Parallelizing Load/Stores on Dual-Bank Memory Embedded Processors

XIAOTONG ZHUANG and SANTOSH PANDE
Georgia Institute of Technology

Many modern embedded processors such as DSPs support partitioned memory banks (also called X–Y memory or dual-bank memory) along with parallel load/store instructions to achieve higher code density and performance. In order to effectively utilize the parallel load/store instructions, the compiler must partition the memory-resident values and assign them to X or Y bank. This paper gives a postregister allocation solution to merge the generated load/store instructions into their parallel counterparts. Simultaneously, our framework performs allocation of values to X or Y memory banks. We first remove as many load/stores and register–register moves as possible through an excellent iterated coalescing based register allocator by Appel and George [1996]. We then attempt to parallelize the generated load/stores using a multipass approach. The basic phase of our approach attempts the merger of load/stores without duplication and web splitting. We model this problem as a graph-coloring problem in which each value is colored as either 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 that could be merged. We propose a notion of pseudofixed boundaries so that the load/store movement is less affected by register dependencies. We prove that the coloring problem for MSG is NP-complete and solve it with two different heuristic algorithms with different complexity. We then propose a two-level iterative process to attempt instruction duplication, variable duplication, web splitting, and local conflict elimination to effectively merge the remaining load/stores. Finally, we clean up some multiple-aliased load/stores. To improve the performance, we combine profiling information with each stage coupled with some modifications to the algorithm. We show that our framework results in parallelization of a large number of load/stores without much growth in data and code segments. The average speedup for our optimization pass reaches roughly 13% if no profile information is available and 17% with profile information. The average code and data segment growth is controlled within 13%.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*; B.1.4 [**Microprogram Design Aids**]: Languages and compilers; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms: Design, Performance

Additional Key Words and Phrases: DSP architectures, parallel load/stores, memory bank allocation, profile driven optimization

Authors' addresses: Xiaotong Zhuang and Santosh Pande, Georgia Institute of Technology, Atlanta, Georgia 30332.

1. INTRODUCTION

Advances in hardware design have greatly speeded up microprocessors. However, low memory bandwidth and high latency remain major bottlenecks for modern computer systems. This is because of the relatively slow progress in memory design compared to the CPU design. Techniques to bridge this gap have been widely studied. Many hardware/software solutions such as larger, faster, multilevel caches, speculative loads, and prefetching are developed to keep the processors busy. Meanwhile, the demand for more memory ports has been increased because of the need to parallelize simultaneous memory accesses. However, increasing the memory ports inevitably costs more space and power to be invested.

Recently, some processor designers (especially in the DSP area) have developed special microarchitectural features and instructions to improve the effective bandwidth and memory access speed without increasing the number of ports. These instructions can access memory faster by performing the loads and stores in parallel on partitioned memory banks (also called X–Y memory or dual-bank memory) using parallel data and address buses. Designers for embedded DSP chips or network processors prefer such techniques over more expensive alternatives, such as increasing memory ports to simplify processor design with low cost. Examples of processors that support dual-memory bank 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, the memory is partitioned into two banks, called X memory and Y memory. An instruction, such as PLDLD r1 $\leftarrow$ a, r2 $\leftarrow$ b can load variables a and b from memory into registers r1 and r2 simultaneously as long as a and b are in different banks. This is a different architectural solution than certain SIMD type of instructions which require a and b to be next to each other in memory; r1 and r2 needs to be a register pair (and not an arbitrary set of registers). In the above PLDLD 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.

Utilizing parallel load/stores could lead to more parallelism being exploited for memory accesses and faster execution speed. In this work, our main objective is to improve the performance (i.e., execution speed). We also demonstrate that through a series of optimizations, we can achieve this goal with small code growth and an affordable amount of compilation time. To better utilize parallel load/store instructions (such as PLDLD), many approaches are possible at different stages of compilation. In this work, we propose a postpass solution, in which we attempt to combine loads and stores to generate *parallel* load/stores. In other words, our postpass solution is motivated by the goal to capture all the spills, because on an embedded processor with limited number of registers, a significant part of load/stores are generated because of spill values, which can only be captured after physical registers are allocated. Before our pass, an excellent iterated register coalescing allocator by Appel and George [1996] is invoked to remove as many spills and register–register moves as possible. Next, we propose a framework with three stages. The baseline stage involves our basic

approaches to merge load/store instructions that are in the code without duplication and web splitting.[1] The second stage includes three optimization steps, i.e., instruction duplication, variable duplication, and web splitting, organized into two nested iterations. Finally, the third stage tackles multiple-aliased load/store instructions. We show incremental performance improvements with more optimization steps being added in the second stage. To further enhance the effectiveness of our approaches, profile information can be used to direct each optimization step.

Our solution deals with irregularities of the DSP processors as well. The Instruction Set Architecture of our target chip—the SONY pDSP—is heterogeneous, which means there are several limitations on the permissible register usage and instruction type. Our algorithms take these constraints into consideration.

This paper is organized as follows. Section 2 describes preliminaries. Section 3 gives an overview of the approaches. Section 4 presents baseline stage. Section 5 talks about the other two stages. Section 6 is about global optimizations. Section 7 deals with the profile-driven parallelizing extension. Section 8 shows evaluation results. Section 9 talks about related work and Section 10 is the conclusion.

2. PRELIMINARIES

This paper deals with compiler optimizations to utilize parallel load/store instructions on an embedded processor with dual-bank memory system. Our goal is to reduce execution time. Our approaches also attempt to control code growth and compilation time once the performance objective has been achieved.

2.1 Assumptions

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

(1) This is a postpass approach and captures full spills. However, it does not have access to higher-level information about array subscripts and, therefore, cannot do intraarray data layout. Such tradeoffs are quite common and, in fact, a postpass approach to capture all the spills was also followed in Cooper and McIntosh [1996]. On the other hand, our work allows us to rely on the best register allocators to remove as much spill code as possible and then optimize the rest into parallel load/store instructions.
(2) We account for ISA constraints because of encoding of parallel load/store instructions. Similar ISA constraints exist on other embedded processors with dual-memory bank and parallel load/store support. We tackle them on our target processor to demonstrate how they are incorporated into our framework.

[1]"Web" will be defined in Section 2.5.

2.2 Introduction to SONY pDSP

The SONY pDSP has eight general-purpose data registers d0 to d7 and eight address registers addr0 to addr7. Two banks of memories are accessible through separate buses, called X bus and Y bus. Both X bus and Y bus are 24-bit wide. Indirect memory accesses must go through address registers. Two parallel memory operations can work on different data registers and arbitrary memory locations as long as they are in separate banks. There are a number of ISA restrictions, as listed below. Those heterogeneities are considered in our solution to designing search strategies to achieve practical results.

(1) Only registers d0 to d3 can be used in the parallel load/store instructions. Other registers cannot be used because of encoding constraints (only two bits are available for designating a register).
(2) Registers used in a parallel load/store instruction cannot be the same.
(3) Only PLDST (load/store) or PSTLD (store/load), and PLDLD (load/load) are available. No PSTST (store/store) is available.

2.3 Classification of Memory Access Instructions

If we regard 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 address register addr

LD r, [main.x] 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 r, [main.x] and ST [addr], r. In the latter case, alias (actual symbol) analysis will be performed in an attempt to determine the actual symbol addr being pointed to. *Base-Offset load/store or Type II load/store* is written as ST $[r_{base} + r_{offset}]$, r or LD r, $[r_{base} + r_{offset}]$. The token r_{base} and r_{offset} can be either immediate or register operand, but at least one of them should be register operand. Thus, Type II load/store instructions contain at least two register operands. Normally, it is used to access aggregate data structures like an array. DSP instruction sets only contain very limited number of memory addressing modes. For our experiments on the SONY pDSP processor, all memory operations are either Type I or Type II load/stores. Our framework can also be applied to other types of load/store instructions as long as the register dependencies are considered.

For memory access instructions with indirect addressing mode like ST [addr], r, we need to determine what it actually points to. A simple alias analysis algorithm (refer to Section 4.1 for details) is invoked to find out the aliases for each indirect mode instruction. If the indirect addressing has multiple aliases, i.e., it might access more than one memory locations, we call it *multiple-aliased*.

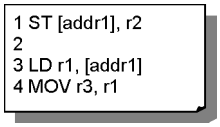

Fig. 1. Two types of conflicts.

2.4 Dependencies

There are two types of 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) cannot be moved before 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 instructions must not be violated. Register conflicts can happen between memory access and nonmemory access instructions.

For example, in Figure 1, the LD instruction cannot be moved beyond the ST instruction because of the address conflict. It also cannot be moved below MOV r3, r1 because of the register conflict.

2.5 Webs and Value Separation

In order to separate memory references, which can be independently considered for allocation purposes, we use a concept called "web." *Web* [Muchnick 1997] is defined as the maximal union of du-chains. A similar term called *live range* is defined by Chaitin et al. [1981]. For each definition d and use u, either u is in the du-chain of d or there exist $d = d_0, u_0 \ldots, d_n, u_n = u$, 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 definitions and uses within a web to a single memory location. In short, different symbol groups can be put in different banks. Notice that a single variable can be in several webs, which allows it to be allocated to different memory locations (banks) at different program points. Thus, separating variables into webs achieves effective value separation and finer granularity for bank assignment. On the other hand, webs can be split under certain circumstances. In our optimizations, we first assume webs are not separable. Later, we split large webs in one of the optimization steps to reduce the code size and speed up the execution.

2.6 Motion Range

A load/store instruction can move up and down, obeying dependencies. *Motion range* is defined as the interval between program points where a load/store instruction can be legally moved. In our framework, nonload/store instructions are assumed to be immovable. Each load/store instruction's motion range is bounded by *boundary instructions*, which could be load/store instructions or nonload/store instructions. In the latter case, the boundary is called *fixed boundary*, because nonload/store instructions are assumed not movable. On the other hand, if the boundary instruction is a load/store instruction, it is a *movable*

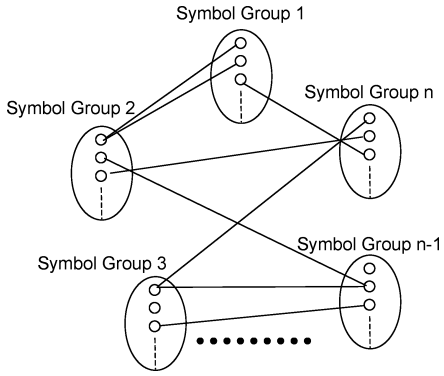

Fig. 2. A typical MSG.

boundary. For example, in Figure 1, the motion range for the LD instruction is <2,3> with the MOV as a fixed and the ST instruction as a movable boundary.

Notice that the movable boundary could be problematic, since itself might be moved. The so-called *movable boundary problem* will be introduced in Sectio 4.

2.7 Motion Schedule Graph (MSG)

After motion ranges are determined, a *motion schedule graph* (*MSG*) is built. The MSG is defined as follows:

DEFINITION: Motion Schedule Graph (MSG)

(1) An undirected graph.
(2) Each load/store instruction is a node on the MSG.
(3) An edge between two nodes tells us that they can possibly be combined, i.e., they have overlapped motion ranges where they could be moved and merged. Besides, ISA restriction should allow such merger.
(4) 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 because of the unique memory allocation given to a web).
(5) Each symbol group is designated by a big supernode on MSG, encompassing several nodes in that symbol group. A symbol group can only be colored with one of the two colors, which is equivalent to the bank assignment of the memory location to it.
(6) All the loads and stores within a given symbol group must take place as the assigned memory bank and location. Load/store nodes in the same symbol group assume the same color given to a symbol group.

An example of the MSG is shown in Figure 2.

After constructing the MSG, we need to solve the MSG. The solving of the MSG is to color each node such that a maximal number of node pairs (i.e., two nodes with different colors that are connected) could be picked. In other words, solving the MSG is to decide both the color (bank) for each node (symbol group) and the node pairs.

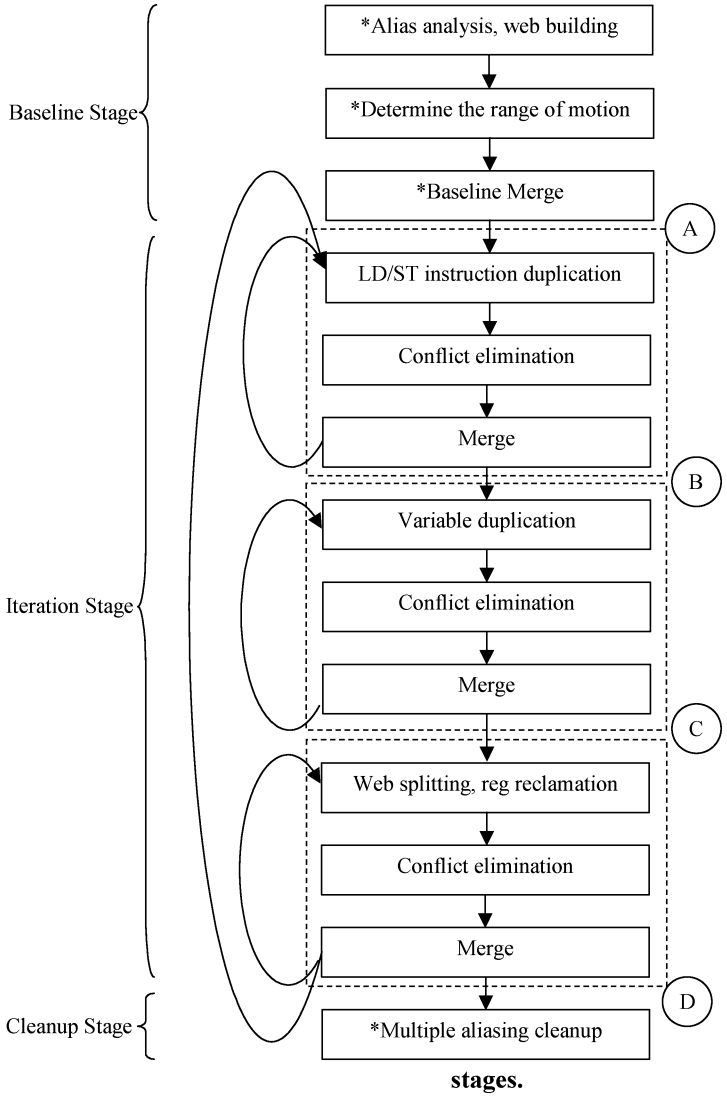

Fig. 3. Optimization stages.

3. OVERVIEW

Figure 3 shows the overall approach. Multiple steps are categorized into three major stages—*baseline stage*, *iterative stage*, and *cleanup stage*. In Figure 3, phases marked by "*" are mandatory stages.

In the baseline stage, we first identify the variables pointed to by address registers via indirect addressing mode. We then build webs to maximally separate the values. Next, the motion ranges of load/store instructions are determined. The last step in this stage attempts parallelization of the load/stores with so called *baseline merge*, i.e., without any duplication (including load/store instruction duplication and variable duplication) and web splitting. Evaluation results

for the baseline stage are collected at point A (cleanup stage is always included as the last and mandatory stage).

After the baseline stage, we have the iteration stage with three optimization steps. Since the optimization steps interfere with each other, i.e., one optimization step may create new chances for other optimization steps, a top-level loop covering all three steps is applied to capture the new opportunities. The first optimization step tries to perform cross-basic block load/store duplication only when it is profitable—it examines the remaining load/stores and determines the ones that can create new merges after duplication. After a load/store is duplicated, conflicts are resolved and a merger is then attempted. To capture the secondary effects of load/store duplication, we iterate until no more opportunities are left. In the second optimization step, we merge the remaining load/stores through variable duplication, which inserts extra stores to put variables in both banks. However, it creates new opportunities to merge loads, since the variable is available from both banks. In an iterative manner, such opportunities are exhausted. The third optimization step tries web splitting and register reclamation. These break the limitation of variable location because of webs and allocates the variable on the web to different memory banks. It also looks for regions with light register pressure after web splitting so that some of the spilled load/stores can be reclaimed.

The phase ordering of instruction duplication, variable duplication, and then web splitting is motivated by the fact (as we will show in the results) that after baseline merge, instruction duplication contributes the most to speedup, variable duplication the second, and web splitting the least. These steps are designed to be optional and incremental, i.e., we can invoke none of them, or invoke the first step only, the first and second steps, or all the steps. This is because, by adding more optimization steps, it costs more compilation time, possible code growth, but diminishing returns for speedup. Therefore, it is up to the user if he wants to pursue more speedup. The iterations are guaranteed to finish, as will be addressed in Section 5.6.

Depending on the number of steps contained in the iteration stage, we have three configurations for data collection. If the data collection point is B, the iteration stage only has one step, i.e., the instruction duplication step. If the data collection point is C, the iteration stage does both instruction duplication and variable duplication and iterates repeatedly until convergence. If the data is collected from point D, three steps are all included. By configuring the data collection differently, we can show the incremental effects of each step.

Finally, the cleanup stage finds out multiple-aliased indirect mode load/store instructions; the goal of this stage is to combine multiply aliased load/stores when it is safe to do so. This stage is always conducted for all data collections.

4. BASELINE STAGE

The baseline stage takes generated code in CFG (control-flow graph) form as input. We first identify all the load/stores in the program and then determine the exact variable (memory location) for each load/store through alias analysis. A combined data-flow analysis (liveness and reaching definition) builds

webs among all load/store instructions. We then determine the motion range of each load/store instruction. Then comes the issue of movable versus fixed boundary for load/store instructions. We will propose an approach to solve the graph problem assuming all the boundaries to be fixed and then resolve the conflicts afterward. Solution to his graph problem is shown to be NP-complete. The two heuristics in this section have different efficiency and code quality. The fast heuristic algorithm is preferred for practical applications with minor performance loss.

4.1 Alias Analysis

We first identify load/stores and then perform alias analysis to determine the association of a load/store with memory locations (memory disambiguation). To determine aliases of load/stores, we use a simple algorithm from Aho et al. [1986]. Our statistics show that in about 94% cases, the pointed variable can be uniquely determined. Among the rest, most of them are multiple-aliased, which means the address register can point to two or more different variables. For a multiple-aliased variable, we must assume the worst case, i.e., it can point to any one of the possible memory locations. If a load/store is potentially aliased with multiple memory locations, we initially exclude the given load/store from potentially combining it with other load/stores. This is done because of the following. If a multiple-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. On the contrary, we put a cleanup stage after all other stages that tries to merge those instructions whose target registers point to the memory locations that are *all* allocated in the same bank. In such cases, it is safe to parallelize such load/stores, since the access is definitive with respect to either from X or Y bank.

4.2 Building Webs

Once we select the load/stores to be included in our analysis, we build webs to achieve value separation, i.e., a single variable could be on several webs and assigned to different memory locations (banks). Value separation offers us more freedom in assigning banks and increasing parallel load/store opportunities than working on the variables themselves. This is especially true for temporaries that are recycled. Each web is found by undertaking a transitive closure of du/ud chains [Muchnick 1997]. One of the important properties of 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 it, which obviates the need for any analysis during the motion.

4.3 Movable Boundary Problem and Conflict Resolution

Once webs are built, we determine the motion range for all load/store instructions and build the MSG. As mentioned earlier, if two load/stores have

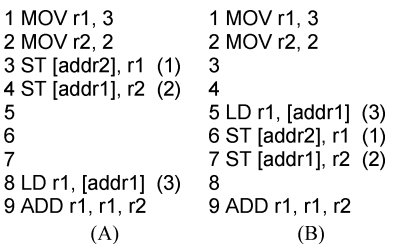

Fig. 4. The movable boundary problem (A) original code (B) illegal code after motion.

overlapped motion ranges, they are linked with an edge on the MSG, which means they can be combined. However, here we are faced with an issue of fixed versus movable boundaries—motion of a load/store could impact motion of others. Figure 4 shows an example.

Instruction (1) has boundary of <2,7>, which signifies that the instruction cannot 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 movable 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 movable (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 the one at label 8 is fixed (register conflict). After motion of load/stores, in Figure 4B, the fixed boundaries remain unmodified, but movable boundaries could assume different values and thus one must update boundaries and ranges for load/stores or illegal code (such as shown in Figure 4.B) might result. However, updating boundaries and edges might lead to nonconvergence of the algorithm, i.e., the solution we got from the graph may not be a feasible solution in the code. Thus, we have to go back and forth and end up with repeatedly solving the problem, which leads to the suboptimality and unanalyzability of the problem. Therefore, we propose our solution as follows.

Given that movable boundaries can complicate the solving of MSG, our approach temporarily assumes all movable boundaries to be "fixed" before the MSG solving. We call these boundaries *Pseudofixed Boundaries*, i.e., they are actually movable boundaries, but assumed to be fixed. The MSG is solved without consideration to the movable boundary problem. Once we have colored all nodes and picked node pairs to merge, the movable boundary problem is tackled afterward. To avoid illegal code that results from the movable boundary problem, we need an additional step after MSG solving called *Conflict Resolution*. Next, we illustrate how conflict resolution is carried out. There are two kinds of conflicts that should be resolved, i.e., address conflicts and register conflicts. We only look at the Type I load/store as defined in Section 2.3, since Type II load/stores can be tackled in a similar manner, except that more register conflicts should be considered.

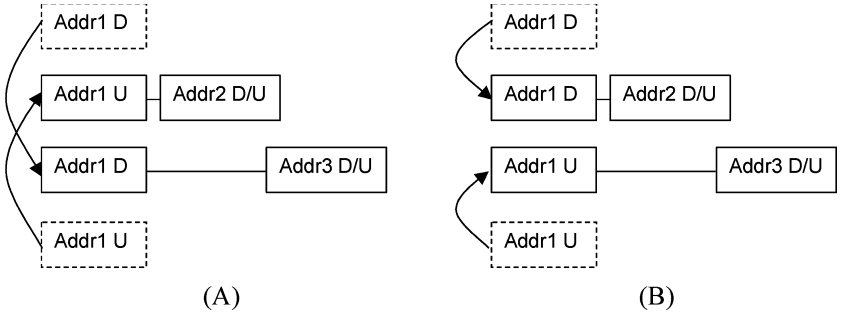

Fig. 5. Resolving address conflicts.

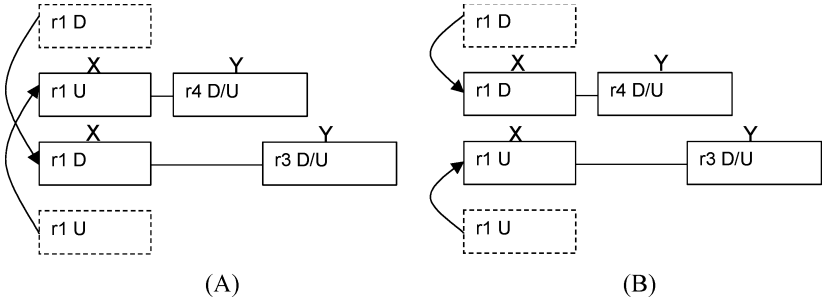

Fig. 6. Resolving register conflicts.

4.3.1 *Resolve Address Conflicts.* In Figure 5, the notation addr D denotes the definition of address addr. Similarly, addr U denotes the use of address addr. The notation r D denotes the definition of register r and r U denotes the use of register r. Because of the motion of load/stores, sometimes address conflicts can occur. The address conflicts are always resolvable, as shown below.

Figure 5A 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 causes the use of Addr1 going before the definition of Addr1, violating the semantics. However, as shown in Figure 5B, one could always exchange the instructions merged with Addr2 and Addr3 and combine Addr1 D with Addr2 and Addr1 U with Addr3.

4.3.2 *Resolve Register Conflicts.* There are two cases for register conflicts. Figure 6 illustrates one type of register conflict, if the memory banks of both instructions using r1 happen to be X. We can resolve the conflict in a way similar to the address conflict discussed earlier, by exchanging the position of the two instructions. We now look at the other type of register conflict. In Figure 7, if we 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 thing happens to 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 here is to combine r4 D/U and r3 D/U and this is possible only if they have overlapped motion ranges.

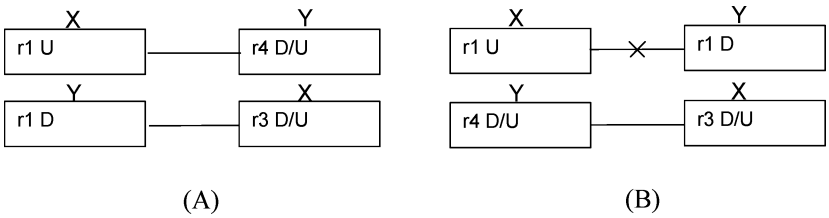

Fig. 7. Another kind of register conflict.

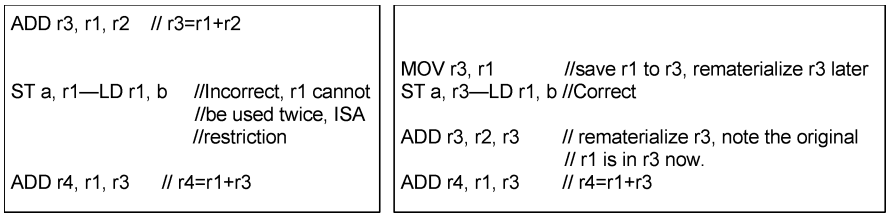

Fig. 8. Free registers through rematerialization.

If this does not hold, we have to give up the merger. r1 U and r1 D always share part of the motion range, however, they cannot be merged because of the same register (r1) used. Note that, in this case, r1 must be used in a PSTLD (or PLDST) instruction, in which the instruction stores the content in r1 to a memory location and then fills it with value from another memory location. Thus, the content of r1 before this instruction will no longer be used after it is stored into the memory location. Here, we only need a temporary register to hold this value and to legitimate the parallel memory access instruction. The temporary register can be freed again after this instruction. In case there is an unoccupied register, we need to insert a MOV instruction to transfer the content of r1 to this register and replace one of the registers (the source of the ST) in the PSTLD/PLDST instruction with this register. If no unused register is available at any point, the two instructions can be merged, we try to rematerialize some registers to create a free register.

Figure 8 shows an example, which illustrates rematerialization[2] and allows merging the load/store instructions using the freed register, otherwise the merge is illegal because of ISA restrictions. Register r3 is thought to be rematerializable. Therefore, we save the content of r1 to r3 before the merged load/store. Memory locations a and b are in different banks. The merged instruction is legal, which stores r3 (the same as r1) to a and loads b to r1. We then, rematerialize r3 by adding r1 and r2 again noticing the original r1 has been saved in r3. Finally, we get the same contents in r1 and r3 as the original code. Our experiments show that we need the conflict resolution rarely and almost all the conflicts can be resolved with the above approaches.

[2]Rematerialization is a compiler technique which replaces memory access with instructions that recalculates the value [Chaitin et al. 1981; Briggs et al. 1994].

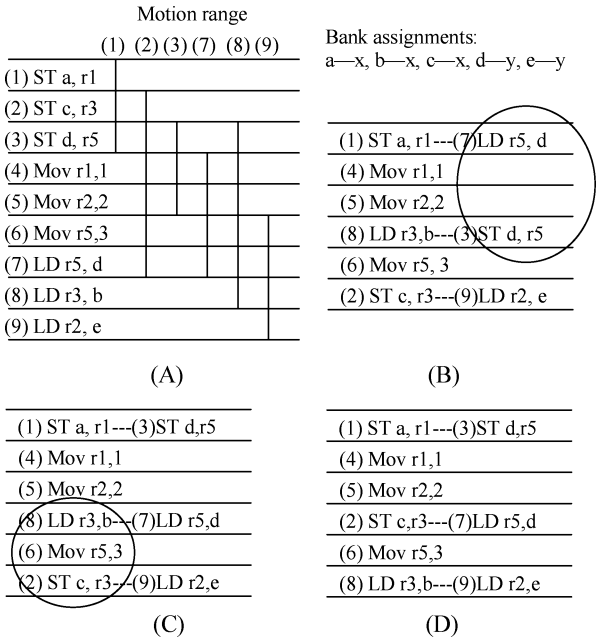

Fig. 9. Example for address/register conflict resolution.

4.3.3 *Example.* Here, we give a more complicated example to illustrate both address and register resolution. The example in Figure 9 shows a nine-instruction assembly code segment. In Figure 9A, we list the motion range of each memory instruction. As noted earlier, the motion ranges put constraints on instruction motion and thus obeying these constraints we could move and merge the instructions as in Figure 9B without any violation of the motion ranges of the instructions. However, because of 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 9C, the address conflict is resolved by exchanging instruction (3) and (7). In Figure 9D, the register conflict is resolved by exchanging instruction (2) and (8). The resulting code is now legal and all load/store instructions are merged.

4.4 Solving the MSG

To solve the MSG, both the color (bank) of each symbol group and the maximal number of 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. The problem is NP-complete, as proved in the following theorem.

THEOREM: *MSG solving (find the bank assignments to the nodes and determine the maximal number of node pairs) 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. First, 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.

(1) Edges and nodes belong to the graph.
(2) Two nodes on the edge have different colors.
(3) The number of node pairs is greater than k.

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

Given are objects with sizes $A_1, A_2, \ldots A_n$. Can these objects be split into two 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 numbers $A_1, A_2 \ldots .A_n$. All nodes are fully connected (all edges exist). Solving our problem optimally means that we have found the

$$MIN\left(\left|\sum_{i \in T} A_i - \sum_{i \in S} A_i\right|\right)$$

Remember that $\sum A_i$ is a fixed number, so

$$MIN\left(\left|\sum_{i \in T} A_i - \sum_{i \in S} A_i\right|\right)$$

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 the MSG is NP-complete. □

Given this intractable problem, we want to reduce the search space as much as possible so that exhaustive search is feasible for small size problems. Also, proper heuristic methods should be proposed for MSGs with large sizes to give out tolerable compilation time and acceptable code quality. Note that after assigning color to each symbol group (node), we must give out a schedule of which edges should be used for merging. Thus, the optimizing problem can be separated into two parts: (1) assign bank to each symbol group; and (2) maximally pick edges on the MSG given the bank assignments to symbol groups. The following sections first identify the edge-picking problem to be polynomial-solvable. Then we propose heuristics for the bank assignment problem.

4.4.1 *Maximal Pair Picking Modeled as Maximal Flow Problem.* As one of our objectives, we should maximize the node pairs given the bank (color) assignment of each symbol group. Here we show that the optimal solution of finding maximal number of pairs can be solved in polynomial time. Suppose there are n symbol groups, some symbol groups are assigned memory bank X and others are in memory bank Y. We prove that the problem of selecting maximal number of combinable edges given the colored graph is polynomial time solvable after it is modeled as the maximal flow problem. The problem is

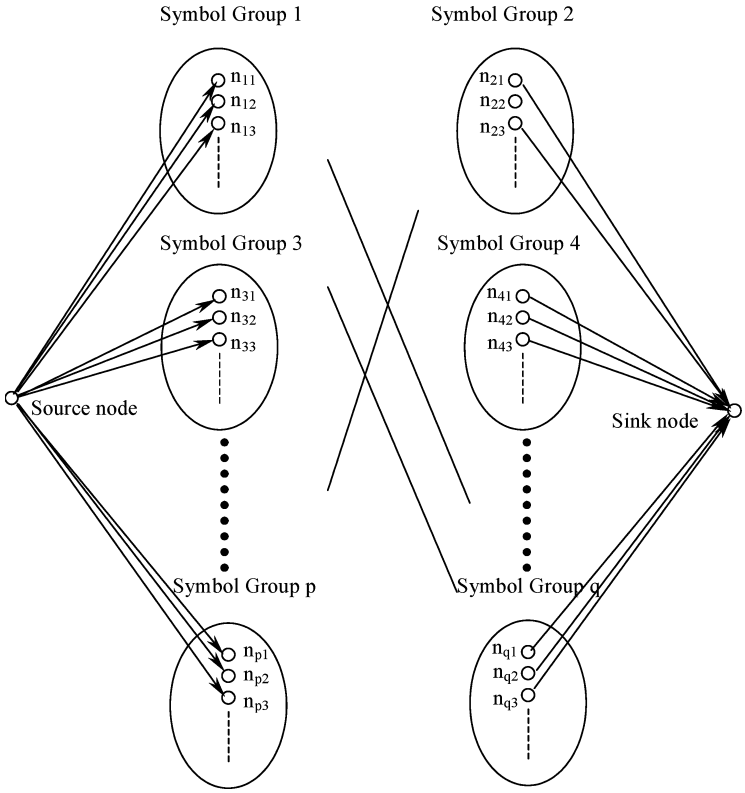

Fig. 10. Convert MSG to maximal flow graph after bank assignment.

actually equivalent to a classic graph problem called *maximal bipartite graph matching* [Gross and Yellen 1999].

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 is deleted. 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, some of the edges are occupied. We can see the edges with flow in the middle 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 maximized in a sense that the number of pairs is maximized.

4.4.2 *Brute Force Searching.* The complexity of solving the maximal flow problem is $|V|^3$ [Papadimitriou and Steiglitz 1998]. However, as a special case, this problem can be solved in $|V|^2$ time by marking edges and nodes (Figure 11), where $|V|$ is the number of nodes in the graph. The algorithm takes the bank assignments represented as an n bit vector V, where each bit stands for the bank assignment (0 for x, 1 for y) of one symbol group and returns the value of the maximal flow.

```
Input: Bank assignment as a n-bit vector
Output: value of the max flow.
Algorithm:
Max_flow(V)
1.   mf=0
2.   for each Node n_x in X-bank {
3.      Pick an edge from n_x to n_y where both n_y and the edge are not marked
4.      mf++
5.      Mark n_x , n_y , edge (n_x , n_y)
6.   }
7.   return mf
```

Fig. 11. Calculate the maximal flow in $O(n^2)$.

Obviously, an $O(|V|^2 2^n)$ exhaustive searching algorithm can get the optimal solution, where n is the number of symbol groups (we color the n symbol groups first and then do the maximal flow on the colored graph), which leads to this complexity. Comparing to [Powell et al. 1992], our complexity is greatly reduced. Actually, for all the DSPstone programs, the exhaustive searching can finish the compilation within an affordable time period (from several minutes to one-half an hour, which is affordable as a one-time optimization phase). However, it fails for large-sized programs prompting heuristic solutions.

4.4.3 *Heuristic Solutions.* We choose *simulated annealing (SA)* algorithm [Aarts and Korst 1989] as the first heuristic solution 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 bank assignments of the n symbol groups are still represented as an n bit vector V, so the search space is the 2^n n-bit vectors.

Initially, we pick a random vector as the starting point. From each vector, we check directions to go inside the search space by inverting some of the bits. If the new vector is better than the maximum of all previous ones, we will definitely accept it as the next step to go. Otherwise, the new vector is accepted with certain probability. The probability decreases as the temperature T decreases. The detailed pseudocode of the algorithm is shown in Figure 12. The maximal iteration step max_step, the initial temperature T_{init}, and the iteration number for each temperature step k are inputs to the function. The function outputs the number of merged pairs mf_max and the bank assignments $V_{\max}$. Here, function *max_flow* (Figure 11) calculates the optimal number of disjoint edges or the maximal flow.

The second heuristic algorithm (Figure 13) is a greedy algorithm, which searches the solution space by gradually adding node groups to two determined sets (DETX for the X bank and DETY for the Y bank). After bank assignment is determined by function *assign_bank()*, we call *max_flow* to get the maximal number of node pairs. We define the *connectivity* between two symbol groups as the number of edges between the nodes of the two symbol groups. During each step, we pick an undetermined symbol group with maximal connectivity to one of the determined set (DETX or DETY) and add it to the other determined set, therefore using a greedy strategy to maximize the connectivity (number of

```
Input: MSG
       max_step //total iteration to do
       T_init  //initial temperature
       k      //#steps for each temperature
Output:
       V_max //bank assignment of the n symbol groups
       mf_max //max number of merged pairs
Algorithm: Graph_solving
1.   V_old=initial random n-bit vector
2.   T=T_init
3.   While(step <>max_step){
4.      for each temperature, iterate k times{
5.         V=Change several bits of V_old
6.         mf=max_flow(MSG, V)
7.         if (mf>mf_max) {
8.            mf_max=mf
9.            V_max=V_old=V
10.        }
11.        else{
12.           if (exp^(mf-mf_max)/T>Rand(0..1))
13.              V_old=V
14.           else
15.              keep V_old
16.        }
17.        step++
18.     }
19.     decrease T
20.  }
21.  return V_max, mf_max
```

Fig. 12. The simulated annealing algorithm to determine the bank for each symbol group.

Input: the MSG
Output: an n-bit vector representing the bank assignment for the n symbol groups

E= the set of all symbol groups
DETX={S_x} where S_x is the symbol group with all the nodes that must be put in X bank memory.
DETY={S_y} where S_y is the symbol group with all the nodes that must be put in Y bank memory.
C_{ij} = the number of edges between symbol group i and symbol group j

Algorithm: Assign_bank
1. DETX=DETY=Φ
2. While (E<>DETX $\cup$ DETY) do
3. For each $S_k \in$ E-DETX $\cup$ DETY {
4. Calculate $CX_k = \sum_{S_i \in DETX} C_{ki}$ and $CY_k = \sum_{S_i \in DETX} C_{ki}$
5. }
6. CX_max=max(CX_k| $S_k \in$ E-DETX $\cup$ DETY)
7. CY_max=max(CY_k| $S_k \in$ E-DETX $\cup$ DETY)
8. If (CX_max>CY_max){
9. Add the symbol group with CX_max to DETY
10. } else{
11. Add the symbol group with CY_max to DETX
12. }
13. }
14. convert DETX, DETY to a n-bit vector V and return V

Fig. 13. A greedy heuristic algorithm to determine the bank for each symbol group.

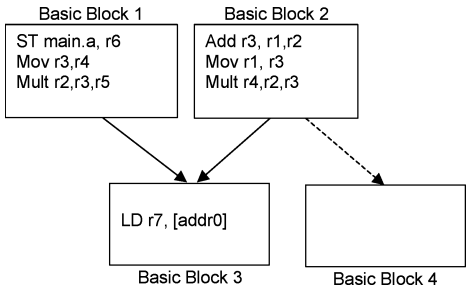

Fig. 14. Example for instruction duplication.

edges) between DETX and DETY. This algorithm can finish very fast. In the results section we will show that the greedy heuristic gives a comparable solution to exhaustive approach with far superior compilation speed. Moreover, by performing some other optimizing steps discussed later, we are able to outperform the simulated annealing approach (which does not have those steps) in some circumstances.

5. ITERATION STAGE AND CLEANUP STAGE

In this section, we discuss the other stages, i.e., the iteration and the cleanup stage. The iteration stage consists of a two-level iteration loop and the cleanup stage is a mandatory stage. We will also address the complexity and convergence of the iteration stage in Section 5.6.

5.1 Merge with Load/Store Instruction Duplication (Cross-BB Merge)

Duplicating load/store instructions across basic blocks can create new opportunities for combination. *Cross-BB merge* can link uncombined load/store instructions in different basic blocks if they are in different banks. Generally, moving a load/store instruction across-basic blocks requires duplication of the instruction in the successor or predecessor of the basic block, so cross-BB merge is also called *load/store instruction duplication*.

However, cross-BB merge may not be profitable. Figure 14 shows a program segment with four basic blocks. If the instruction ST main.a, r6 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 type of motion, control flow may frequently go from basic block 2 to basic block 3 (we assume no profiling information is available to prevent this). On the other hand, if LD r7, [addr0] is moved to basic block 1, it must also be moved to basic block 2. If the control frequently goes from basic block 2 to basic block 4, most likely the load instruction will be unnecessarily executed. Actually this is because the control flow edge from basic block 2 to basic block 3 is a critical edge [Knoop et al. 1992].

To prevent degradation, several conditions are added to load/store instruction duplication in order to determine whether it is profitable to perform the optimization, which serve as useful guidelines:

```
Algorithm: Ldst_Insn_Duplication
1.    for each b ϵ set_of_basic_blocks do {
2.       /*for load instructions*/
3.       build the reverse EBB tree rooted from b: t_rebb
4.       for each unmerged load instruction i {
5.          /*keep only basic blocks where the variable accessed by i is live*/
6.          t_rebb=Prune_not_live(t_rebb, i)
7.          if  (all b's predecessors are on t_rebb) AND
8.               (i can be  merged in one of b's predecessors on t_rebb) {
9.               remove i from b
10.              push i to b's predecessors to merge it
11.              insn_dup_changed=TRUE
12.           }
13.      }
14.      /*for store instructions*/
15.      build the EBB tree rooted from b: t_ebb
16.      for each unmerged store instruction i do {
17.         t_ebb=Prune_not_live(t_ebb, i)
18.         if (all b's successors are on t_ebb) AND
19.            (i can be  merged in one of b's successors on t_ebb) {
20.            remove i from b
21.            push i to b's successors to merge it
22.            insn_dup_changed=TRUE
23.         }
24.      }
25.   }
26.   return insn_dup_changed
```

Fig. 15. Algorithm for load/store instruction duplication.

(1) Unmerged load/store instructions should only be moved to the successors/predecessors, where the reference is live.
(2) The motion range of a store instruction is the EBB (Extended Basic Block) rooted from the original basic block. The motion range of a load instruction is the reverse EBB rooted from the original place.
(3) For each uncombined load/store, cross-BB merge is only performed if pushing it to at least one of its live predecessors/successors (inside the EBB/reverse EBB) allows it to be combined with other load/stores.

The first condition prevents the creation of dead load/store instructions after cross-BB motion. Figure 14 explains the second condition. The LD instruction in basic block 3 is moved to predecessors if there is no basic block 4 (so basic block 1 and 2 are on the reversed EBB of basic block 3). Similarly, ST main.a, r6 cannot be pushed down to basic block 3, as basic block 3 is not on the EBB of basic block 1. Finally, the third condition is obviously necessary, because otherwise the duplicated load/store instructions are wasted.

Figure 15 gives the algorithm for load/store duplication. Basically, it works on all basic blocks one-by-one. For each basic block, the algorithm has two parts, one for unmerged load instructions and one for unmerged store instructions. For unmerged load instruction i, the reversed EBB rooted from basic block b is built and stored to t_rebb. Then the function *Prune_not_live(t_rebb,i)* removes all basic blocks where the variable accessed by i is not live. Note that the variable

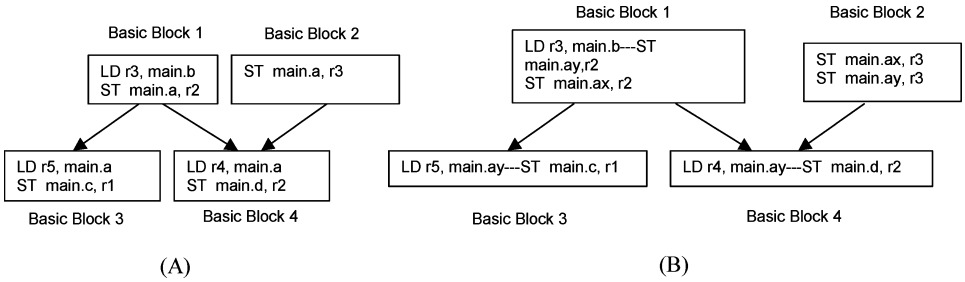

Fig. 16. Example for variable duplication.

accessed by i must be on a web, which is a connected graph. Thus, *Prune_not_live* returns a connected tree of basic blocks, on which instruction i can be moved. In the next step, we check if b's predecessors are all on the t_rebb and if merge is possible in at least one of the predecessors of b on t_rebb. Notice that we only try the merger for one level. Although i may be moved up again on the t_rebb, it will lead to more code growth for this instruction. The otherone-half of the code deals similarly with store instructions. Finally, we return variable insn_dup_changed, which indicates if there is any change in this invocation. Once the return value is false, the iteration of this step comes to an end.

5.2 Merge with Variable Duplication

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

As a simple example, we illustrate how the variable duplication is performed in Figure 16.

In Figure 16A, if all variables are in bank X (after the previous phases), all load/stores are uncombinable in the graph. During variable duplication, we check the web of variable main.a. We can store main.a to bank Y as well as to bank X (copy of main.a in bank X has to be kept, since it is preferred by the previous optimizations). In this figure, we have two store instructions for main.a, so we must add two store instructions, which store main.a to a place in Y bank. For distinction, we write main.ax and main.ay as the two memory locations for 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 16B shows the resulting control-flow graph.

The pseudocode of our algorithm is shown in Figure 17. The algorithm works on webs one by one. First, each store instruction on the web is duplicated into two store instructions that store the variable to both banks. All load instructions

```
Algorithm: Var_Duplication
1.   for each w ϵ set_of_webs do {
2.      var load_merge_num=0
3.      var store_merge_num=0
4.      var store_insert_num=number of stores on the web
5.      Duplicate each store instruction to two stores that store to both banks.
6.      for each load l on the web {
7.         Assume l can be load to either X or Y bank
8.         if l can be merged within its basic block
9.            load_merge_num++
10.     }
11.     for each store s on the web {
12.        if s can be merged within its basic block
13.           store_merge_num++
14.     }
15.     if (load_merge_num+store_merge_num>store_insert_num) {
16.        var_dup_changed=TRUE
17.        Commit merges and store instruction insertions
18.     }else
19.        Rollback changes.
20.  }
21.  return var_dup_changed
```

Fig. 17. Algorithm for variable duplication.

are then, assumed to be dual-bank, i.e., they can always be merged with other load/store instructions regardless of the bank assignments. We attempt to merge these load instructions within their basic blocks. All store instructions (after duplication) are also tried for merge within the basic blocks. However, the store to X bank memory will assume X bank and the store to Y bank memory must assume Y bank. Finally, to decide the profitability, we count the number of duplicated store instructions store_insert_num, the number of merged loads, load_merge_num, and the number of merged stores, store_merge_num. If load_merge_num + store_merge_num>store_insert_num, we have reduced the number of memory instructions, all changes are committed. Otherwise, the original code is restored. If any change is committed during this optimization step, we mark the global variable var_dup_changed to be TRUE, so other optimization steps will be reattempted.

5.3 Web Splitting and Spill Reclamation

In this section, we propose web splitting—an approach to cut the webs into smaller size and create more chances for the load/store instructions on the web to be merged with other memory access instructions.

As mentioned before, webs are maximal union of du chains. Since the load/store on the same web must be allocated to the same memory location, which means they must be put in the same memory bank, large size webs may prohibit load/store merges because of the memory bank restriction, i.e., it is hard to require that a large number of load/stores to be in the same memory bank and still get merged. Web splitting can assign part of the web in one memory bank and part of the web in the other bank. To keep the content consistent,

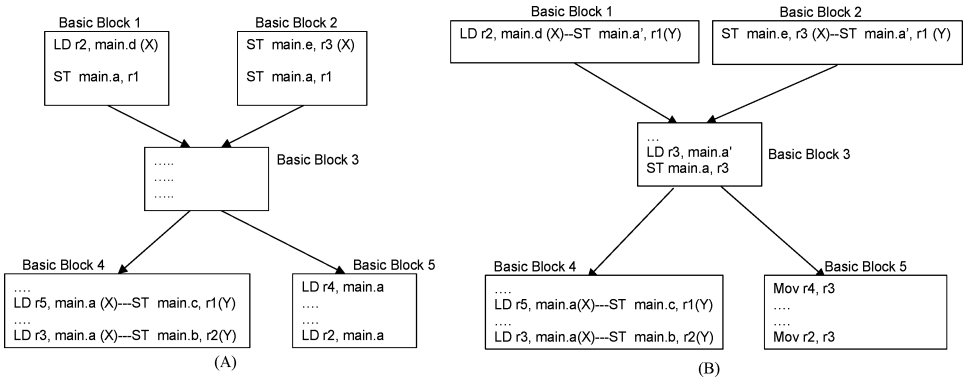

Fig. 18. Web splitting example.

additional copy instructions should be inserted in proper locations. Web splitting differs from variable duplication in that the variable is not duplicated in both banks throughout the web, but is distributed in different banks for different parts of the web. Web splitting, therefore, does not need to insert copy instructions for each definition on the web.

Another by-product of web splitting is that we may create chances to reclaim the spilled variable and put part of the split web back to register. As a matter of fact, spill code counts for most memory access instruction. Generally, register allocation has to push register variable to memory when register pressure is high in some points. According to several classic register allocation algorithms like [Chatin et al. 1981; George and Appel 1996], once a virtual register is decided to be spilled, all its occurrences are spilled, even if the register pressure is not high for some occurrences. By splitting the web, we can actually rediscover the part of the spill code locations with low register pressure. These load/stores can then be put back to available registers.

5.3.1 *Example.* Here we give an example to show how web splitting is profitable, how to split the web, and how spill code is reclaimed. In Figure 18A, variable main.a appears in four basic blocks and its web spans five basic blocks.

The two load instructions in basic block 4 must be tied to bank X to be combined. Since the two uncombined ST instructions of main.a in basic block 1 and 2 are on the same web as the ones in basic block 4, they are left unmerged—as the other two memory access instructions in basic block 1 and 2 are already assigned to memory bank X. Furthermore, the two stand-alone load instructions in basic block 5 cannot be merged anyway. However, the web of main.a is special in that basic block 3 acts as a connecting point between the upper and lower half of the web. In other words, the cut size of the web is only 1. Also, the register pressure is heavy only in basic block 4, while we have free registers in all the other basic blocks.

Actually, we can identify a significant number of webs in benchmark programs like this, which have uneven register pressure or biased merge situations in different parts of the web. In Figure 18B, the web is split into three parts. First, variable main.a in basic block 1,2 is renamed to main.a′. By splitting the

web, we put the same variable into different memory locations, so they can be allocated independently. The newly inserted variable duplication instructions in basic block 3 can keep the two memory locations consistent. Instruction LD r3, main.a′ terminates the web for main.a′ and instruction ST main.a, r3 starts the web for main.a. To exploit the low register pressure in basic blocks 3 and 5, we simply put part of the web in register and eliminate all the load/store instructions in basic block 5. Here, we put the variable in register r3 and substitute it for all memory access instructions in basic block 5 (other optimization techniques have not been applied after the substitution). Compared with Figure 18A, we have merged two additional load/store instructions and eliminated other two. The cost is the memory copy operation, which moves main.a′ in bank Y to main.a in bank X. In case of no direct memory copy instruction, we need two instructions to complete the memory copy.

To split webs, we need to decide how the web is cut into smaller parts. Firstly, we separate out the web from the original flow graph. Then, starting from each boundary load/store instruction, i.e., the boundary instructions of the web's live range, we extend the instruction to a group of instructions with similar properties, like register pressure and memory bank preference. Finally, the cut locations (where the copy instructions should be inserted) are determined. This approach involves several heuristics to decide splitting. After decisions are made, a tentative splitting will be performed to see whether it is profitable. The number of inserted copy instructions will be compared against the load/store instruction reduction in terms of both newly merged load/stores and those reclaimed to registers. If the splitting is not profitable, we simple give up the web splitting.

However, web splitting has both pros and cons. If we split a web into several smaller webs, the compiler must allocate more space in memory, while originally, only one memory location is enough for each web. This increases run-time memory pressure. Values on different split webs should also be consistent. Therefore, we must insert additional memory copy instructions like in Figure 18B, which can slow down the program execution speed. Given these disadvantages, we should apply web splitting cautiously. In our framework, web splitting is done as the last optimizing step. We will look at the webs with big size and large portion of uncombined load/stores. Webs with uneven register pressure will be also scanned for potential spill reclamation.

5.3.2 *Algorithm for Web Splitting and Spill Reclamation.* Figure 19 gives the pseudocode for web splitting and spill reclamation. We have to omit some details especially those dealing with webs with different shapes. The program maintains a number of sets called web_sub_group. Each web_sub_group initially consists of one load/store instruction on the boundary of the web, i.e., the boundary of the live range. We gradually add each internal load/store instruction to one and only one of the web_sub_groups. An internal load/store instruction is added to one of the groups when it is adjacent to one of the groups, i.e., it is a neighbor to one of the instructions in that web_sub_group in program execution order where only instructions on the web are considered. According to preference of this instruction (we look at the other unmerged load/stores that can be

```
#DEFINE WEB_SPLIT_LIMIT 6
struct {
   insns: set of load/store instructions in the sub-group
   reg_pressure: interger //maximal register register among all load/store instructions in the sub-group
   num_x_prefer: interger //number of load/store instructions which prefer X bank
   num_y_prefer: interger  //number of load/store instructions which prefer Y bank
} web_sub_group[]

Algorithm: Split_Webs
1.    for each w ϵ set_of_webs and sizeof(w)> WEB_SPLIT_LIMIT {
2.       let k=number of boundary load/store instructions on web w
3.       let X1, X2, …Xk are the boundary load/store instructions
4.       let web_insns=set of all load/store instructions on web w
5.       for i=1 to k do {
6.          web_sub_group[i].insns={Xi}
7.          remove Xi from web_insns
8.       }
9.       while (web_insns <> Φ) {
10.         get one instruction t from web_insns
11.         if t prefers X bank then
12.            move t to one of the web_sub_group[].insn with highest num_x_prefer and adjacent to t*
13.            web_sub_group[].num_x_prefer++
14.         else
15.            move t to one of the web_sub_group[].insn with highest num_y_prefer and adjacent to t*
16.            web_sub_group[].num_y_prefer++
17.         fi
18.         update reg_pressure if the register pressure at this instructions is higher than that.
19.      }
20.      /*tentative merge and cost calculation*/
21.      for each web_sub_group do {
22.         if num_x_prefer> 2*num_y_prefer then
23.            assign this group to bank X
24.         else if num_y_prefer> 2*num_x_prefer then
25.            assign this group to bank Y
26.         else if reg_pressure># of available registers then
27.            /* Later we need to check if one common free register is available for all instructions in
28.            the subgroup*/
29.            put this group as register reclamation candidate
30.         else leave it un-modified
31.      }
32.      insert all copy and additional load/store instructions at the join point to make values consistent
33.      if possible reclaim spills for register reclamation candidates, change the instructions.
34.      calculate the number of additional instructions and the number of merged instructions
35.      if cost is greater than gain then {
36.         Commit all modifications
37.         web_splitting_changed=TRUE
38.      } else
39.         Rollback changes
40.   }
41.   return web_splitting_changed
*adjacent means one of the instructions in web_sub_group[].insns is a neighbor in program execution
order to t, if only the instructions in w is considered.
```

Fig. 19. Algorithm for web splitting and spill reclamation.

merged with this instruction, if more are in bank X then this one prefers bank Y, otherwise it prefers bank X), the instruction is added to the web_sub_group with maximal value of its preferences. Meanwhile, we update the register pressure of the web_sub_group if the register pressure at the program point of this instruction is higher than the register pressure of that web_sub_group (i.e., the maximal register pressure for all instructions in the sub-group). In the next step, we

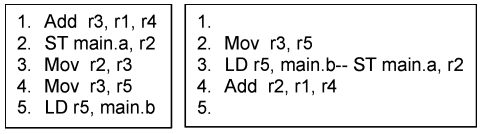

Fig. 20. Local conflict elimination.

try to do the merger and instruction insertion to see if it is profitable. Heuristically, for each web_sub_group, we check the value of num_x_prefer (number of instructions that prefer the X bank) and num_y_prefer (number of instructions that prefer the Y bank). If num_x_prefer is twice of num_y_prefer, we assign this group to bank X, which means the variable is stored to memory bank X when execution goes to this web_sub_group. Similarly, we decide the assignment for bank Y. Among the remaining web_sub_groups, probably, some can be totally put into registers if the maximal register pressure for all instructions in the group is less than the available registers. They are picked as candidates for register reclamation, although further checking is needed to find a register that is common to all instructions and no other instructions along the path where the web_sub_group occupies use that register. If a web_sub_group cannot be fit into any one of the above categories, it remains unchanged. The cost is calculated as the number of additional load/store instructions and register copy instructions required. The gain is calculated as the newly created merges for the instructions on the web. If the overall gain is larger, we commit all changes; otherwise, everything is restored.

5.4 Local Conflict Elimination

Because of dependencies, motion ranges are greatly hampered by immovable instructions. To eliminate the conflicts locally, we find those instructions with the conflicting registers and check if they are rematerializable (we follow the algorithm in Briggs et al. [1994] to determine the rematerializability). When a register can be freed and reconstructed after the merger, the instruction is removed to increase motion ranges so that more chances are created for the load/store instructions around it. Figure 20 shows a simple example for local conflict elimination.

The original code is listed in Figure 20A. The ST and LD instructions cannot be merged because of the two blocking MOV instructions. However, the value of r2 can be rematerialized by adding r1 and r4 again. In Figure 20B, the assignment to r2 is delayed to the end so that the two load/store instructions can be combined. This optimization works by looking up the instructions between unmerged load/store instructions and attempts to remove some of the blockades through rematerialization.

5.5 Multiple-Alias Cleanup

The last stage tries to merge all multiple-aliased load/store instructions. As mentioned before, if all variables pointed to by the address register are in the same memory bank, the instruction can be merged with other load/store

instructions in the opposite bank. Our compilation pass starts from each mergeable (all aliases are assigned to the same bank) multiple-aliased load/store instruction and checks for opportunities to merge it with nearby load/store instructions in the same basic block. If the instruction can be merged, the corresponding instructions are rewritten as parallel load/store instructions.

5.6 Complexity and Convergence of the Stages

Notice that our algorithms are all heuristic and are guaranteed to be polynomial time. From the results, we will see the compilation time is affordable. Although we cannot determine the exact number of iterations in the loops, the algorithm is guaranteed to converge eventually. As a matter of fact, during each iteration, the number of merged load/stores can only increase, otherwise the algorithm will stop because no change occurs. Although extra memory instructions might be created because of instruction duplication and variable duplication, the code growth has been controlled. Therefore, the total number of load/stores only increases slightly. Consequently, the algorithm must finish once no more load/stores can be merged.

6. GLOBAL OPTIMIZATIONS

We now discuss some enabling compiler optimizations to increase the effectiveness of our approach. By extending our bank assignment approach to include global variables and global optimizations, we show that the MSG solver can be easily applied with a few modifications.

6.1 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 must treat it as a fixed-bank variable. There are other reasons we have fixed-bank variables, like pointers passed as arguments, where the pointed data have already been placed in a memory bank. The caller function has to assume such variables to have fixed-bank assignment. Fixed-bank variables are easily incorporated into our algorithm. On 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. Our algorithms are modified slightly, e.g., for the greedy heuristic in Figure 13, symbol groups with fixed bank are already in the determined set, while other parts of the algorithm can remain unchanged. Besides, more future work could be conducted to analyze the instruction/variable duplication and web splitting for those variables.

6.2 Global Optimization Through Procedure Cloning

Our implementation also performs 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 procedures so that global optimization can be

fulfilled at the price of code duplication. Since, global optimization with procedure cloning and function inlining may cause dramatic code size increase, we only perform such optimizations for small functions or infrequently invoked functions (heuristically we designate functions with less than 20 instructions as small functions), so that the code size can be properly controlled.

7. PROFILE-DRIVEN OPTIMIZATIONS

Profiling information gives us more knowledge of the probability each path will take and each instruction will be executed. Heavily visited load/store instructions have higher priority to be merged. Actually, most approaches proposed earlier in this paper can be extended to include profiling information.

7.1 Basic Block Profiling and Path Profiling

Basic block profiling counts the number of times each basic block executes. For intra-basic-block merge, each edge on the MSG belongs to a basic block, i.e., the merge must occur in the basic block where the two nodes reside. After obtaining the profiling information for the basic block, we know how much benefit can be obtained by merging the two load/stores. For cross-basic-block merge (instruction duplication), the moved instruction can use the frequency counting in the destination basic block. Thus, our profiling information helps in both cases.

Path profiling looks at the execution frequency of edges between two basic blocks. By counting the number of times paths are taken, hot paths can, therefore, be identified. Our optimizing strategies proposed before can be modified to include profiling information and favor frequent load/store instructions for merging.

7.2 Profile-Driven MSG Solving

Previously, on MSG, all the edges have the same weight, or is unweighted. Straightforwardly, we should put weights on the edges, which subsume profiling information. As we know, any two neighboring nodes are in the same basic block. Therefore, the execution frequencies of all instructions in a basic block are identical. Therefore, we can simply assign the weight for an edge as the execution frequency of one of the nodes on the edge. The optimization problem then becomes how to select disjoint edges with maximal weights on the MSG. Still, each node should be assigned a color (bank assignment) and the two nodes on any selected edge should have distinct colors.

With some modifications, the approaches proposed in previous section can still work for profile-driven MSG solving. Again, we can separate the problem into two parts: node coloring and finding the maximal weight disjoint edges. The second step is still polynomial time solvable as we have shown below.

Notice that for predetermined node colors (bank assignments), although edges have nonuniform weights, the edge weights between nodes in the same basic block are equal. Or, the edges for each node have the same weight, since they connect to the neighbors in the same basic block, which have the same

execution frequency (i.e., the execution frequency of the basic block). Therefore, we can separate the nodes based on the basic blocks they belong to. Then, for each group of nodes that are in the same basic block, the previous MSG-solving algorithm (the maximal flow model) can be run with small changes (in line 4, Figure 11, it becomes $mf = mf + weight(n_x, n_y)$), so it can return the weight of all selected edges.

The simulated annealing algorithm needs no modification as long as the function max_flow can provide the sum of weights of all selected edges for the bank assignments of symbol groups. The greedy heuristic can still be used except that the connectivity of two symbol groups is redefined to sum the weights of all edges between the two symbol groups instead of just the number of edges. The symbol groups can then be selected to the determined sets based on the algorithm in Figure 13. Formally, the value C_{ij} is now defined as the sum of edge weights between symbol group i and symbol group j.

7.3 Profile-Driven Instruction Duplication

Pushing load/store instruction to predecessor/successor basic blocks can be more effective with profiling information. Instruction motion focuses on hot paths to maximally combine load/stores that are most frequently executed. For load instructions, the restriction of pushing them only on the reverse EBB can be released. When pushing an instruction to one of the predecessors that have multiple successors, if the execution frequency of that path is higher than all the other paths from that predecessor, we will still push it. For instance, in Figure 14, if basic block 2 has another successor called basic block 4, but the path from basic block 2 to basic block 3 is hotter than the one from basic block 2 to basic block 4, we can still push the LD instruction up to basic block 2 as long as it can be merged with another instruction. For the algorithm in Figure 15, the building of t_rebb (line 3) becomes to build a tree rooted at b that can include branches that are hotter than edges to other successors, i.e., we will include basic block 2 in Figure 14, if the path from it to basic block 3 is hotter than the one to basic block 4. Similarly, for store instructions, we can release the restriction of pushing them only on EBB by examining all successors with hot edges. Considering the execution frequency of each path gives us more chance to push load/stores aggressively outside the basic block boundary.

7.4 Profile-Driven Variable Duplication

For variable duplication, the profitability judgments are not simply based on the change of total number of instructions. If two load/store instructions with high execution frequency are merged, adding several less executed instructions is still profitable.

Figure 21 shows an example to do variable duplication when profiling information is available. Assume that, in Figure 21A, both the path from basic block 1 to 3 and the path from basic block 1 to 4 are heavily executed, while the paths going out from basic block 2 are lightly executed. However, the variable c and d are in different banks after the previous passes. Thus, we have difficulty to

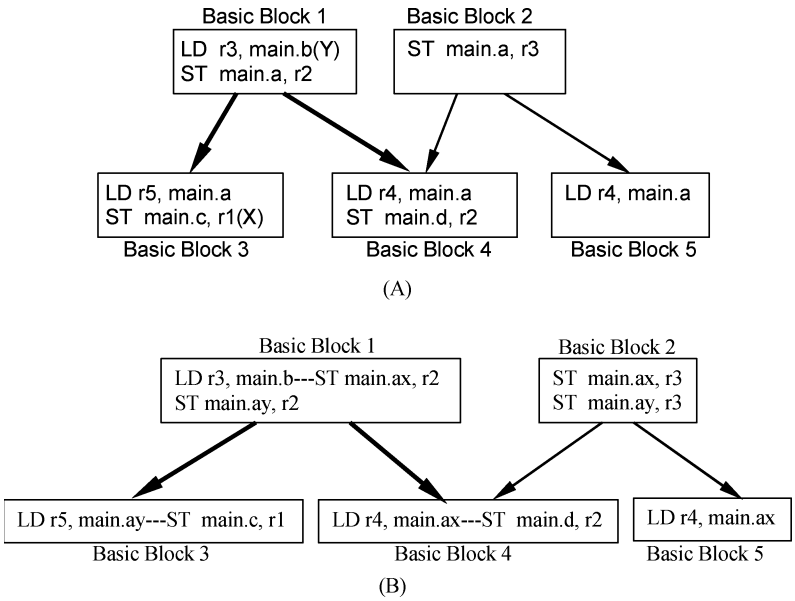

Fig. 21. Variable duplication with profiling information.

merge a with both c and d in basic block 3 and 4. In Figure 21B, we duplicate variable a to both X and Y memory banks. Therefore, two instructions are added to basic block 1 and 2. Although the execution path from basic block 2 cannot get any benefits (even worse for the path from basic block 2 to basic block 5), the hot path from basic block 1 executes faster. Thus, in this case variable duplication is profitable. The algorithm shown in Figure 17 is changed as instead of summing the number of instructions merged/inserted, we count the execution frequency, i.e., each instruction merged/inserted is weighted by the execution frequency of this instruction. The inequality in line 15 can still be used to guide variable duplication.

7.5 Profile-Driven Web Splitting and Register Reclamation

Web splitting can also benefit from profiling information in the phase to determine places to cut the web and which memory bank to put that part of the web. Profiling information tells which parts of the web (load/store instructions) are hotly executed, and, therefore, should be merged with best effort. The method we follow is similar to the one without profiling information. In Figure 19, we give each load/store instruction a weight, i.e., the execution frequency of that instruction. The algorithm in Figure 19 is modified in several places to factor in the weights. In line 13 and line 16, instead of increasing num_x_prefer or num_x_prefer by 1, we add the weight of the newly joined load/store instruction. Besides, in line 34, when calculating the cost and gain, we add up the weights for each instruction that is merged or inserted. The other parts of the algorithm can remain the same as in Figure 19.

Table I. Benchmark Properties

	Total Instr.	Total LD/ST	#LD	#ST	#var sym	#add sym	Alias found (single reference)	% Alias found/total load store
biquad_N_sections	157	26	15	11	10	16	26	100
complex_update	118	26	16	10	6	20	20	76.92
n_complex_update	295	67	26	41	35	32	53	79.10
n_real_update	181	18	5	13	6	12	16	88.89
gsm	3992	401	225	176	48	353	397	99.00
g721	1847	191	113	78	66	125	191	100
rawcaudio	204	21	13	8	11	10	21	100
rawdaudio	190	26	15	11	17	9	26	100
bzip2	12256	1432	840	592	424	1008	1398	97.63
vpr	46954	2927	1763	1164	903	2024	2872	98.12
epic	5988	884	634	250	687	197	846	95.7
gzip	13127	1801	1286	515	1438	363	1706	94.73
mcf	4422	571	410	161	272	299	512	89.67
Average								93.83

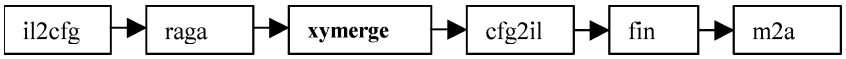

Fig. 22. Processing pipeline.

8. PERFORMANCE EVALUATION

The algorithm has been implemented with the SUIF/Mach-SUIF compiler [Mach-SUIF 2000] targeted to the SONY pDSP architecture. A separate pass called xymerge is added after the raga pass—a standard pass in mach-SUIF which runs Appel and George's iterated register allocation algorithm [1996]. Figure 22 shows the passes in mach-SUIF. The pass il2cfg changes the instruction list to a control-flow graph. We then do the register allocation (raga) and xymerge. The cfg is changed back to instruction list (cfg2il) and finally translated to assembly code (m2a).

Benchmark programs are selected from DSPStone, Mediabench, and SPEC2000. Among them, all DSP benchmarks are filtering applications based on a variety of algorithms; gsm is a speech transcoding program; g721 performs voice compression; rawcaudio and rawdaudio are encoding and decoding programs for adaptive differential modulation; bzip2 and gzip are two data-compression algorithms; vpr is a placement and routing program for IC chips; epic is an image data-compression algorithm, and mcf does combinatorial optimization for vehicle scheduling. These benchmark programs contain many array operations and indirect addressing; thus, an alias analysis pass is performed in order to determine the symbols associated with the memory locations pointed by the address registers. We have tried both simulated annealing (SA Figure 12) and the greedy heuristic algorithm (Figure 13) at point A (Figure 3) to color the MSG. We also report the results at point B, C, and D.

Table I lists the properties of the 13 benchmark programs after the alias analysis pass. The second column shows the total number of load/store instructions in the program, which is after procedure cloning and inlining. For the

Table II. Classification of Generated Parallel Load/Stores

	Merge type				Failed merge		
	Without dup	With dup	LDLD	LDST	ISA	STST	Resolved
biquad_N_sections	4	1	2	3	0	1	2
complex_update	6	2	3	5	3	3	2
n_complex_update	9	3	4	8	2	3	3
n_real_update	0	1	0	1	0	2	0
gsm	73	14	32	55	4	10	9
g721	17	4	6	15	4	4	4
rawcaudio	5	2	2	5	1	1	1
rawdaudio	7	2	4	5	3	5	3
bzip2	177	43	78	142	10	13	12
vpr	394	103	154	343	26	30	45
epic	122	43	60	107	15	21	30
gzip	237	94	97	234	14	32	23
mcf	95	31	51	75	5	15	19

benchmark vpr, the number of memory operation instructions is about 3K. Columns 3–6 list the distribution of those load/store instructions. The address symbols (column 6) are those actual variable symbols that need to be determined through alias analysis. Column 7 shows the determined symbols (the symbols that do not have multiple aliases) after alias analysis, and column 8 shows the percentage of determined symbols over total number of load/store. On average, about 94% of load/stores can be determined after alias analysis. In this work, we have implemented a flow-sensitive alias analysis pass [Aho et al. 1986] on the generated code. It is very effective. 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 greatly impacts the performance of the xymerge pass, because undecided load/store instructions not only reduce the merge rate, but also creates an 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 II casts some insights into where the merges happen and several reasons that cause merge failure. The results are taken from point D. 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 numbers of LDs and STs are close to each other, so LDST should outnumber LDLD. All ST instructions must be combined in the form of LDST because of the lack of STST instruction. Columns 6–7 list the reasons for failed merges. ISA failure is specific for the pDSP architecture because of the address register limitation in parallel memory instructions, which has been listed in Section 2.2. Some of the failures happen when two ST instructions need to be combined (i.e., PSTST is not supported by ISA). This type of failure takes a big share among all failures. This is understandable, because STST instructions should occur frequently. The last column shows the frequency of resolution invoked during the compilation. This number is very low compared with the number of load/stores and merged pairs.

Table III. MSG Properties

	Grp#	Node#	Edge#	Max web size	Avg. web size
biquad_N_sections	8	26	12	5	3.3
complex_update	11	26	27	8	2.4
n_complex_update	21	67	49	6	3.2
n_real_update	7	18	7	6	2.6
gsm	129	401	486	20	3.1
g721	32	191	130	11	6.0
rawcaudio	10	21	25	10	2.1
rawdaudio	9	26	31	14	2.9
bzip2	395	1,432	1,109	15	3.6
vpr	937	2,927	3,104	15	3.1
epic	320	884	1,182	18	2.8
gzip	574	1,801	1,771	13	3.1
mcf	194	571	619	16	2.9

Table III 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 strongly related to the number of nodes in the graph and it also varies greatly among different benchmark programs. Thus, we can conclude that 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 5 is the maximal size of the webs. We observe that webs can be pretty big in most of the benchmarks. Column 6 shows the average size of web in each program. They are typically small, which justifies our conclusion that splitting the web is not profitable, in most cases.

Table IV lists the execution time for the 13 benchmarks in different optimizing phases. For each benchmark program, we collect six different number of execution cycles. The second column represents the original execution cycles. The third and fourth column are the cycle counts for the greedy heuristic algorithm and SA after the baseline stage at point A. The next three columns show the cycle counts at point B, C, and D (all with the greedy heuristic) to incrementally show the improvements. On average, greedy(A) and SA(A) achieve speedup of 10.4 and 12.8%, while the speedups for point B, C, and D are 11.5, 12.2, and 12.75%, respectively. Without the second stage, SA performs better than greedy (A) at the cost of tremendous amount of compilation time (we will show this in Table V). However, with other optimizing steps in the second stage, the greedy heuristic can approach the SA algorithm at point A. With all optimization steps enabled, the speedup of the greedy heuristic algorithm is only 0.36% less than the SA algorithm at point A. For 8 out of 13 benchmark programs, the SA algorithm is worse than the greedy algorithm with all the optimization steps.

Table V shows the compilation time for various methods. Columns 2 to 7 correspond to the 6 columns in Table IV. The greedy heuristic at point A is much faster than the SA algorithm at point A, since SA is run longer enough to approach the optimal solution. The speedups range from 11 to 183 with and

Table IV. Comparison of Execution Cycles

	Original	Greedy (A)	SA (A)	LD/ST replica (B)	Var replica (C)	Web split (D)	(%) Web split(D)/orig	(%) Web split(D)/SA
biquad_N_sections	949,751	875,101	866,458	850,861	839,119	839,028	11.66	3.17
complex_update	16,177	15,598	15,656	15,325	15,313	15,313	5.34	2.19
n_complex_update	3,119,310	3,003,584	2,959,289	2,993,973	2,982,296	2,956,350	5.22	0.10
n_real_update	1,233,199	1,161,427	1,147,368	1,142,380	1,133,012	1,119,756	9.20	2.41
gsm	76,521,677	54,789,521	51,499,089	53,929,326	53,465,533	53,123,354	30.58	−3.15
g721	52,011,133	41,385,259	37,187,960	41,372,843	40,847,408	40,671,765	21.80	−9.37
rawcaudio	104,900,449	90,529,087	90,717,908	89,370,315	88,020,823	87,633,531	16.46	3.40
rawdaudio	46,774,338	41,863,033	39,613,187	41,507,197	41,141,934	41,136,997	12.05	−3.85
bzip2	209,449,714	178,932,892	168,753,635	175,783,673	174,236,777	173,365,593	17.23	−2.73
vpr	89,316,526	84,359,459	82,798,809	83,423,069	82,805,738	81,596,775	8.64	1.45
epic	1,372,506	1,216,040	1,183,815	1,196,705	1,181,148	1,175,715	14.34	0.68
gzip	47,299,691	44,887,407	42,732,811	43,594,650	43,298,206	43,038,417	9.01	−0.72
mcf	7,600,580	7,406,765	7,403,062	7,368,991	7,367,517	7,281,317	4.20	1.64
Average							12.75	−0.37

Table V. Comparison of Compilation Time (s)

	Original	Greedy (at A)	SA	LD/ST replica (at B)	Var replica (at C)	Web split (at D)	%Speedup (after LD/ST replica/SA)	%Speedup (after web split/SA)
biquad_N_sections	7.75	8.92	292.37	11.23	17.85	22.15	3,170.74	1,216.83
complex_update	4.99	6.14	1,125.82	6.85	10.75	16.91	18,358.85	6,556.92
n_complex_update	10.10	12.33	511.32	14.73	21.60	26.34	4,055.24	1,840.13
n_real_update	6.34	6.91	294.65	7.61	12.49	17.62	4,174.10	1,575.58
gsm	391.03	430.47	8,718.12	556.03	902.50	1,344.17	1,927.39	548.55
g721	250.16	295.29	4,060.73	412.85	614.94	889.70	1,276.63	356.45
rawcaudio	15.81	18.32	675.26	21.42	35.36	50.07	3,587.16	1,247.55
rawdaudio	11.43	14.64	1,984.41	20.04	31.72	47.36	13,489.11	4,088.33
bzip2	1,378.82	1,745.41	32,765.78	2,234.98	3,331.24	5,304.27	1,777.28	517.75
vpr	2,237.76	2,690.81	57,196.59	3,157.40	4,386.57	5,902.13	2,025.65	869.09
epic	1,046.64	1,330.05	30,172.11	1,681.12	2,629.27	3,182.99	2,168.57	847.91
gzip	1,711.00	2,210.38	48,236.92	3,085.16	5,232.43	7,119.24	2,082.67	577.55
mcf	582.64	829.84	10,381.35	986.04	1,467.22	2,148.02	1,150.72	383.28
Average							4,557.24	1,586.61

average of 45. The last column shows the compilation time comparison for SA at point A with the greedy heuristic at point D. The speedups are still huge, which range from 3.8 to 65 with an average of 15.9.

Tables V and VI tell us that the greedy heuristic algorithm can greatly shorten the compilation time. If it is combined with other optimizations, the performance is comparable to the SA algorithm, which consumes huge amount of compilation time.

Table VI shows the code and data segment size comparison for different methods. The third and fourth column are sizes for the greedy heuristic and the SA at point A. Both of them reduce the size (on average, 4.4% for greedy (A) and 5.3% for SA(A)). Inevitably, the three optimization steps in the iteration stage increase the code and data segment size. However, the size increases have been properly controlled. On average, the size increases at point B, C, and D are 6.13, 14.20, and 12.58%, respectively. Note that in the last step, where web splitting actually reduces the size, both variable and instruction duplication add more space requirements.

Upon this point, we have seen results showing the incremental effect of the three steps in the iteration stage. It is also important to look at the impact on performance achieved by each individual step. However, because of the page limitation, we only report them briefly as follows. As can be perceived, the three steps have decreasing influences on the speedup. If only the second step, i.e., variable duplication is included in the iteration stage, the average speedup is 11%. While only the third step, i.e., web splitting and spill reclamation is invoked, the average speedup is 10.8%. Both of them are smaller than the speedup at point B, i.e., "step 1 only," as mentioned earlier. For code growth, "step 2 only" gives an increase of 2.8%, while "step 3 only" reduces it by 5.8%. Both are smaller than "step 1 only."

The rest of the tables and figures present the results related to profile-driven optimizations. To obtain sufficient profiling information, we conducted a series of trial runs with different inputs that are typical to the benchmark and then obtained all the results for a particular test run.

In Table VII, we list the same items as Table II. Roughly, the number of merges without duplication decreases about 5%, while the number with duplication increases about 5%. We can see the changes of the number and type of merges are not significant. Thus, profile information cannot add more merges. However, by merging instruction on the hot paths, the execution is actually accelerated.

Table VIII shows the execution cycle for the profile-driven optimizations. The speedup for greedy(A) and SA(A) are 13.6 and 14.4%, respectively. Compared with the ones without profiling information, both greedy(A) and SA(A) are somewhat improved. On average, the speedup for data collection point B, C, and D are 14.8, 15.9, and 16.7%, respectively. In short, profiling information adds about 4% speedup if all three optimization steps are enabled (at point D) compared with the one without profile information. Most importantly, the profile-driven optimization achieves faster execution speed in 12 of the 13 benchmark programs compared with the SA algorithm at point A, as shown in the last column (with an average speedup of 2.4%).

Table VI. Comparison of Code and Data Segment Size

	Original	Greedy (A)	SA (A)	LD/ST replica (B)	B/orig (%)	Value duplica (C)	C/orig (%)	Web split (D)	D/orig (%)
biquad_N_sections	157	151	147	167	6.37	180	14.65	180	14.65
complex_update	118	106	102	135	14.41	148	25.42	148	25.42
n_complex_update	295	273	271	321	8.81	347	17.63	334	13.22
n_real_update	181	179	179	197	8.84	209	15.47	198	9.39
gsm	3,992	3,843	3,818	4,053	1.53	4284	7.31	4,239	6.19
g721	1,847	1,813	1,805	1,909	3.36	2074	12.29	2,014	9.04
rawcaudio	204	193	190	218	6.86	234	14.71	227	11.27
rawdaudio	190	174	172	209	10.00	221	16.32	221	16.32
bzip2	12,256	12,102	12,073	12,452	1.60	13150	7.29	13,120	7.05
vpr	46,954	46,280	46,194	48,338	2.95	52214	11.20	52,167	11.10
epic	5,988	5,329	5,317	6,231	4.06	6766	12.99	6,713	12.11
gzip	13,127	12,986	12,964	13,531	3.08	14570	10.99	14,512	10.55
mcf	4,422	4,390	4,377	4,770	7.87	5230	18.27	5,185	17.25
Average					6.13		14.20		12.58

Table VII. Classification of Generated Parallel Load/stores—Profile-Driven

	Merge type				Failed merge		
	Without dup	With dup	LDLD	LDST	ISA	STST	Resolved
biquad_N_sections	5	1	2	4	0	1	2
complex_update	5	2	2	5	2	4	3
n_complex_update	10	2	4	8	4	3	2
n_real_update	1	1	0	2	1	3	1
gsm	70	17	30	57	5	13	10
g721	17	5	8	14	4	3	5
rawcaudio	4	3	3	4	1	2	3
rawdaudio	7	1	3	5	4	7	4
bzip2	169	44	69	144	8	15	14
vpr	387	104	169	322	25	27	43
epic	126	44	60	110	17	23	34
gzip	210	101	134	177	11	34	26
mcf	94	35	48	81	6	14	16

Table IX does the comparison of compilation time for various methods with profile-driven information. Compared with the ones without profile-driven information, most results are very close. This agrees with our complexity analysis that profile-driven optimizations do not increase the time complexity for the previous optimizations. Table X compares data and code segment size for profile-driven optimizations. In all comparisons, the differences are less than 1%. In general, profile-driven optimizations can reduce sizes slightly; however, the differences are almost trivial.

If we look at the contribution of each step individually, "step 2 only" achieves a speedup of 14.48% and "step 3 only" gives 14.25%. Both are about 3% higher than nonprofile-driven ones, but less than "step 1 only." On the other hand, code growths for individual steps are quite close to those without profile information.

Finally, Table XI gives results for multiple-alias cleanup. The second column is the number of multiple-aliased memory access instructions. Among the aliased undetermined memory access instructions, most of them are found to be multiple aliased, i.e., only a small number of them are totally undecidable. The third column shows, among the multiple-aliased ones, the average number of aliases. This number is typically from 2 to 3. In other words, most of them are aliased with two variables. The last two columns present the number of merged instructions for both nonprofile-driven and profile-driven cases after they are determined to reference only a single bank. About 28% multiple-aliased instructions can be merged successfully within the last stage. About 30% multiple-aliased instructions can be merged when profile information is enabled. Since the number of multiple-aliased instructions is small, this stage actually has a very limited impact on the overall performance.

9. RELATED WORKS

The work on storage allocation to optimize load/stores can be classified into two categories: (1) SIMD type instructions, which allow fetching adjacent memory values into register pairs and (2) memory bank architectures (as in this paper), which allow parallel load/stores from different

Table VIII. Comparison of Execution Cycles—Profile-Driven

	Original	Greedy(A)	SA (A)	LD/ST replica (B)	Var replica (C)	Web split (D)	(%) Web split (D)/Orig	(%) Web split (D)/SA
biquad_N_sections	949751	840112	854629	816841	805568	805480	15.19	5.75
complex_update	16177	14931	15463	14670	14658	14659	9.39	5.20
n_complex_update	3119310	2909785	2901772	2900474	2860157	2859528	8.33	1.46
n_real_update	1233199	1137397	1113789	1118744	1109570	1107491	10.19	0.57
gsm	76521677	52275887	50813533	51455156	51012642	50686161	33.76	0.25
g721	52011133	39436800	36343719	39424969	38924271	38266451	26.43	−5.29
rawcaudio	104900449	86756830	89671931	85646343	84353083	83054046	20.83	7.38
rawdaudio	46774338	39896350	38865369	39557231	39209127	38601386	17.47	0.68
bzip2	209449714	170852104	165824072	167845107	166368070	163290260	22.04	1.53
vpr	89316526	81692857	80689261	80786066	80188249	79707119	10.76	1.22
epic	1372506	1191347	1155707	1174907	1149998	1116304	18.67	3.41
gzip	47299691	43416806	42296765	43073813	41906513	41399444	12.47	2.12
mcf	7600580	7074823	7266046	6922715	6796721	6734871	11.39	7.31
Average							16.69	2.43

Table IX. Comparison of Compilation Time (s)—Profile-Driven

	Original	Greedy(at A)	SA	LD/ST replica (at B)	Var replica (at C)	Web split (at D)	%Speedup (after LD/ST replica/SA)	%Speedup (after web split/SA)
biquad_N_sections	7.75	9.03	311.75	11.19	18.34	23.52	3,352.38	1,226.56
complex_update	4.99	6.22	1,203.99	7.35	11.41	18.62	19,256.75	6,366.11
n_complex_update	10.14	12.49	547.09	15.45	23.6	28.30	4,280.22	1,833.18
n_real_update	6.34	7.54	310.34	8.38	14.09	20.44	4,015.92	1,418.32
gsm	391	462.78	9,317.78	547.78	872.66	1,256.11	1,912.48	641.76
g721	249.71	305.49	4,361.06	445.15	694.21	1,039.09	1,329.86	319.71
rawcaudio	15.82	20.71	719.75	25.06	40.11	55.18	3,377.05	1,203.89
rawdaudio	11.18	16.46	2,104.01	23.64	38.35	59.23	12,651.58	3,454.07
bzip2	1,378.82	1,789.37	35,125.97	2,237.64	3,200.95	4,904.46	1,862.76	616.23
vpr	2,237.76	2,712.81	60,197.27	3,264.60	4,731.38	6,649.95	2,119.41	805.23
epic	1,046.64	1,367.47	31,171.83	1,770.19	2,680.07	3,378.54	2,178.65	822.66
gzip	1,711.40	2,303.31	54,236.57	3,330.14	5,448.11	7,085.81	2,255.06	665.43
mcf	582.64	861.96	11,080.66	1,067.16	1,630.61	2,485.06	1,185.50	345.90
Average							4,598.27	1,516.85

Table X. Comparison of Code and Data Segment Size—Profile-Driven

	Original	Greedy (A)	SA (A)	LD/ST replica (B)	B/orig (%)	Value duplica (C)	C/orig (%)	Web split (D)	D/orig (%)
biquad_N_sections	157	149	148	163	3.82	181	15.29	181	15.29
complex_update	118	107	101	138	16.95	144	22.03	144	22.03
n_complex_update	295	277	270	331	12.20	350	18.64	350	18.64
n_real_update	181	176	176	191	5.52	216	19.34	212	17.13
gsm	3,992	3,800	3,771	4,081	2.23	4,197	5.14	4,123	3.28
g721	1,847	1,808	1,794	1,898	2.76	2,065	11.80	2,014	9.04
rawcaudio	204	195	192	220	7.84	240	17.65	230	12.75
rawdaudio	190	171	168	202	6.32	213	12.11	201	5.79
bzip2	12,256	11,971	12,010	12,343	0.71	12,793	4.38	12,720	3.79
vpr	46,954	46,862	46,739	49,555	5.54	52,701	12.24	52,610	12.05
epic	5,988	5,259	5,411	6,073	1.42	6,621	10.57	6,611	10.40
gzip	13,127	13,195	12,866	13,765	4.86	14,791	12.68	14,750	12.36
mcf	4,422	4,372	4,462	4,701	6.31	5,178	17.10	5,157	16.62
Average					5.88		13.77		12.24

Table XI. Results for Multiple Alias Cleanup

	# of Multiple aliased	Avg. # of variables aliased with	# of cleanups	# of cleanups for profile-driven
biquad_N_sections	0	0	0	0
complex_update	5	2.2	2	3
n_complex_update	10	3.1	4	3
n_real_update	2	2.5	0	0
gsm	4	3.3	1	1
g721	0	0	0	0
Rawcaudio	0	0	0	0
Rawdaudio	0	0	0	0
bzip2	33	2.4	6	8
Vpr	45	2.1	12	11
Epic	33	2.6	14	16
gzip	87	2.4	23	23
mcf	53	2.0	18	20
Average		2.5		

memory banks. The ISA definition of the former is quite different from the latter, which is the focus of this paper.

(1) [Davidson 1994] talks about the coalescence of narrow 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) [Cooper 1998] proposes a hardware solution to add compiler-controlled memory (CCM), which can be thought of as the on-chip memory on some new DSP chips. Their focus is to spill values quickly to the on-chip memory instead of the cache. Such 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 X–Y memory bank model 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 compiler optimizations for such architectures. [Sudarsanam and Malik 2000] discuss this problem and attempt 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 [Sudarsanam and Malik 2000]. The compilation time of their approach is very high, making it unlikely for a practical system. Their main goal is to reduce code size in contrast to ours, which is to increase the execution 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. Although it is still

unclear if a combined approach generates better results, the high compilation cost already poses a great obstacle for such approach. Moreover, a postpass approach can greatly benefit from an excellent register allocator to generate 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 mach-SUIF [George and Appel 1996] and then concentrate on placement of values. In contrast, we perform several optimizations, such as increasing the range of motion, conflict resolution, and instruction/variable duplication.

We would like to compare our approach with Leupers and Kotte [2001] and Sudarsanam and Malik [2000]. 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 time and small code growth. This is achieved by our framework, which systematically attempts to parallelize load/stores by enhancing range of motion followed by other optimizations. In addition, with profile information, our approach further improves the program execution speed. Leuper and Kotte [2001] proposed a two-pass approach. The first pass determines the exact set of memory accesses. The IR is then annotated with partitioning information and passed to the back-end again. Variable partitioning is modeled as ILP based on an *interference graph*. However, they do not consider the movable boundaries for load/stores, which unconstraints the range of motion. In any postpass approach, the largest constraint is the range of motion, because of register dependencies. However, the postpass solution is attractive in that 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 increase opportunities for combining load/stores. Another difference is that they use ILP approaches to solve the 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 too 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. They also do not perform other optimizations like instruction or variable duplication.

Saghir, Chow, and Lee [1996] present a postpass approach called Compaction Based (CB) data partitioning and partial data duplication. The method first attempts to exploit parallelism among load/store instructions and allocate them to memory accordingly. Partial data duplication is then used to augment CB partitioning in instances where parallel accesses are made to elements of the same array. To minimize the impact on code size, instruction duplication is only applied to ensure the integrity of duplicated data (i.e., only store operations that access duplicated data are duplicated). The net impact of using partial data duplication is a 1% increase in code size, while CB partitioning reduces code growth by 1%. They have shown speedups of 3 to 15% on full application benchmarks, with one benchmark achieving a speedup of 34% because of partial data duplication. In contrast, this paper proposes more aggressive application of variable and instruction duplication, as well as web splitting, etc., which

expose more potential to speedup the execution. We also organize these passes into a framework with two-level iterations to exploit more performance gains. In addition, we propose the notion of pseudofixed boundaries to tackle the problem of register dependencies encountered in postpass approaches. We do a conflict resolution to maintain legality after motion takes place without degrading the merged load/store pairs.

Recently, Cho, Paek, and Whalley [2002] study memory and register assignments for nonorthogonal architectures. They have used different approaches like MST and graph coloring to assign memory banks to variables. A register class allocation phase is also 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 which adopts an iterative approach of instruction duplication followed by variable duplication and web splitting.

Other earlier work [Powell et al. 1992] on this problem adopts very simplistic approach of allocating X/Y memory on an alternating basis without any analysis.

10. CONCLUSION AND FUTURE WORKS

This paper proposes a framework for analyzing load/stores instructions on embedded processors and for moving them to combine into parallel load stores. An important contribution of this paper is to propose the notion of pseudofixed boundaries to enhance the range of motion and then perform conflict resolution to preserve legality of code while keeping most load/store pairs. We also undertake rematerialization to free registers to enhance boundary of motion. We first replicate the instructions and variables and then split the webs and iterate until no profitable mergers result. We show that by undertaking such an approach, our solution comes close to exhaustive one with much less compilation time. We use the freed time to perform optimizations, which lead to solutions better than those from an exhaustive approach (without optimization) in most of the benchmarks. The conclusion of this work is that by systematically enhancing the range of motion of instructions and by undertaking duplication of instructions and variables along with web splitting, one can generate code quality comparable or superior to that generated by exhaustive methods. Furthermore, our optimization steps can be guided by profiling information to boost run-time performance significantly while keeping the data and code growth and compilation time almost unaffected. In summary, our approach can achieve a speedup of about 13% without profiling information and 17% with profiling information for the 13 benchmark programs. This result is quite close (better if with profiling information) to the exhaustive algorithm without the iteration stage, but the compilation time is greatly shortened. The code and data segment growth is properly controlled—within 13% for both cases.

Although this work does intraprocedural load/store merge quite successfully, it is still insufficient to analyze and catch the interprocedural opportunities aggressively. For future improvements, we would expect to generalize some of the

approaches to capture parameters across function boundaries. For global variables, our approach assumes they can only be put in a single location. More studies can be done on global variable duplication. Furthermore, a few modern DSP cores now have more than two memory banks available for parallel accesses. It would be interesting to extend our techniques to support more memory banks.

ACKNOWLEDGMENTS

We would like to thank Greenhill Software INC. for providing the software, documents, and developing experiences on the pDSP processor. Special thanks go to Mr. Greenland and Mr. Reed for their help throughout our project.

A preliminary version of this paper appeared in the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), 2002 [Zhuang et al. 2002].

REFERENCES

AARTS, E. AND KORST, K. 1989. Simulated annealing and Boltzmann Machines, Courier Int'l.

AHO, A. V., SETHI, R. AND ULLMAN, J. D. 1986. *Compilers Principles, Techniques and Tools*, Addison-Wesley, Reading, MA.

BRIGGS, P., COOPER, K., AND TORCZON, L. 1994. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*.

CHAITIN, G.J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. 1981. Register allocation via coloring. *Computer Language*, 6, 47–57.

CHO, J., PAEK, Y., AND WHALLEY, D. 2002. Register and memory assignment for non-orthogonal architectures via graph coloring and MST algorithms. *Proc. of LCTES'02* (June), 130–138.

COOPER, K. D. AND HARVEY, T. J. 1998. Compiler-controlled memory. *In 8th ASPLOS* (Oct.)

COOPER, K. D. AND MCINTOSH, N. 1999. Enhanced code compression for embedded RISC processors. *Proc. SIGPLAN '1999 Conf. Programming Language Design and Implementation* (May), 139–149.

DAVIDSON, J. W. AND JINTURKAR, S. 1994. Memory access coalescing: A technique for eliminating redundant memory accesses. *Proc. SIGPLAN '94 Conf. Programming Language Design and Implementation* (June) 186–195.

GEORGE, L. AND APPEL, A. W. 1996. Iterated register coalescing. In *Proc. SIGPLAN '96 Conf. Programming Language Design and Implementation*.

GROSS, J. AND YELLEN, J. 1999. *Graph theory and its applications*. CRC Press. Boca Raton, FL.

KNOOP, J., RUTHING, O., AND STEFFEN, B. 1992. Lazy code motion, *Proc. SIGPLAN '1992 Conf. Programming Language Design and Implementation* (July).

LEUPERS, R. AND KOTTE, D. 2001. Variable partitioning for dual memory bank DSPs. *ICASSP* (May).

Mach-SUIF Backend Compiler, 2000. *The Machine-SUIF 2.1 compiler documentation set*. Harvard University, Sept. http://ececs.harvard.edu/hube/research/machsuif.html.

PAPADIMITRIOU, C.H. AND STEIGLITZ, K. 1998. *Combinatorial optimization Algorithms and Complexity*, Dover Publications, 1998.

POWELL, B., LEE, E.A., AND NEWMAN, W.C. 1992. Direct synthesis of optimized DSP assembly code from signal flow block diagrams. *Proceedings International Conference on Acoustics, Speech, and Signal Processing*. 553–556.

SAGHIR, M. A. R., CHOW, P., AND LEE, C. G. 1996. Exploiting dual data-memory banks in digital signal processors. *Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operation Systems*, 234–243.

Stanford SUIF Compiler Infrastructure, 2000. *The SUIF 2 Compiler Documentation Set*, Stanford University, Sep. http://suif.stanford.edu/suif/index.html.

SUDARSANAM, A. AND MALIK, S. 2000. Simultaneous reference allocation in code generation for dual data memory bank ASIPs. *ACM Trans. on Design Automation of Electronic Systems*, Vol. 5, 242–264 (Apr.).

ZHUANG, X., PANDE, S., AND GREENLAND J. S. JR. 2002. A framework for parallelizing load/stores on embedded processors. In *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, 68–70 (Sep.).

Received March 2003; revised November 2004; accepted November 2005