain in the loop, more operations that previously depended on them can be scheduled earlier in the new loop. Instruction-level parallelism improves accordingly, with higher utilization of multiple FUs. This leads to a reduction in the length of the schedule of the loop. Third, when scheduling register operations, we consider register constraints and FU constraints. We only remove a redundant load when we have enough registers for replacement. This alleviates the spilling of registers that has a great impact on performance.

5.3.3. Expanding code size. This section presents the percentages of code size expansion for DSPstone and MiBench. The experimental results are shown in Table V. In this table, static instructions refer to the instructions at compile time, and the code size of each benchmark is evaluated by the number of static instructions. For DSPstone, with the configurations of 16, 32, and 64 registers, our LSMAR technique leads to an average expansion in code size of 1.90%, 2.00%, and 2.02%, respectively. For most of the benchmarks in DSPstone, our technique only introduces one or two extra static instructions. For benchmarks n_complex and n_complex-fix with more than 150 static instructions for the baseline scheme, the proposed LSMAR technique incurs four extra static instructions. For benchmarks that have a large number of static instructions (i.e., fft_stage/fix and fft_input/fix), our technique leads to a maximum of 3.55% in code size expansion. These results show that with the setting of the threshold value, our technique can effectively replace redundant load operations and introduces limited expansion in code size.

For MiBench, with the configurations of 16, 32, and 64 registers, on average, our LSMAR technique leads to an expansion in code size of 0.81%, 0.87%, and 0.90%, respectively. Compared with the benchmarks in DSPstone, the benchmarks in MiBench have a larger code size. For example, benchmarks bfencrypt, bfdecrypt, and gsmdecode have several thousands of static instructions. For these large-sized benchmarks, our technique introduces a maximum of 1.23% in code size expansion. For benchmarks cjpeg, djpeg, and gsmencode that have more than ten thousands of static instructions, our technique can achieve even better performance. For these benchmarks, our technique incurs less than 1% code size expansion.

The reason for the expansion is that our technique may use more than one register operation to replace the redundant load operation and promote it into the prologue. However, in our technique, the expansion in code size is controlled by the maximum number of delays that determines the maximum number of register operations used to replace one redundant load operation. From the experimental results, we can find that the code size expansion is very small. This shows the effectiveness of the proposed technique. With such a small expansion in code size, our technique is very suitable for embedded systems.

Table V. The expansion in code size under various register constraints for benchmarks.

	Number of static instructions								
	Number of registers = 16			Number of registers = 32			Number of registers = 64		
Benchmark	Base [4]	LSMAR	EXP (%)	Base [4]	LSMAR	EXP (%)	Base [4]	LSMAR	EXP (%)
				DSPstone					
Conv.	43	44	2.33	43	44	2.33	43	44	2.33
Conv.-fix	45	46	2.22	45	46	2.22	45	46	2.22
product	64	65	1.56	64	65	1.56	64	65	1.56
product-fix	30	31	3.33	30	31	3.33	30	31	3.33
IIR	137	139	1.46	137	139	1.46	137	139	1.46
IIR-fix	140	142	1.43	140	142	1.43	140	142	1.43
fir	81	82	1.23	81	82	1.23	81	82	1.23
fir-fix	93	94	1.08	93	94	1.08	93	94	1.08
fir2dim	278	280	0.72	266	268	0.75	266	268	0.75
fir2dim-fix	323	325	0.62	311	313	0.64	311	313	0.64
lms	160	162	1.25	140	142	1.43	140	142	1.43
lms-fix	193	195	1.04	161	163	1.24	137	139	1.46
mat.1x3	72	74	2.78	72	74	2.78	72	74	2.78
mat.1x3-fix	38	40	5.26	38	40	5.26	38	40	5.26
mat.	154	156	1.30	154	156	1.30	154	156	1.30
mat.-fix	149	151	1.34	149	151	1.34	149	151	1.34
mat.2	166	168	3.03	166	168	3.03	166	168	3.03
mat.2-fix	183	185	1.09	183	185	1.09	183	185	1.09
n_complex	189	193	2.12	165	169	2.42	165	169	2.42
n_complex-fix	203	207	1.97	163	167	2.45	159	163	2.52
n_real	100	102	2.00	100	102	2.00	100	102	2.00
n_real-fix	98	100	2.04	98	100	2.04	98	100	2.04
fft_stage-fix	364	372	2.18	316	323	2.22	292	299	2.34
fft_input-fix	348	359	3.19	332	343	3.32	276	286	3.55
Average expansion (%)			1.90			2.00			2.02

Table V. *Continued.*

	Number of static instructions								
	Number of registers = 16			Number of registers = 32			Number of registers = 64		
Benchmark	Base [4]	LSMAR	EXP (%)	Base [4]	LSMAR	EXP (%)	Base [4]	LSMAR	EXP (%)
				Mibench					
bfencrypt	8,224	8,320	1.17	7,878	7,971	1.18	7,430	7,521	1.23
bfdecrypt	7,590	7,677	1.14	7,340	7,425	1.16	7,084	7,169	1.20
cjpeg	14,402	14,450	0.33	14,436	14,492	0.39	14,741	14,501	0.41
djpeg	18,329	18,448	0.65	17,874	17,972	0.71	17,385	17,514	0.74
gsmencode	11,988	12,087	0.74	11,542	11,645	0.89	11,234	11,336	0.91
gsmdecode	5,783	5,826	0.75	5,783	5,830	0.82	5,899	5,949	0.85
rawcaudio	230	232	0.86	206	208	0.89	206	208	0.89
rawdaudio	196	198	0.85	180	182	0.87	180	182	0.87
Average expansion (%)			0.81			0.87			0.90

EXP, expansion; IIR, infinite impulse response; LSMAR, loop scheduling with memory access reduction.

5.3.4. Effectiveness of register allocation and instruction scheduling. This section compares the proposed LSMAR technique with the REALM technique in [55]. REALM explores hidden redundant load operations and migrates them outside loops on the basis of loop-carried data dependence analysis. Figure 14 presents the experimental results of our LSMAR technique and the REALM algorithm for Mibench benchmarks in terms of the number of dynamic load operations, the number of execution cycles, and the expansion in code size. To make a fair comparison, in the experiments, we compare both LSMAR and REALM with the baseline scheme of Trimaran. The experimental results show the improvement of LSMAR and REALM over the baseline scheme.

In REALM, it first builds up a data flow graph to describe the inter-iteration data dependence among multiple memory operations. Then code transformation is performed by exploiting these dependence with registers to hold the values of redundant loads and migrating these loads outside loops. Therefore, REALM can effectively eliminate redundant load operations of loops. For the problem of loop scheduling with memory access reduction, only identifying redundant load operations may not obtain a feasible and effective loop schedule. Our LSMAR technique further considers how to perform register allocation and partial instruction scheduling. Compared with REALM, the proposed LSMAR technique is a comprehensive and complete solution that can effectively solve the problem of loop scheduling with memory access reduction.

Register allocation and partial instruction scheduling are two important phases in our LSMAR technique. By exploring these two phases, LSMAR fully utilizes all registers and presents an efficient and practical solution. From the experimental results in Figure 14(a), compared with the baseline scheme, REALM can reduce the number of dynamic load operations by 8.1%, whereas LSMAR can further reduce the number of dynamic load operations with a result of 11.4% reduction. For the number of execution cycles, as shown in Figure 14(b), LSMAR and REALM can improve the performance by 9.16% and 6.31%, respectively. This shows that register allocation and partial instruction scheduling can help improve the performance. Because register allocation and partial instruction scheduling may introduce extra processing instructions, LSMAR will incur the

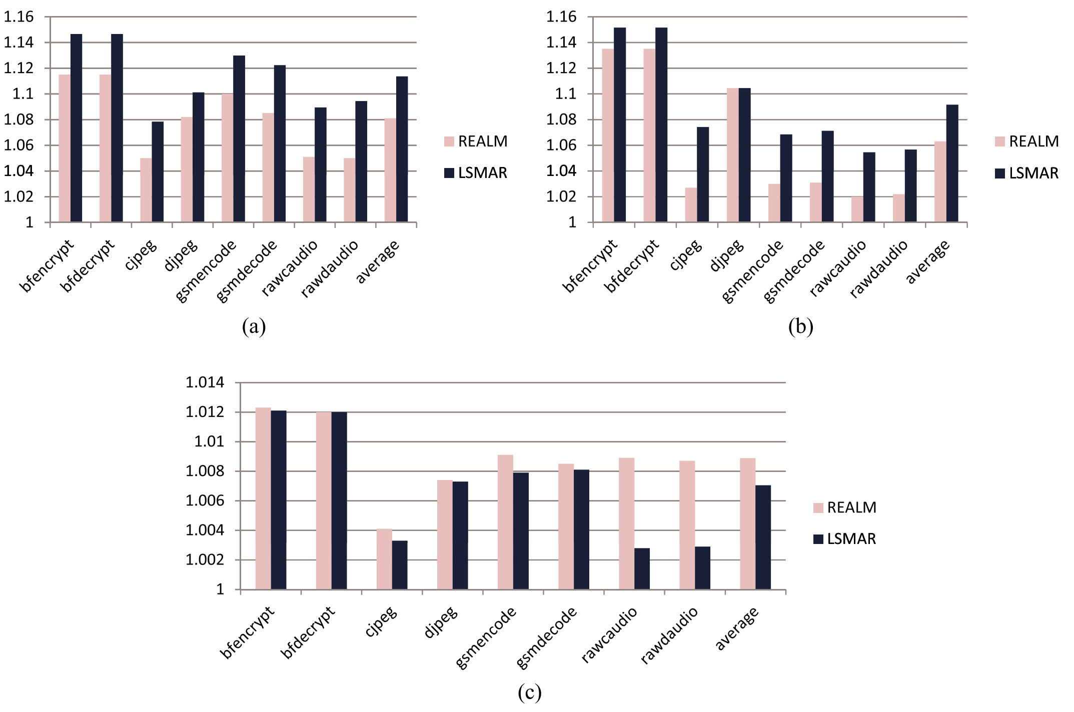


Figure 14. Comparison of our loop scheduling with memory access reduction (LSMAR) technique and the redundant load exploration and migration (REALM) algorithm in [55]. (a) The number of dynamic load operations . (b) The number of execution cycles. (c) The expansion in code size.

expansion in code size. As shown in Figure 14(c), LSMAR only slightly increases (by 0.71%) the code size, and LSMAR introduces less code size expansion compared with REALM (by 0.89%). From the results, the LSMAR technique can effectively improve the performance with negligible code size overhead.

6. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a loop scheduling algorithm, LSMAR, to reduce memory accesses for DSP applications with loops. In our approach, we built up the MAG to describe the inter-iteration dependence among memory operations. From this graph, we performed register allocation and instruction scheduling to eliminate redundant load operations subject to register constraints. We implemented our technique into the Trimaran compiler [4] and conducted experiments using a set of benchmarks from DSPstone [5] and MiBench [6] on the basis of the cycle-accurate VLIW simulator of Trimaran. The experimental results show that our technique can effectively reduce the number of memory accesses.

There are several directions for future work. How to extend our technique to the multicluster VLIW architectures is one future direction. In addition, our technique currently works well for DSP applications with simple control flow. How to extend our algorithm to general-purpose applications with complicated control branches is another important problem to be addressed in the future. Finally, energy and temperature are important issues in embedded systems. How to combine our technique with efficient energy optimization techniques is another important problem that needs to be investigated.

ACKNOWLEDGEMENTS

The work described in this paper is partially supported by grants from the Innovation and Technology Support Programme of Innovation and Technology Fund of the Hong Kong Special Administrative Region, China (ITS/082/10), the Germany/Hong Kong Joint Research Scheme sponsored by the Research Grants Council of Hong Kong and the Germany Academic Exchange Service of Germany (reference no. G_HK021/12), the National Natural Science Foundation of China (project 61272103), and the Hong Kong Polytechnic University (4-ZZD7, G-YK24, and G-YM10).

REFERENCES

1. Texas Instruments, Inc. TMS320C6000 Optimizing Compiler User's Guide, 2001.
2. Texas Instruments, Inc. TMS320C6000 CPU and Instruction Set Reference Guide, 2000.
3. Chang PP, Mahlke SA, Chen WY, Warter NJ, Hwu WW. Impact: an architectural framework for multiple-instruction-issue processors. *Proceedings of the 18th International Symposium on Computer Architecture*, Toronto, Ontario, Canada, 1991; 266–275.
4. The Trimaran Compiler Research Infrastructure. Available from: http://www.trimaran.org/ [Accessed on 2012].
5. Zivojnovic V, Pees S, Schlager C, Willems M, Schoenen R, Meyr H. DSP processor/compiler co-design: a quantitative approach. In *Proceedings of the 9th International Symposium on System Synthesis, ISSS '96*. IEEE Computer Society: Washington, DC, USA, 1996; 108–113.
6. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB. Mibench: a free, commercially representative embedded benchmark suite. *Proceedings of the IEEE International Workshop on Workload Characterization*, Dallas, Texas, 2001; 3–14.
7. Zhang Y, Yang J. Procedural level address offset assignment of DSP applications with loops. *Proceedings of the 2003 International Conference on Parallel Processing*, Kaohsiung, Taiwan, 2003; 21–28.
8. Leventhal S, Yuan L, Bambha NK, Bhattacharyya SS, Qu G. DSP address optimization using evolutionary algorithms. *Proceedings of the 2005 Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, 2005; 91–98.
9. Franchetti F, Voronenko Y, Püschel M. FFT program generation for shared memory: SMP and multicore. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06*. ACM: New York, NY, USA, 2006; 115–126.
10. Bonelli A, Franchetti F, Lorenz J, Puschel M, Ueberhuber CW. Automatic performance optimization of the discrete Fourier transform on distributed memory computers. *Proceedings of the International Symposium on Parallel and Distributed Processing and Application (ISPA)*, 2006; 818–832.

11. Ko M, Shen C, Bhattacharyya SS. Memory-constrained block processing for DSP software optimization. *Journal of Signal Processing Systems (JSPS)* 2008; **50**(2):163–177.
12. Salmela P, Gu R, Bhattacharyya SS, Takala J. Efficient parallel memory organization for turbo decoders. *Proceedings of the European Signal Processing Conference*, Poznan, Poland, 2007; 831–835.
13. Murthy PK, Bhattacharyya SS. Buffer merging—a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 2004; **9**(2):212–237.
14. Yan J, Zhang W. Exploiting virtual registers to reduce pressure on real registers. In *ACM Transactions on Architecture and Code Optimization*, Vol. 4. ACM: New York, NY, USA, 2008; 3:1–3:18.
15. Jun Y, Zhang W. Virtual registers: reducing register pressure without enlarging the register file. *Proceedings of the 2007 International Conference on High Performance Embedded Architectures & Compilers*, Ghent, Belgium, 2007; 57–70.
16. Wang Y, Tang Y, Jiang Y, Chung JG, Sone SS, Lim MS. Novel memory reference reduction methods for FFT implementations on DSP processors. *IEEE Transactions on Signal Processing (TSP)* 2007; **55**(5):2338–2349.
17. Salmela P, Shen CC, Bhattacharyya SS, Takala J. Register file partitioning with constraint programming. *Proceedings of the 2006 International System-on-Chip Symposium*, Tampere, Finland, 2006; 137–140.
18. Raghavan P, Lambrechts A, Jayapala M, Catthoor F, Verkest D, Corporaal H. Very wide register: an asymmetric register file organization for low power embedded processors. *Proceedings of the 11th IEEE/ACM DATE Conference: Design, Automation and Test in Europe*, Nice, France, 2007; 1066–1071.
19. Puschel M, Moura JMF, Johnson J, Padua D, Veloso M, Singer B, Xiong J, Franchetti F, Gacic A, Voronenko Y, Chen K, Johnson RW, Rizzolo N. Spiral: code generation for DSP transforms. *Proceedings of the IEEE* 2005; **93**(2):232–275.
20. Xue J, Cai Q. A lifetime optimal algorithm for speculative PRE. *ACM Transactions on Architecture and Code Optimization (TACO)* 2006; **3**(2):115–155.
21. Song Y, Xu R, Wang C, Li Z. Data locality enhancement by memory reduction. *Proceedings of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, 2001; 50–64.
22. Liu L, Li Z, Sameh AH. Analyzing memory access intensity in parallel programs on multicore. *Proceedings of the 22nd Annual International Conference on Supercomputing*, Island of Kos, Greece, 2008; 359–367.
23. Ding Y, Li Z. A compiler scheme for reusing intermediate computation results. *Proceedings of the 2004 Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, 2004; 279–291.
24. Wang Z, Sha EHM, Wang Y. Partitioning and scheduling DSP applications with maximal memory access hiding. *EURASIP Journal on Applied Signal Processing* 2002; **2002**(1):926–935.
25. Huang C, Ravi S, Raghunathan A, Jha NK. Generation of heterogeneous distributed architectures for memory-intensive applications through high-level synthesis. *IEEE Transactions on VLSI Systems* 2007; **15**(11):1191–1204.
26. Khouri KS, Lakshminarayana G, Jha NK. Memory binding for performance optimization of control-flow intensive behavioral descriptions. *IEEE Transactions on VLSI Systems* 2005; **13**(5):513–524.
27. Huang C, Ravi S, Raghunathan A, Jha N. Eliminating memory bottlenecks for a JPEG encoder through distributed logic-memory architecture and computation-unit integrated memory. *Proceedings of IEEE Custom Integrated Circuit Conference*, San Jose, California, 2005; 239–242.
28. Balasa F, Kjeldsberg PG, Vandecappelle A, Palkovic M, Hu Q, Zhu H, Catthoor F. Storage estimation and design space exploration methodologies for the memory management of signal processing applications. *Journal of Signal Processing Systems* 2008; **53**(2):51–71.
29. Kjeldsberg PG, Catthoor F, Verdoolaege S, Palkovic M, Vandecappelle A, Hu Q, Aas EJ. Guidance of loop ordering for reduced memory usage in signal processing applications. *Journal of Signal Processing Systems* 2008; **53**(3):301–321.
30. Balasa F, Kjeldsberg PG, Palkovic M, Vandecappelle A, Catthoor F. Loop transformation methodologies for array-oriented memory management. *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors*, Steamboat Springs, CO, 2006; 205–212.
31. Achteren TV, Deconinck G, Catthoor F, Lauwereins R. Data reuse exploration techniques for loop-dominated application. *Proceedings of the IEEE/ACM DATE Conference: Design, Automation and Test in Europe*, Paris, France, 2002; 428–435.
32. Panda PR, Catthoor F, Dutt ND, Danckaert K, Brockmeyer E, Kulkarni C, Vandercappelle A, Kjeldsberg PG. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 2001; **6**(2):149–206.
33. Kandemir M. A compiler-based approach for improving intra-iteration data reuse. *Proceedings of the Conference on Design, Automation and Test in Europe*, Paris, France, 2002; 984–990.
34. Hu J, Kandemir M, Vijaykrishnan N, Irwin MJ. Analyzing data reuse for cache reconfiguration. *ACM Transactions on Embedded Computing Systems (TECS)* 2005; **4**(4):851–876.
35. Mamidipaka M, Dutt N, Hirschberg D. Efficient power reduction techniques for time multiplexed address buses. *2000 Great Lakes Symposium on VLSI*, Kyoto, Japan, 2000; 207–212.
36. Mamidipaka M, Hirschberg D, Dutt N. Adaptive low power encoding techniques using self organizing lists. *IEEE Transactions on VLSI Systems* 2003; **11**(5):827–834.

37. Issenin I, Brockmeyer E, Miranda M, Dutt N. Data reuse analysis technique for software-controlled memory hierarchies. In *Proceedings of the Conference on Design, Automation and Test in Europe*, Vol. 1*, DATE '04*. IEEE Computer Society: Washington, DC, USA, 2004; 202–207.
38. Feautrier P. Compiling for massively parallel architectures: a perspective. *Microprocessing and Microprogramming* 1995; **41**:425–439.
39. Lefevre V, Feautrier P. Optimizing storage size for static control programs in automatic parallelizers. *The Conference on EuroPar*, Passau, Germany, 1997; 356–363.
40. Weinhardt M, Luk W. Memory access optimisation for reconfigurable systems. *Proceedings of IEEE—Computers and Digital Techniques* 2001; **148**(3):105–112.
41. Kolson D, Nicolau A, Dutt N. Elimination of redundant memory traffic in high-level synthesis. *IEEE Transactions on Computer-aided Design* 1996; **15**(11):1354–1363.
42. Wang Z, Sha EHM, Hu XS. Combined partitioning and data padding for scheduling multiple loop nests. *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Atlanta, Georgia, USA, 2001; 67–75.
43. Song Y, Li Z. Applying array contraction to a sequence of DOALL loops. *Proceedings of the 2004 International Conference on Parallel Processing*, Montreal, Quebec, Canada, 2004; 46–53.
44. Sarkar V, Gao GR. Optimization of array accesses by collective loop transformations. *Proceedings of the 5th International Conference on Supercomputing*, Cologne, West Germany, 1991; 194–205.
45. Gao GR. A maximally pipelined tridiagonal linear equation solver. *Journal of Parallel and Distributed Computing (JPDC)* 1986; **3**(2):215–235.
46. Wang Z, TW O, Sha EHM. Optimal loop scheduling for hiding memory latency based on two-level partitioning and prefetching. *IEEE Transactions on Signal Processing (TSP)* 2001; **49**(11):2853–2864.
47. Chen F, Tongsima S, Sha EHM. Loop scheduling algorithm for timing and memory operation minimization with register constraint. *Proceedings of SiPs'98*, Cambridge, MA, USA, 1998; 579–588.
48. Govindarajan R, Altman ER, Gao GR. Minimal register requirements under resource-constrained software pipelining. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, California, USA, 1994; 85–94.
49. Rong H, Douillet A, Gao GR. Register allocation for software pipelined multi-dimensional loops. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, USA, 2005; 154–167.
50. Zhang Y, Hu XS, Chen DZ. Efficient global register allocation for minimizing energy consumption. *ACM SIGPLAN Notices* 2002; **37**(4):42–53.
51. Davidson JW, Jinturkar S. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. *The 28th Annual International Symposium on Microarchitecture*, Ann Arbor, MI, 1995; 125–132.
52. Callahan D, Carr S, Kennedy K. Improving register allocation for subscripted variables. *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York, USA, 1990; 53–65.
53. Allen R, Kennedy K. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann: San Francisco, CA, 2001.
54. SMuchnick S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann: San Francisco, CA, 1997.
55. Wang M, Shao Z, Xue J. On reducing hidden redundant memory accesses for DSP applications. *IEEE Transactions Very Large Scale Integration Systems* 2011; **19**(6):997–1010.
56. Aho AV, Lam MS, Sethi R, Ullman JD. *Compilers: Principles, Techniques, and Tools*, 2nd edn. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2006.
57. Nykanen M, Ukkonen E. The exact path length problem. *Journal of Algorithms* 2002; **42**(1):41–53.
58. Rau BR. Iterative modulo scheduling: an algorithm for software pipeling loops. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, California, USA, 1994; 63–74.