A Framework for Parallelizing Load/Stores on Embedded Processors

Xiaotong Zhuang
Georgia Institute of Technology
College of Computing
801 Atlantic Drive
Atlanta, GA, 30332-0280
xt2000@cc.gatech.edu

Santosh Pande
Georgia Institute of Technology
College of Computing
801 Atlantic Drive
Atlanta, GA, 30332-0280
santosh@cc.gatech.edu

John S. Greenland Jr.
Green Hills Software, Inc.
Advanced products
30 W. Sola St.
Santa Barbara, CA 93101
jsg@ghs.com

Abstract

Many modern embedded processors (esp. DSPs) support partitioned memory banks (also called X-Y memory or dual bank memory) along with parallel load/store instructions to achieve code density and/or performance. In order to effectively utilize the parallel load/store instructions, the compiler must partition the memory resident values into X or Y bank. This paper gives a post-register allocation solution to merge the generated load/store instructions into their parallel counter-parts. Simultaneously, our framework performs allocation of values to X or Y memory banks.

We first remove as many load/stores and register-register moves through an excellent iterated coalescing based register allocator by Appel and George[14]. We then attempt to maximally parallelize the generated load/stores using a multi-pass approach with minimal growth in terms of memory requirements. The first phase of our approach attempts the merger of load stores without replication of values in memory. We model this problem in terms of a graph coloring problem in which each value is colored X or Y. We then construct a Motion Scheduling Graph (MSG) based on the range of motion for each load/store instruction. MSG reflects potential instructions which could be merged. We propose a notion of pseudo-fixed boundaries so that the load/store movement is minimally affected by register dependencies. We prove that the coloring problem for MSG is NP-complete. We then propose a heuristic solution, which minimally replicates load/stores on different control flow paths if necessary. Finally, the remaining load/stores are tackled by register rematerialization and local conflicts are eliminated. Registers are re-assigned to create motion ranges if opportunities are found for merger which are hindered by local assignment of registers. We show that our framework results in parallelization of a large number of load/stores without much growth in data and code segments.

1. Introduction

Advances in hardware design have greatly speeded up the microprocessors, however, low memory bandwidth and high latency remain major bottlenecks for modern systems. This is due to the relatively slow progress in memory design compared to the CPU design. Techniques to bridge this gap such as larger, faster, multi-level caches, speculative loads, pre-fetching attempt to keep the processors busy.

Recently, some processor designers (esp. in the DSP area) have developed special microarchitectural features and instructions to improve the effective bandwidth and memory access speed. These instructions can access memory faster by performing the loads and stores in parallel on partitioned memory banks using parallel data and address buses. Designers for embedded DSP chips or network processors prefer such techniques over more complicated hardware mechanisms involving, pre-fetching or speculation to simplify processor design with low cost. Examples of processors which support partitioned memory architecture with parallel load/stores include the Motorola DSP56000 series, NEC 77016, SONY pDSP, Analog Devices ADSP-210x, Starcore SC140 processor core, etc. For example, in Sony pDSP processor, an instruction such as PLDXY r1, @a, r2, @b can load variables a and b from memory into registers r1 and r2 simultaneously. To achieve this, the compiler must place a and b into different memory banks. This is a different architectural solution than certain SIMD type of instructions which need a and b to be next to each other in memory as well as r1-r2 needs to be a register pair (and not arbitrary set of registers). In the above PLDXY instruction, r1, r2 can be any two registers and the only restriction on variables a and b is that they must be allocated in different memory partitions (called X-Y partitions commonly).

To solve the problem of maximally generating parallel load/stores instructions (such as PLDXY) many approaches are possible at different stages of compilation. In this work, we propose a post-pass solution. We attempt to maximally combine loads and stores to generate parallel load/store instructions by undertaking a memory placement of register values after code is generated. We undertake a post-pass solution to capture *all* the load/stores; on an embedded processor with limited number of registers, a number of load/stores result due to spill values which we need to capture after physical registers are allocated. We first remove as many spills and register-register moves as possible by using an excellent iterated register coalescing allocator due to Appel and George[14] and then invoke our phase. Generation of parallel load/stores could be driven either by a desire to improve code density and reduce code size or by a desire to improve performance. In our case, we take a view of the problem that attempts to maximally improve performance by minimally increasing the sizes of data and code segments. In order to improve the effectiveness of our post-pass approach, our framework undertakes the following: First, in order to circumvent the limitation on motion of load/stores involved in any post-pass approach due to register dependencies we introduce a concept of pseudo-fixed boundaries which maximize the range of motion. We also perform local register re-materialization to remove certain limitations on motion to enable better motion. Secondly, in order to minimize the growth of data and code segments, we undertake a systematic approach. We first merge load/stores without any instruction replication; only the remaining load/stores are then considered for replication without replicating any value (ie, each value remains assigned only to X or Y memory but not both). For the remaining load/stores then we iteratively perform value replication followed by instruction replication stopping when no more load/stores can be merged. We thus pay for code and data segment growth on a demand driven basis. Our results show the code-size increase is within tolerable range.

Our solution also deals with irregularities of the DSP processors. The Instruction Set Architecture of our target chip—the SONY pDSP is non-orthogonal, which means only certain registers could be used within certain instructions. We summarize them as the following 3 ISA restrictions:

- Only register d0 to d3 can be used in the parallel load/store instructions. Other registers can not be used due to encoding constraints (only 2 bits are available for designating a register). This prompts us to perform re-assignment at the point of generation of parallel load/store. We use re-materilization as illustrated later to free the desired register.
- Registers in a parallel load/store instruction cannot be the same.
- Only PLDST (load/store), PSTLD (Store/Load), PLDLD(Load/Load) are available. No PSTST (Store/Store) is avalailable.

This paper is organized as follows. Section 2 describes the overview of the approach, section 3 presents the details of the framework. In section 4 we show performance evaluation results. Section 5 talks about related work, and section 6 concludes our paper.

2. Overview

Our framework for maximally parallelizing load/stores with minimal data and code segment growth consists of multiple phases. Figure 1 shows the overall approach.

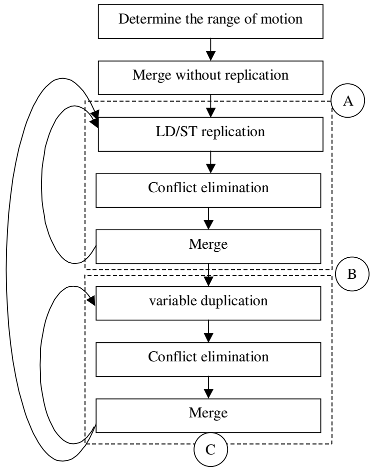


Figure 1. Post-pass phases

We first determine the range of motion of each load/store instructions and determine movable boundaries. We then attempt to parallelize the load/stores without any value or instruction duplication. The third stage attempts to perform duplication only when there is a benefit – that it examines the remaining load/stores and determines those which when duplicated could be combined. After a load/store is duplicated, first conflicts are resolved and then a merger is attempted. To capture the secondary effects of load/store duplication, we iterate until no more opportunities left. In the last stage, we merge the remaining load/store through the duplication of variables. Variable duplication inserts extra stores to put variables in both banks, however it creates new opportunities to merge the loads since the variable is available from both banks. In an iterative manner, such opportunities are exhausted. Finally, value duplication might open opportunities for instruction duplication. We then go back and do instruction duplication and repeat until convergence. The phase ordering of first attempting instruction duplication and then value duplication is motivated by the fact that instruction duplication leads to growth only in code segment as compared to value duplication which leads to growth in both data and code segments and is more expensive.

2.1 Assumptions

We make the following assumptions before the analysis of the problem.

- This is a post-pass approach and captures full spills. However, it does not have access to higher level information about arrays subscripts and therefore can not do intra-array data layout. Such tradeoffs are quite common and in fact a post-pass approach to capture all the spills was in fact also followed in [19]. We use one of the best allocators to remove as much spill code as possible first and then optimize the rest.
- Our goal is to minimize execution time, even at the expense of larger code size. We carefully control code growth by first doing instruction duplication and then value duplication.
- We account for ISA constraints due to encoding of parallel load/stores specializing our framework to a

particular ISA during coloring pass of the framework. These ISA restrictions are discussed earlier.

- We perform cloning of functions followed by inlining for allowing more opportunities to separately optimize each call. Currently we only handle non-recursive calls.
- Our framework is global (intra-procedural) and is not just limited to basic blocks unlike [20].

2.2 Basic concepts

2.2.1 Classification of memory access instructions

If we identify the memory location as a variable, then the Store instruction identifies a definition of the variable (memory location) and the Load instruction identifies a use of the variable (memory location). For example:

ST [addr], r identifies the definition of a memory address pointed by register addr.

LD [main.x], r identifies the use of a memory address at main.x, which is an immediate operand.

We classify the memory access operations into two categories: when only one operand is used in the address expression, we call it *Simple load/store* or *Type I load/store*, like LD [main.x], r and ST [addr], r. In the latter case, alias (actual symbol) analysis will be performed in an attempt to determine the actual symbol addr points to.

Base-Offset load/store or Type II load/store is written as load/store [r_{base}+r_{offset}], r. The token r_{base} and r_{offset} can be either immediate operand or register operand, but at least one of them should be register operand. So, Type II load/store instructions contain at least 2 register operands. Normally, it is used to access an array or a structure.

DSP instruction sets only contain very limited number of memory addressing modes. For our experiments on SONY pDSP processor, all the memory operations are either Type I or Type II load/store. Our framework can also be applied to other type of load/store instructions as long as the dependencies between registers are considered.

2.2.2 Dependencies

There are two possible dependence conflicts for load/store instructions.

Address Conflicts or Addr Conflicts. Assuming the memory location is a variable, then LD instructions (use of the variable) can not be moved before the ST instructions (the definition of the variable). Address Conflicts only happen among memory access instructions targeting the same memory location.

Register Conflicts or r Conflicts. The dependencies for each register operands in the instruction should not be violated. Register conflicts can happen between memory access instructions and non-memory access instructions.

For example, in Figure 2, the LD instruction cannot be moved beyond the ST instruction due to the Address Conflict. It also cannot be moved below x=r1 due to the Register Conflict.

```
1 ST [addr1],  r2
2
3 LD [addr1],  r1
4 x=r1
```

Figure 2. Two types of conflicts.

We introduce a notion of movable boundaries to make motion ranges unconstrained as explained in section 5.

2.2.3 Webs and Value Separation

In order to separate memory references which can be independently considered for allocation purposes, we use a concept of web of definitions and uses. *web* [9] is defined as the maximal union of du-chains such that, for each definition d and use u, either u is in the du-chain of d or there exist d=d_0,u_0…..,d_n,u_n=u, such that, for each i, u_i is in the du-chains of both d_i and d_{i+1}. Each web builds up a separate *symbol group*, i.e one must bind all the definitions and uses within a web to a single memory location. In this manner, we are able to achieve effective value separation at different program points even for a given temporary. In short, variables in different symbol groups can be put in different banks—they are treated as different values or variables.

Webs are assumed to be inseparable. Splitting a web could intuitively remove the restriction that variables on the web must be put into the same bank; however, additional load/store must be inserted in the join points of the web so that the contents of different parts of the web can be consistent. Web splitting may be profitable when the combined load/stores outnumber newly inserted load/stores. However, from our observation as illustrated in section 8, the size of the web (number of nodes on the web) is generally too small to meet this requirement and splitting only degrades performance.

2.2.4 Motion range

Motion range is defined as the interval between program points where a load/store instruction can be legally moved. A load/store instruction can move up and down obeying dependencies. In our framework, non-load/store instructions are assumed to be unmovable, while load/store instructions can be moved. When non-load/store instruction defines the boundary of motion range, it defines a fixed boundary which is not movable. Difficulty comes when the boundary is another load/store instruction. Since this kind of boundary will be moving too, we call it a *Movable Boundary*, in contrast to the boundary of non-load/store instruction, which is assumed to be stationary. Special techniques are introduced in section 3 to solve the *Movable Boundary Problem.*

2.2.5 Motion Schedule Graph (MSG)

After the determination of motion range, a MSG (Motion Schedule Graph) is built. It is an undirected graph with special meanings for nodes and edges. MSG is defines as follows:

- Each load/store instance is a node of the MSG.
- An edge between two nodes tells us that they can be possibly combined, i.e. they have overlapped motion ranges where they could be moved and merged.
- load/stores (nodes) that belong to the same web form a symbol group which is our unit of memory placement (i.e., each symbol group is allocated either in X or Y memory by our algorithm).
- Each symbol group is designated by a symbol node. A symbol node can only be colored with two colors, which is equivalent to the bank assignment of the memory location associated with it.

- All the loads and stores within a given symbol node must take place from the assigned memory bank and location. load/store nodes in the same symbol group assume the same color given to a symbol node.

Solving the MSG graph is to decide which edges should be included for combining load/stores and to determine coloring of the corresponding symbol node. Edge pruning is done to decide on which load/stores should be combined (please note that a given load could be potentially combined with more than one other load or a store but eventually has to be combined only with one – edge pruning allows us to remove multiple edges incident on a given load or store leaving only one which will be used for combining it). Note that, all the edges left should be disjoint to each other and two nodes on each edge should have different colors, which means they are in different banks. These are the restrictions which define the problem of combining load/stores and of placement of values in memory banks.

The graph algorithm should thus give out a scheme of node coloring and edge selection and try to maximize the number of load/store pairs. The algorithm will also attempt to replicate load/store instructions and/or underlying values to maximize parallelization under the constraint of minimal code and data segment growth.

3. Framework

An outline of our framework is as follows.

```
Input: the flowgraph of the generated code
Output: the flowgraph with merged LD/ST instructions.
Algorithm:
Load_store_identification();
Alias_analysis();
Build_webs();
Range_determination();
Graph_construction();
Graph_solving();
Conflict_resolution();

While(have_changes){
    While (Ldst_duplication_profitable()){
            Duplicate_Ldst();
            Find_and_remove_conflicts();
            Merge_ldst();
    }
    While (Var_duplication_profitable()){
            Duplicate_var();
            Find_and_remove_conflicts();
            Merge_ldst();
    }
}
```

Figure 3. Top-level procedures of the algorithm.

We now describe these phases in details.

3.1 Alias Analysis

We first identify load/stores and then perform alias analysis to determine the association of a load/store with memory location(s). If a load/store is aliased with multiple memory locations, then we exclude the given load/store from potentially combining it with other load/store. This is done due to the following reason. If a multiply aliased load/store were to be included, all the aliased memory locations must be allocated to the same bank which would eliminate any opportunities to optimize their load/stores through XY placement. This is too much of a penalty over excluding that specific load/store. We must however determine load/store aliases to check if any other load/stores can be moved across. For determining aliases of load/stores we use a simple algorithm from [8].

3.2 Building webs

Once we select the load/stores to be included in our analysis, we build webs to determine the value separation. As pointed out earlier, the value separation gives us more control over placement of values in memory than working on the variables themselves. This is especially true for temporaries which are recycled. Each web is found by undertaking a transitive closure of du/ud chains [9]. One of the important properties of the webs is that any motion of load/stores within a web does not have any impact on the load/stores belonging to another web. In other words, the web provides a safe boundary for motion of load/stores within which obviates the need for any analysis during motion.

3.3 Range Determination

Once webs are built, we determine the motion range for all load/store instructions. If two load/store have overlapped motion ranges, they will be linked by an edge in the graph which means they can be combined. However, here we are faced with an issue of fixed versus moving boundaries – motion of a load/store could impact motion of others. Here is an example.

```
1 MOV r1, 3                 1 MOV r1, 3
2 MOV r2, 2                 2 MOV r2, 2
3 ST [addr2], r1  (1)       3
4 ST [addr1], r2  (2)       4
5                           5 LD [addr1], r1  (3)
6                           6 ST [addr2], r1  (1)
7                           7 ST [addr1], r2  (2)
8 LD [addr1], r1  (3)       8
9 MOV r1, 0                 9 MOV r1,0
         A                           B
```

Figure 4. The movable boundary problem.

In the figure above, instruction (1) has boundary of <2,7> which signifies that the instruction can not be moved before label 2 (register conflict) and can not be moved beyond label 7 (register conflict). Here label 2 is a fixed boundary since instruction at label 1 is not a load/store and optimizer will not attempt to move it; however, instruction at label 8 is a load and might be moved by the optimizer. Thus, the boundary at label 7 is a moving boundary. Similarly, instruction (2) has boundaries <3,7> of which boundary at label 3 is fixed (register conflict due to instruction at label 2) and boundary at label 7 is moving (address conflict due to instruction at label 8). Instruction (3) has boundary <5,8> in which boundary at label 5 is movable (address conflict) and one at label 8 is fixed (register conflict). After motion of load/stores, in Fig.4.B, the fixed boundaries remain unmodified but movable boundaries could assume different values and thus one must update boundaries and ranges for load/stores else illegal code (such as shown in Fig.4.B) might result. However, updating boundaries and edges might lead to non-convergence of the algorithm. In our case, we solve the problem as explained in the following section.

3.3.1 Simple load/store (Type I)

Consider the Type I load/store as explained in section 2.3. We move all the Store instructions to the earliest places (boundaries) without conflicts. Similarly, all Load instructions are moved to the latest places without conflicts. Our first step is to convert the moving boundary problem to a pseudo fixed-boundary problem with constraints, so it can be solved as a fixed boundary problem. Then we try to resolve the conflicts after graph coloring. Noticeably, due to the constraints we put on the graph, the solution we get from the graph is always feasible and close to ideal solution of the problem. We show in section 8 that the constraints only causes a loss of very few merges.

We now define pseudo fixed boundaries for both loads and stores as follows.

Pseudo fixed-boundary for store (ST)

We move a store (ST) as early as possible assuming other instructions to be fixed. The pseudo fixed upper boundary is the current program point of the store (ST) which has been moved earliest. The stores (STs) can therefore be only moved down until we run into a conflict.

Pseudo fixed-boundary for load (LD)

We move a load (LD) as late as possible assuming other instructions to be fixed. The pseudo fixed upper boundary is the current program point of the load (LD) which has been moved latest. The loads (LDs) can therefore be only moved up until we run into a conflict.

These pseudo boundaries only put constraints on motion of load/stores. Note that, in this step, only the motion range is determined. No actual move of the load/store happens. MSG is constructed after the motion ranges are determined for all load/stores. As discussed earlier, in a MSG, each load/store instruction is represented by a node and nodes in the same symbol group (accessing the same memory location) are encompassed as a group. The graph below shows a typical MSG.

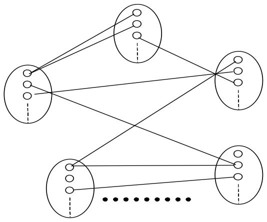


Figure 5. A typical MSG.

Although pseudo fixed boundaries decide the range of motion, they can not preclude certain types of conflicts which must be resolved. In other words, motion within these boundaries could lead to conflicts as discussed below.

In the following graphs, the notation $\boxed{\text{addr D}}$ denotes the definition of address addr. Similarly, the $\boxed{\text{addr U}}$ denotes the use of address addr. The notation $\boxed{\text{r D}}$ denotes the definition of register r and $\boxed{\text{r U}}$ denotes the use of register r.

Due to the motion of load/stores, sometimes address conflicts can occur. The address conflicts are always resolvable as shown below.

Figure 6.A shows an address conflict where the use of Addr1 is moved above the definition of Addr1, so it could be combined with Addr2. The definition of Addr1 is supposed to be combined with Addr3. This results in use of Addr1 before the definition of Addr1, violating the semantics. However, as shown in figure 6.B, one could always exchange the Addr2 and Addr3 parts and combine Addr1 D with Addr2 and Addr1 U with Addr3.

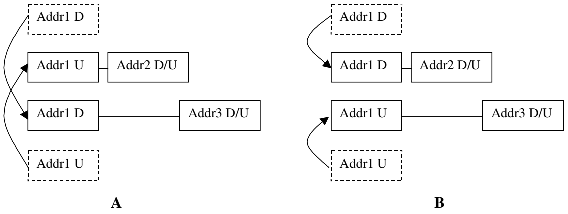


Figure 6. Address conflicts and resolution.

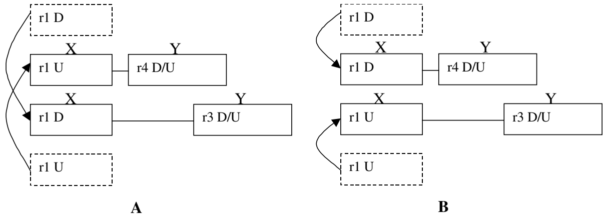


Figure 7. Register conflicts and resolution.

Some other problems might happen due to register conflicts. There are two cases for register conflicts.

Figure 7 illustrates a kind of register conflict, if the memory bank of both instructions using r1 happen to be X. We can resolve the conflict in a way similar to the address conflict earlier, by exchanging the position of the two instructions.

We now discuss another case of register conflicts. In this case (Figure 8), due to the attempt to combine r1 U and r4 D/U, r1 D and r3 D/U, r1 D might be moved before r1 U causing the conflict. However, in this case, we cannot combine r1 D with r4 D/U, because they are in the same bank. Same is the case of r1 U and r3 D/U. Thus, we cannot resolve the conflict as in the first case. Similarly, we cannot combine r1 U and r1 D. The only option in this case is to combine r4 D/U and R3 D/U and this is possible only if they have overlapping motion ranges. If this doesn’t hold, we have to give up the merge. r1 U and r1 D always share part of the motion range, however, they cannot be merged because of the same register (r1)used. We first try to get a free register at the point we want to merge r1 U and r1 D so that one of the r1 can be replaced by the free register and then we do a copy to return the value to r1. In case of no free register is available, our algorithm will check all the live registers to find one that can be rematerialized. We temporally use this register and reconstruct its contents after the parallel load/store instruction.

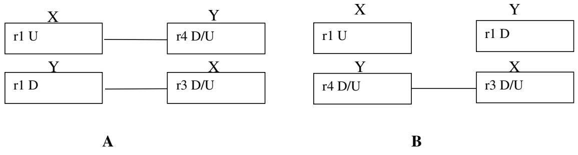
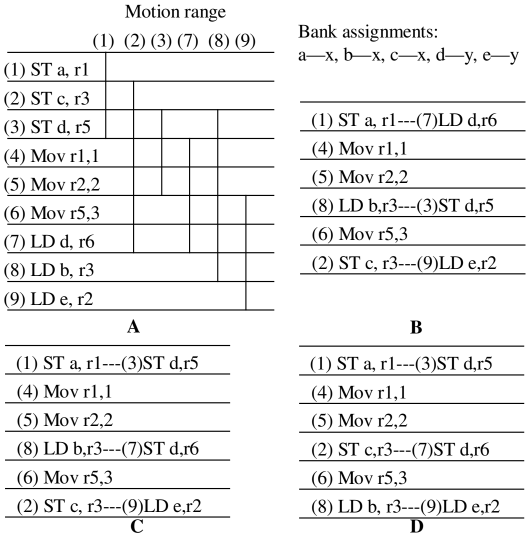


Figure 8. Another kind of register conflict

Our experiments show the address resolution happens rarely and almost all of them can be resolved easily.

We give an example as follows to illustrate the address and register resolution.



```
A
(1) ST a, r1
(2) ST c, r3
(3) ST d, r5
(4) Mov r1,1
(5) Mov r2,2
(6) Mov r5,3
(7) LD d, r6
(8) LD b, r3
(9) LD e, r2

B
(1) ST a, r1---(7)LD d,r6
(4) Mov r1,1
(5) Mov r2,2
(8) LD b,r3---(3)ST d,r5
(6) Mov r5,3
(2) ST c, r3---(9)LD e,r2

C
(1) ST a, r1---(3)ST d,r5
(4) Mov r1,1
(5) Mov r2,2
(8) LD b,r3---(7)ST d,r6
(6) Mov r5,3
(2) ST c, r3---(9)LD e,r2

D
(1) ST a, r1---(3)ST d,r5
(4) Mov r1,1
(5) Mov r2,2
(2) ST c,r3---(7)ST d,r6
(6) Mov r5,3
(8) LD b, r3---(9)LD e,r2
```

Figure 9. Example for address/register resolution.

The example shows a 9-instruction assembly code segment. In figure 9.A, we list the motion range of each memory instruction. As noted earlier the motion ranges put constraints on motion and thus obeying these constraints we could move and merge the instructions as in figure 9.B without any violation of the motion ranges of the instructions. However, due to the movable boundary problem, instruction (7) has been moved above instruction (3), which is an address conflict and instruction (8) has been moved above instruction (2), which is a register conflict. In figure 9.C, the address conflict is resolved by exchanging instruction (3) and (7). In figure 9.D, the register conflict is resolved by exchanging instruction (2) and (8). The resulting code is now legal.

3.3.2 Base-offset registers

Base-offset load/store instructions have more registers, however, we only need to consider more register conflicts with the same approaches applicable as stated in section 3.3.1.

3.3.3 Solving the two-coloring problem on MSG

After the previous procedures, we now have MSG ready for coloring. To solve the problem, both the colors (bank) of each symbol group and the maximal node pairs should be decided. In other words, our goal is to find the maximal node pairs by assigning proper colors to each symbol group. Note that, after assigning color to each symbol group (node), we must give out a schedule of which edges should be used for merging.

We have proved the MSG solving is a NP-complete problem. Please refer to Appendix A for the full proof.

3.3.4 Heuristic Solution

To get an approximate solution, only heuristic algorithms can be used. Several different approaches will be considered in this section.

First, we want to show that, if bank assignments are confined to the symbol groups without splitting, the optimal solution of finding maximal number of pairs can be obtained in polynomial time. Suppose that, there are N symbol groups. Some symbol groups are assigned memory bank X and others are in memory bank Y.

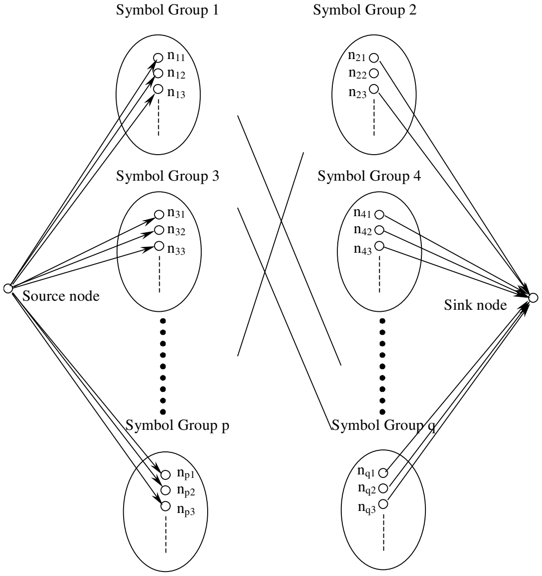


Figure 10. Convert MSG to maximal flow graph after bank assignment.

We prove that the problem of selecting maximal number of combinable edges given the colored graph is polynomial time solvable due to its reduction to the maximal flow problem as shown below.

The solution can be found by connecting each node in a symbol group assigned X bank to the source node and each node in a symbol group assigned Y bank to the sink node. An original edge remains if it connects two nodes in different banks, otherwise the edge will be deleted. Besides, all the edges in the graph are given a capacity of 1. An example graph is shown in figure 10. After determining the maximal flow on the graph, part of the edges will be occupied. We can see the edges in the middle that have flow are disjoint, because the nodes on the left only have one input edge and the nodes on the right only have one output edge. On the other hand, the overall flow is maximal in a sense that the number of pairs is maximized.

The complexity of solving the maximal flow problem is $|V|^3$[7], however, as a special case, this problem can be solved in $|V|^2$ time, where $|V|$ is the number of nodes in the graph.

Obviously, a $O(|V|^2 2^n)$ exhaustive searching algorithm can get the optimal solution, where n is the number of symbol groups (we do the coloring on n symbol groups first and then do the maximal flow on the colored graph, which leads to this complexity. Compared to [16], our complexity is greatly reduced. Actually, for all the DSPstone programs, the exhaustive searching can finish the compilation within an affordable time period. However, it fails for large-sized programs prompting heuristic solutions.

Our implementation picks simulated annealing algorithm [6] to find the optimal bank assignments. It has been used in various combinatorial optimization problems. Normally, the longer time it runs, the better solution will come out. In the worst case, finding the optimal value requires the same amount of time as the exhaustive searching.

The heuristic algorithm searches the solution by gradually adding the nodes to two determined sets (for X bank and Y bank). During each step, we pick the symbol group with maximal connectivity for the purposes of coloring and therefore uses a greedy strategy to do coloring. This algorithm is quite fast. Figure 11 gives the complete algorithm. In the results section we show that our heuristic approach gives a comparable solution to exhaustive approach with far superior compilation times. Moreoever, by performing some of the optimizations of code and data duplication discussed below, we are able to outperform the exhaustive approach which do not perform duplication in some cases.

```
Denote E as the set of all symbol groups
Let Set DETX={$S_x$} where $S_x$ is the symbol group with all the nodes
that must be put in X bank memory.
Let Set DETY={$S_y$} where $S_y$ is the symbol group with all the nodes
that must be put in Y bank memory.
Define the Cij as the number of edges between two symbol groups

While (E<>DETX ∪ DETY) do
   For each $S_k \in$ E-DETX ∪ DETY
         Calculate $CX_k = \sum_{S_i \in DETX} C_{ki}$ and $CY_k = \sum_{S_i \in DETX} C_{ki}$
   End for
   CX_max=max( $CX_k$| $S_k \in$ E-DETX ∪ DETY ),
   CY_max=max( $CY_k$| $S_k \in$ E-DETX ∪ DETY),

   If (CX_max>CY_max)
         Add the symbol group with CX_max to DETY
   Else
         Add the symbol group with CY_max to DETX
   Endif
End while
```

Figure 11. Heuristic algorithm to determine the bank for each symbol group.

3.4 Merge with duplication

Duplicating load/store across basic blocks can create new opportunities of mergers. The load/stores should only be moved to the successors/predecessors, where the reference is live.

We further require that: (1) The motion range of ST instruction is the EBB (Extended Basic Block) rooted from the original basic block. (2) The motion range of LD instruction is the reverse EBB rooted from the original place.

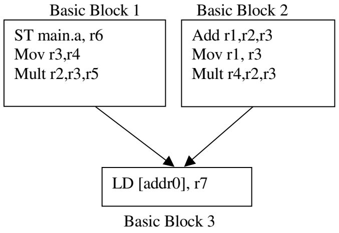


Figure 12. Merge with duplication.

Figure 12 shows that if the ST main.a, r6 instruction is moved from Basic Block 1 to Basic Block 3, it will be unnecessarily executed for the control flow going from Basic Block 2 to Basic Block 3. Even if no side-effect occurs after this kind of moving, control flow may frequently go from Basic Block 2 to Basic Block 3. So, it's potentially not profitable to push ST outside EBB. Similarly, LD instructions should only be moved within the reverse EBB.

To assure the motion is profitable, we make sure that for each uncombined load/store, if pushed to at least one of its live predecessors/successors (inside the EBB/reverse EBB) allows it to be combined with some other instructions. In other words, at least on one of the execution paths, the program executes faster, while no slowdown occurs on the other paths. In the above figure, we may choose to move the LD [addr0],r7 instruction upward to basic block 1 and also duplicate it at the end of basic block 2.

3.5 Merge with variable duplication

Variable duplication is done to intentionally store some variables into both memory banks so the LD instructions of these variables can be combined with other load/store instruction regardless their bank assignments. We must notice, the duplication is not always profitable, because additional ST instructions must be added to every ST on the web. Hopefully, the extra ST instructions can be combined with other un-combined instructions to reduce the cost. Our algorithm runs a profitability determination procedure to decide whether the variable duplication should be performed.

As a simple example, we illustrate how the variable duplication is done in the following figure.

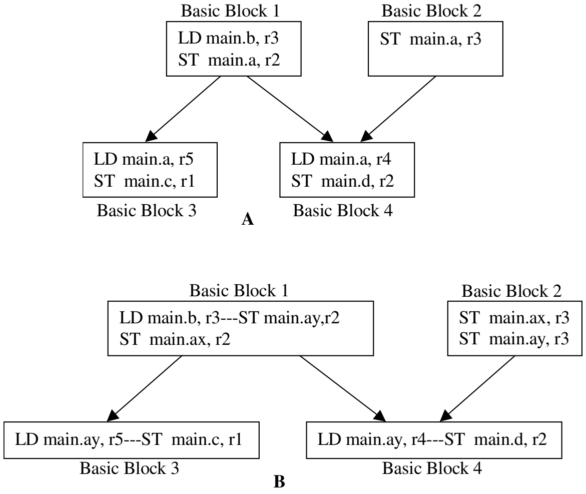


Figure 13. Variable duplication.

In figure 13.a, if all variables are in bank X after the previous procedures, we will have no zero merger opportunities for load/stores. Variable duplication will look at the web of variable main.a. We can store main.a to bank Y as well as to bank X (we should still keep the copy of main.a in bank X, since it is preferred by the previous optimizations). In this figure, we have two store instructions of main.a, so we will add two store instructions which store main.a to a place in Y bank memory. For distinction, we write main.ax and main.ay as the two memory locations for the variable main.a. This duplication is profitable, because the two instructions in basic block 3 and 4 can now be merged. Although we must introduce two stores in basic block 1 and 2, one of them can be merged as well. Figure 13.b shows the resulting flow graph.

3.6 Local conflict elimination

If the register allocator tends to assign the same register to the neighboring disjoint ranges, it could lead to a lot of register conflicts hampering motion. To eliminate such conflicts locally, we find those instructions with the conflicting registers and check if they are rematerializable. When a register can be freed and reconstructed after the merge, the instruction is removed to increase motion ranges and therefore more chances are created for the load/store instructions around it. The following graph shows a simple example doing local conflicts elimination.

```
Add r3, r1, r4
ST main.a, r2
Mov r2, r3
Mov r3, r5
LD main.b, r5
```

```

Mov r3, r5
LD main.b, r5-- ST main.a, r2
Add r2, r1, r4
```

Figure 14. Local conflict elimination.

The original code is listed on the left-side of figure 14. The ST and LD instructions cannot be merged due to the blocking Mov instructions. However, the value of r2 can be rematerialized by adding r1 and r4 again. On the right side, the assignment to r2 is delayed to the end so that the two load/store instructions can be combined.

3.7 Global optimizations

We now discuss some enabling compiler optimizations to increase the effectiveness of our approach.

3.8 Global and fixed-bank variables

For global variables, their banks can only be determined once. After the bank is decided in one procedure, other procedures will treat it as a fixed-bank variable. There are other reasons we encounter fixed-bank variables, like pointers passed as arguments, where the pointed data have already been put in a memory bank.

Fixed-bank variables are easily incorporated into our algorithm. In MSG, two special symbol groups are created. One contains all the fixed-X-bank variables and the other contains all the fixed-Y-bank variables and are excluded from reassignment. The other parts of the algorithm remain unchanged.

3.9 Global optimization through procedure cloning

Our implementation does global optimization by cloning procedures called by other procedures. When the parameters passed to the callee are different for several callers, different copies are created for each caller. This allows more opportunities for combining load/stores for cloned copies of procedures for different parameters passed at different call sites. To speed up the MSG generation, we also inline these procedure so that global optimization can be fulfilled at the price of code growth.

4. Performance evaluation

4.1 Implementation

The algorithm has been implemented with the SUIF/MACHSUIF compiler targeted to the SONY pDSP architecture. A separate pass called xymerge is added after the raga pass—a standard pass in machsuif which runs Appel & George's iterated register allocation algorithm [14]. Figure 15 shows the passes in Machsuif. The pass il2cfg changes the instruction list to control flow graph, then we do the register allocation and xymerge, the cfg is changed back to instruction list and finally translate to assembly code.

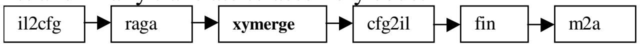


Figure 15. Processing pipeline.

The SONY pDSP has two banks of memories which are accessible through separate buses. Both X bus and Y bus are all 24 bit wide. Indirect memory access must go through address registers adr0 to adr7. As mentioned earlier, SONY pDSP has three parallel memory instructions, namely pldld, pldst, pstld. No pstst exists, so we cannot write to the two banks at the same time. Two parallel memory operations can work on different data register and arbitrary memory locations as long as they are in separate banks. The pDSP ISA also forbid the use of register d4 to d7 in parallel memory instructions due to the bit limit in those instructions. We will mark such kind of merge failure as "due to *ISA limitations*".

We have tried both simulated annealing (SA) and the other heuristic algorithms at point A (in figure1) to color the MSG. We also report the results at point B and C. Test programs are selected from DSPStone, Mediabench and SPEC2000 (bzip2 and vpr) benchmark. These benchmark programs contain lots of array operations; thus, a simple alias analysis pass is performed in order to determine the symbols associated with the memory locations pointed by the address registers.

Table 1 lists the results for 10 benchmark programs after the alias analysis pass. The first column shows the total number of load/store instructions in the program, which is after procedure cloning and inlining. We can see the register allocation pass does a very good job by keeping the number of load/stores small. For the SPEC vpr program, the number of memory operation instructions is about 3000. Column 2 to 5 lists the distribution of those load/store instructions. The address symbols are those actual variable symbols which need to be determined through alias analysis. Column 6 shows the determined symbols (the symbols which do not have multiple aliases) after alias analysis and column 7 shows the percentage of determined symbols over total number of load/store. On average, about 94 percent of load/store can be determined after alias analysis. In this version, we have implemented a flow-sensitive alias analysis pass [8] on the generated code. It works very effectively. Former experience with a poor alias analysis pass (that only determined 63% of load/stores) shows that the ratio of determined alias versus total load/stores has great impacts on the performance of the xymerge pass, because undecided load/store instruction not only reduce the merge-rate of compiler, but also creates obstacle for the motion of other load/store instructions since it is unsafe to move other load/stores across those instructions whose aliases are unknown.

Table 2 gives us some insights into where the merges happen and several reasons for the merge failure. The results are taken from point B. We classify the merges in two ways—without-duplication versus with-duplication and LDLD versus LDST. We can see most merges happen without duplication. The distribution of LDLD instruction and LDST instruction is justified by the fact that in most cases the LDs and STs are close to each other, so LDST should outnumber LDLD. All ST instructions must be combined in the form of LDST due to the lack of STST instruction. Column 5 to 6 list the reasons for failed merge. ISA failure is specific for the pDSP architecture due to the address register limitation in parallel memory instructions. Some of the failure happens when two ST instructions need to be combined. This kind of failure takes a big share among all the failures. It's understandable because STST instructions should occur as often as LDLD instructions on average. The last column shows the frequency of resolution invoked during the compilation. This number is very low comparing to the number of load/store and merged pairs.

Table 3 provides statistics for the MSG at point A. The group number is the number of symbol groups, and multiple nodes exist for each group. As shown by the edge number, the MSG seems to be very sparse given the large number of nodes in the graph. Interestingly, the number of edges is not quite related to the number of nodes in the graph, and it also varies greatly among different programs. Thus, largely, we can conclude MSG is application specific. We also notice that only a small amount of edges are finally selected to serve for the parallel load/store instructions. Column 4 is the average size of web in each program. They are typically small which justifies our conclusion that splitting the web is generally not profitable.

Table 4 lists the execution time for 9 of the benchmarks (due to lack of needed library, we are unable to link and execute the vpr benchmark) in different optimizing stages. The first column is the number of cycles to execute the un-modified program. Column 2 and 3 are the performance at point A with the heuristic and SA algorithm. Column 4 and 5 separately show the cycles at point B and point C. The speedup is in column 6. On average we achieve a 17% speedup over the original one ranging from 5% to 43%. In fact, in 4 out of 9 benchmarks we have a speedup over simulated annealing which gives optimal solution. In one case the performance is almost comparable. In other 4 cases, there's a slowdown (the positive numbers show speedup and the negative numbers show slowdown).

Table 5 shows the compilation time of the various methods. One can see that our method performs significantly faster. Speedups range from 8 to 163 times with an average of 44 for the case of only instruction replication. For the case of both value and instruction duplication, the speedups range from 5 to 103 times with an average value of 28.

Table 6 shows the growth in code and data segments. With only instruction replication, the code growth ranges from 1.5 to 14 percent with an average of 6.47 percent. In the case of duplication with value and instructions, the growth in the data and code segment ranges from 7.3 to 25.4 percent with an average of 14.2 percent. This shows that significant growth in data and code segment is incurred when value and instructions are both replicated. This justifies our phase ordering of attempting instruction replication first followed by instruction and value duplication.

We would like to compare our approach first with [17][18]. Their approach uses simulated annealing. As shown in the above discussion, we achieve a performance comparable or better than simulated annealing with far superior compilation times and with minimal code growth. This is achieved by our framework which systematically attempts to parallelize load/stores by enhancing range of motion followed by duplication of instructions followed by duplication of values.

[20] shows a speedup in the range of 13 to 49 percent on kernel benchmarks which are individual loops. Comparing to them, we achieve a performance gain from 5 percent to 43 percent on full benchmarks with an average gain of 17 percent. Moreover, their method attempts to undertake only a partial or full data replication without attempting instruction replication and suffers a rather large memory requirement growth. In our approach, one can see that the minimal code growth happens due to instruction replication and larger code growth happens through value replication. Therefore, an approach relying only on data replication suffers from the problem of memory requirement growth. In short, our approach of first attempting instruction replication and then data replication significantly boosts performance with a limited code growth.

5. Related work

The work on storage allocation to optimize the load/stores can be classified into SIMD type instructions which allow fetching adjacent memory values into register pair and into memory bank architectures which allow parallel load/stores from different memory banks.

[1] talks about the coalescence of narrow loads into wide loads by proper organization and alignment of the data. It works on loop unrolling--reordering the loop body. Safety and profitability are analyzed before memory access coalescence. They also introduce dynamic analyses to handle aliasing and alignment at runtime. Results show fairly good improvements on some of the Motorola chips.

[2] proposes a hardware solution to add compiler-controlled memory (CCM), which can be thought as the on-chip memories on some new DSP chips. Their focus is to spill value quickly to the on-chip memory area instead of cache. This architecture can improve predictability of the program behavior and reduce register pressure. Their compiler-controlled memory is a separate memory from the main memory. The CCM mechanism can be thought of providing a fast temporary store under compiler control.

The memory bank model due to X-Y memories offers an interesting architecture wherein restrictions of register pairs and layouts in adjacent memory locations do not exist. Thus, it allows more load/stores to be combined. There has been some work on doing compiler optimizations for such architectures. [18] discusses this problem and attempts to combine register allocation and bank assignment. They have built a rather complex model to incorporate many different constraints of register allocation and memory placement. The resulting model subsumes at least three NP-hard problems [18]. The compilation time of their approach is very high making it infeasible for a practical system. Their main goal is to reduce code size in contrast to ours which is to maximize the speed. We separate the register allocation and value placement phases so that we can optimize them separately giving us more control over optimizations for each of them. Combining register allocation with X-Y merge increases the complexity of problem. Moreover, it may not get good results compared to a post-pass approach, which benefits a lot from an excellent register allocator, which only generates very few spills. Especially, for DSP chip with limited number of registers, a substantial part of the load/store instructions are spill code. In our approach, we focus on eliminating as many as load/stores as possible through an excellent allocator built in Machsuif [14], then concentrate on placement of values. In contrast with them, we perform several optimizations such as increase in range of motion, conflict resolution and finally minimal code growth through instruction and value duplication.

R.Leuper and D.Kotte [17] proposed a two-pass approach. The first pass determines the exact set of memory access, then the IR is annotated with partitioning information and passed to the backend again. Variable partitioning is modeled as ILP based on an *interference graph*. However, they do not consider the movable boundaries for load/stores which un-constraints the range of motion. In any post-pass approach, the biggest constraint is the range of motion because of register dependencies. However, the post-pass solution is attractive because it captures the complete sets of spills. In our framework, we develop a resolution technique to relax the motion constraints as much as possible to maximize opportunities for combination of load/stores. Another difference is that they use ILP approach to solve their problem whereas in our case, we show that a greedy heuristic does a great job by reducing the problem to maximal flow. Their ILP solution is again quite slow to incorporate into a practical compiler. The greedy heuristic is extremely fast and can be built as a standard pass in any compiler. Moreover, it gives comparable quality of solution as the exhaustive methods. Also, they do not perform value or instruction duplication.

Saghir, Chow and Lee [20] present compaction based data partitioning and partial data duplication approaches. Theirs is also a post-pass approach. However, their approach is limited only to basic blocks. Unlike us, they do not explore instruction duplication since they are limited only to a basic block. In order to create more opportunities, they attempt partial or full duplication of data. We believe that load/store motion and duplication expose a lot more opportunities than value duplication. Also value duplication results in higher memory requirements for both code and data segments which is undesirable. That is why we phase order instruction motion and duplication before value duplication. We also capture the secondary effects of value duplication on instruction duplication and vice versa using an iterative approach. Moreover, we propose the notion of pseudo-fixed boundaries that allow a large range of motion to tackle the problem of register dependencies encountered in post-pass approaches. We do a conflict resolution to maintain legality after motion takes place without degrading the merged load/store pairs. Therefore our approach results in minimal code growth we believe (in their paper [20] raw performance or code growth numbers are absent to do any quantitative comparison).

Recently, J. Cho, Y. Paek, D. Whalley [21] study memory and register assignment for non-orthogonal architectures. They have used different approaches like MST and graph coloring to assign memory banks to variables. Also, a register class allocation phase is inserted to assign register class before the register allocation, so the register allocation phase can meet those requirements. Therefore, their main contribution is to formulate and solve the heterogeneity model for both register and memory assignments unlike ours in which we attempt iterative solution of first instruction replication followed by value replication which minimizes code growth while maximizing the parallel load/stores.

Other earlier work [16] on this problem adopts very simplistic approach of allocating X/Y memory on an alternating basis without any analysis.

6. Conclusion

This paper proposes a framework for analyzing load/stores stores and for moving them to combine them into parallel load stores maximally. An important contribution of this paper is to propose the notion of pseudo-fixed boundaries to enhance the range of motion and then perform conflict resolution to preserve legality of code while keeping maximal number of load/store pairs. We also undertake rematerialization to free registers to enhance boundary of motion. We minimally first replicate the instructions and then values and iterate until no profitable mergers result. We show that by undertaking such an approach, our solution comes close to exhaustive one with much lesser compilation time. We use the freed time to perform optimizations which leads to solution better than an exhaustive brute force solution (without optimization) in almost half the benchmarks. The conclusion of this work is that by systematically enhancing the range of motion of instructions and by undertaking minimal replication of instructions and values along with their secondary effects, one can generate code quality comparable or superior to that generated by exhaustive methods previously proposed.

Table 1 Results of alias analysis

	Total LD/ ST	#LD	#ST	#var sym	#add sym	alias found (point to single reference)	Alias found/total load/store (%)
Biquad_N_sections	26	15	11	10	16	26	100
Complex_update	26	16	10	6	20	20	76.92
n_complex_update	67	26	41	35	32	53	79.10
n_real_updates	18	5	13	6	12	16	88.89
GSM Untoast	401	225	176	48	353	397	99.00
g721 decoder	191	113	78	66	125	191	100
rawcaudio(adpcm)	21	13	8	11	10	21	100
rawdaudio(adpcm)	26	15	11	17	9	26	100
SPEC2000-Bzip2	1432	840	592	424	1008	1398	97.63
SPEC2000-VPR	2927	1763	1164	903	2024	2872	98.12
Average							93.97

Table 2 Classification of generated parallel load/stores

	Merge type				Failed merge		
	Without dup	With dup	LDLD	LDST	ISA	STST	Re-solved
Biquad_N_sections	4	1	2	3	0	1	2
Complex_update	6	2	3	5	3	3	2
n_complex_updates	9	3	4	8	2	3	3
n_real_updates	0	1	0	1	0	2	0
GSM Untoast	73	14	32	55	4	10	9
g721 decoder	17	4	6	15	4	4	4
Rawcaudio (adpcm)	5	2	2	5	1	1	1
Rawdaudio (adpcm)	7	2	4	5	3	5	3
SPEC2000-Bzip2	177	43	78	142	10	13	12
SPEC2000-VPR	394	103	154	343	26	30	45

Table 3. MSG properties

	Grp #	Node #	Edge #	Avg. Size of web
Biquad_N_sections	8	26	12	3.25
Complex_update	11	26	27	2.363636
n_complex_updates	21	67	49	3.190476
N_real_updates	7	18	7	2.571429
GSM Untoast	129	401	486	3.108527
g721 decoder	32	191	130	5.96875
Rawcaudio(adpcm)	10	21	25	2.1
Rawdaudio(adpcm)	9	26	31	2.888889
SPEC2000-Bzip2	395	1432	1109	3.625316
SPEC2000-VPR	937	2927	3104	3.123799

Table 4. Comparison of execution time (10^4 cycles)

	Original	Heuristic(at A)	SA	LD/ST replication (at B)	Var replication (at C)	%Speedup (after var replication/Original)	%Speedup (after var replication/SA)
Biquad_N_sections	94	87.5	86.64	85.08	83.9	13.18432767	3.258081524
Complex_update	1.6	1.56	1.56	1.53	1.53	5.643817202	2.24142932
n_complex_updates	312	300.3	295.93	299.4	298.2	4.594244163	-0.771453874
N_real_updates	123.3	116.1	114.74	114.2	113.3	8.842528413	1.26705758
GSM Untoast	7652	5478	5149.9	5393	5346	43.12337734	-3.677966343
g721 decoder	5201	4138	3718.8	4137	4084	27.33031336	-8.958826183
Rawcaudio(adpcm)	10490	9052	9071.8	8937	8802	19.17685555	3.064144342
Rawdaudio(adpcm)	4677	4186	3961	4150	4114	13.69017833	-3.715787615
SPEC2000-Bzip2	20945	17893	16875	17578	17423	20.2098194	-3.146948583
Average						17.31060682	-1.160029981

Table 5. Comparison of compilation time (Seconds)

	Original	Heuristic(at A)	SA	LD/ST replication (at B)	Var replication (at C)	%Speedup (after LD/ST replication/SA)	%Speedup (after var replication/SA)
Biquad_N_sections	7.75	8.92	291.75	11.23	17.85	2497.474516	1534.454138
Complex_update	4.99	6.1	1125.99	6.85	10.75	16354.67326	10368.68129
n_complex_updates	10.1	12.3	511.09	14.73	21.60	3370.477952	2265.213625
N_real_updates	6.34	6.91	295.34	7.61	12.49	3781.307223	2264.920316
GSM Untoast	391	430	8717.78	556.03	902.50	1467.852987	865.96204
g721 decoder	250	295	4061.06	412.85	614.94	883.6588128	560.395309
Rawcaudio(adpcm)	15.8	18.3	674.75	21.42	35.36	3049.802212	1808.277119
Rawdaudio(adpcm)	11	14.6	1984.01	20.04	31.72	9800.269262	6155.699016
SPEC2000-Bzip2	1378.82	1745.4	32765.97	2234.98	3331.24	1366.048962	883.5954124
SPEC2000-VPR	2237.76	2690.81	57197.27	3157.40	4386.57	1711.532724	1203.918
Average						4428.309791	2791.112

Table 6. Comparison of code size (# of Instructions)

	Original	Heuristic(at A)	SA	LD/ST replication (at B)	Var replication (at C)	%Code growth (after LDST replication/Original)	%Code growth (after var replication/Original)
Biquad_N_sections	157	151	147	167	180	6.369426752	14.64968153
Complex_update	118	106	102	135	148	14.40677966	25.42372881
n_complex_updates	295	273	271	321	347	8.813559322	17.62711864
N_real_updates	181	179	179	197	209	8.839779006	15.46961326
GSM Untoast	3992	3843	3818	4053	4284	1.528056112	7.314629259
g721 decoder	1847	1813	1805	1909	2074	3.356794802	12.29020032
Rawcaudio(adpcm)	204	193	190	218	234	6.862745098	14.70588235
Rawdaudio(adpcm)	190	174	172	209	221	10	16.31578947
SPEC2000-Bzip2	12256	12102	12073	12452	13150	1.59921671	7.294386423
SPEC2000-VPR	46954	46280	46194	48338	52214	2.947565703	11.20245347
Average						6.472392317	14.22934835

REFERENCES

[1] J.W.Davidson, S.Jinturkar, "Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses", *Proc. PLDI '94*, pp. 186-195, June 1994.

[2] Keith D.Cooper, Timothy J.Harvey, "Compiler-Controlled Memory", *In 8th ASPLOS*, October 1998.

[3] T.H.Cormen, C.E.Leiserson, R.L.Rivest, *Introduction to algorithms*, MIT Press, 1989

[4] Ashish Bhalgat, John Greenland, Santosh Pande, "Instruction Scheduling to Hide Load/Store Latency In Irregular Architecture Embedded Processors", 3rd workshop on MP-DSP, Nov.2001.

[5] Robert Azencott , *Simulated annealing: parallelization techniques*, Braun-Brumfield,INC.

[6] E.Aarts,J.Korst, *Simulated annealing and Boltzmann Machines*, Courier Int'l.

[7] C.H.Papadimitriou, K.Steiglitz, *Combinatorial optimization Algorithms and Complexity*, Dover Publications INC, 1998.

[8] A.V.Aho, R.Sethi, J.D.Ullman, *Compilers Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986

[9] S.S.Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufman,1997

[10] Mach-SUIF Backend Complier, *The Machine-SUIF 2.1 compiler documentation set*. Harvard University, Sep. 2000,

[11] Stanford SUIF Compiler Infrastructure, *The SUIF 2 compiler documentation set*, Stanford University, Sep.2000. http://suif.stanford.edu/suif/index.html.

[12] Motorola INC, *DSP56300 24-bit Digital Signal Processor Family Manual*, Jan. 1995.

[13] A.W.Appel, L.George, "Optimal Spilling for CISC Machines with Few Registers", *Proc. PLDI'01*, pp. 243-253, June 2001.

[14] Lal George, Andrew W. Appel, "Iterated Register Coalescing", *Proc. SIGPLAN '96 Conf. Programming Language Design and Implementation.*

[15] A. Sudarsanam and S. Malik. "Memory Bank and Register Allocation in Software Synthesis for ASIPs", *In Proceedings of the International Conference on Computer Aided Design*, pages 388--392, 1995.

[16] B. Powell, E.A. Lee, and W.C. Newman. "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Di-agrams", *Proceedings International Conference on Acoustics, Speech, and Signal Processing*, 5:553 556, 1992

[17] Rainer Leupers, Daniel Kotte, "Variable partitioning for dual memory bank DSPs", *ICASSP*, May 2001.

[18] Ashok Sudarsanam, Sharad Malik, "Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs", *ACM Trans. On Design Automation of Electronic Systems*, pp.242-264, Vol. 5, No.2, Apr. 2000.

[19] K. D.Cooper, N.McIntosh, "Enhanced Code Compression for Embedded RISC Processors", *Proc. PLDI '99*, pp. 139-149, May 1999.

[20] M. A. R. Saghir, P. Chow, C. G. Lee, "Exploiting Dual Data-Memory Banks in Digital Signal Processors", *Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operation Systems*, pp. 234--243, 1996.

[21] J. Cho, Y. Paek, D. Whalley, "Register and Memory Assignment for Non-orthogonal Architectures via Graph Coloring and MST Algorithms", *Proc. of LCTES'02*, pp. 130-138, Jun. 2002.

APPENDIX A

THEOREM: MSG solving (find the XY assignment to the nodes and determine the maximal number of combinable edges) is NP-complete.

Proof: The optimization problem can be restated as a decision problem, i.e. asking whether it has k node pairs satisfying the conditions.

Firstly, this is a NP problem. Since, given a certificate, i.e. node pairs and color (bank) assignment for each symbol group, we can verify the following facts in polynomial time.

- Edges and nodes belong to the graph
- Two nodes on the edge have different color.
- The number of node pairs is greater than k.

Secondly, we reduce a well-known NP-complete problem–*Partition* to MSG coloring problem. "Partition" is a decision problem defined as:

Given objects with sizes $A_1,A_2,\ldots A_n$, asking whether they can be splitted into 2 disjoint sets S and T, such that $\sum_{i\in T} A_i = \sum_{i\in S} A_i$. We build our graph according to the given "Partition" problem. There are n symbol groups with node number $A_1,A_2\ldots.A_n$. All nodes are fully connected (all edges exist). Solving our problem optimally means we have found the $MIN\left(|\sum_{i\in T} A_i - \sum_{i\in S} A_i|\right)$. Remember that $\sum A_i$ is a fixed number, so $MIN\left(|\sum_{i\in T} A_i - \sum_{i\in S} A_i|\right)$ is equivalent to maximal number of combinable node-pairs. Thus, if this minimum equals 0, we give a positive answer to the "Partition" problem, otherwise, we give a negative answer. This reduction can be done in polynomial time, so we have proved that solving MSG is NP-complete. □