Using Multiple Memory Access Instructions for Reducing Code Size

Neil Johnson and Alan Mycroft

Computer Laboratory, University of Cambridge
William Gates Building, JJ Thompson Avenue,
Cambridge, CB3 0FD, UK
`{Neil.Johnson,Alan.Mycroft}@cl.cam.ac.uk`

Abstract. An important issue in embedded systems design is the size of programs. As computing devices decrease in size, yet with more and more functions, better code size optimizations are in greater demand. For an embedded RISC processor, where the use of compact instructions (*e.g.*, the ARM Thumb) restricts the number of accessible registers at the expense of a potential increase in spill code, a significant proportion of instructions load or store to memory.
In this paper we present a new technique which both identifies sequences of single load and store instructions for combining into multiple load and store instructions, and guides the layout of function stack frames, global storage and register allocation, previously only seemingly done by opportunistic optimization. We implement this in our SolveMMA algorithm, similar to Liao's *Simple Offset Assignment* algorithm.
We implement our algorithm within the Value State Dependence Graph framework, describe how the algorithm is specialized for specific processors, and use the ARM Thumb as a concrete example for analysis.

1 Introduction

With increasing numbers of compact and mobile computing devices, such as mobile phones, wrist-watches, and automotive systems, comes a growing need for more effective code-compacting optimizers. One potentially profitable route to compact code is through the effective use of Multiple Memory Access (MMA) instructions. These include multiple loads and stores where a set of registers (typically defined by some range or bitmap representation) are loaded from, or stored to, successive words in memory. Their compactness comes from expressing a (potentially large) set of registers with (comparatively) few instruction-word bits; for example, the ARM `LDM` instruction uses only sixteen bits to encode up to sixteen register loads (*cf.* 512 bits without the use of the `LDM` instruction).

The ARM Thumb [15] executes a compact version of the ARM instructions. A hardware translator expands, at runtime, the 16-bit Thumb instructions into 32-bit ARM instructions. A similar approach is taken in the MIPS16 embedded processor [9]. A disadvantage of this approach is that fewer instruction bits are available to specify registers. This restriction artificially starves the register allocator, resulting in more register spill code (providing more potential sources of MMA optimization, which is poorly done in existing compilers).

The *"load-store"* nature of a RISC architecture also gives rise to many explicit memory access instructions. In contrast, a CISC machine with memory-access-with-operation instructions can achieve better code size by combining a load or store operation (especially where the address offset is small) with an arithmetic operation (*e.g.* "`add EAX,DS:ESI[EBX*4]+Offset`" on the Intel x86 [1]). Restricting the number of accessible registers, as described above, increases the number of memory access instructions: the *adpcm* benchmark from MediaBench [11] generates approximately 35% more memory access instructions for the ARM Thumb than for the ARM.

However, the state of the art in commercial compilers appears[1] to be based on opportunistic peephole-style optimization. The GCC compiler also takes an ad-hoc approach to MMA optimization[2].

This paper explores the use of MMA instructions as a means of compacting code. Our SolveMMA algorithm identifies profitable memory access sequences for combining into MMA instructions, and selects stack frame layouts that facilitate multiple reads and writes of local variables. Interestingly, it turns out that array accesses and local variable accesses are best treated separately.

We describe SolveMMA as applied to the Control Flow Graph, and then show how the less-constrained Value State Dependence Graph provides greater opportunities for merging multiple load and store instructions. Finally, we specialize this to the ARM Thumb MMA instructions, and apply our algorithm to a selection of embedded software benchmarks.

1.1 Related Work

As remarked above there is no directly related work on MMA optimization for general purpose registers. However, a related area of research is that of optimizing address computation code for Digital Signal Processors (DSP).

For architectures with no register-plus-offset addressing modes, such as many DSPs, over half of the instructions in a typical program are spent computing addresses and accessing memory [16]. The problem of generating optimal address-computing code has been formulated as the Simple Offset Assignment (SOA) problem, first studied by Bartley [4] and Liao *et al* [13], the latter formulating the SOA problem for a single address register, and then extended to the General Offset Assignment (GOA) problem for k address registers. Liao *et al* also show that SOA (and GOA) are NP-hard, reducing the problem to the Hamiltonian path problem. Their approximating heuristic is similar to Kruskal's maximum spanning tree algorithm.

Rao and Pande [14] generalize SOA for optimizing expression trees to minimize address computation code. An improved heuristic for SOA and GOA was

[1] Through literature and private communication with industrial compiler developers.
[2] For example, the GCC compiler uses *hard registers* to enforce a particular register assignment within the RTL intermediate form.

proposed by Leupers and Marwedel [12] with a tie-breaking heuristic for SOA and a variable partitioning strategy for GOA.

DSP-like architectures (*e.g.*, Intel MMX, PowerPC AltiVec and Sun VIS) include MMA-like block loads and stores. However, these instructions are limited to fixed block loads and stores to special data or vector registers, not general purpose registers. A related area to MMA optimization is that of *SIMD Within A Register* (SWAR) [6]. The work presented here considers only word-sized variables; applying the same algorithm to sub-word-sized variables would achieve additional reductions in code size (*e.g.*, combining four byte-sized loads into a single word load).

Koseki *et al* consider the problem of colouring a CFG (where the order of instructions is fixed) in the presence of instructions which have particular preferences for register assignment [10]. They suggest using these preferences to guide register assignment to enable the use of MMA instructions. We differ from their work in two ways: (1) because the VSDG underspecifies the ordering of instructions in a graph[3] we can consider combining loads and stores that are not adjacent to each other in the CFG into provisional MMA nodes; and (2) we use the Access Graph during target code generation to bias the layout of the stack frame for spilled and local variables.

All of the above approaches have been applied to the Control Flow Graph [3], which precisely specifies the sequence of memory access instructions. By contrast, the Value State Dependence Graph [8] under-specifies the ordering of most operations, only specifying *sufficient* partial ordering to maintain the I/O semantics (memory reads and writes) of the original program, giving more opportunities for optimization.

1.2 Paper Structure

The rest of this paper is structured as follows. In Section 2 we summarize Liao *et al*'s SOA algorithm. Section 3 describes our new SolveMMA algorithm in the context of the Control Flow Graph, with Section 4 enhancing the MMA algorithm for the Value State Dependence Graph. In Section 6 we apply SolveMMA, implemented within our VSDG compiler framework, to a number of illustrative examples, and close with concluding remarks (Section 7).

2 Simple Offset Assignment

The Simple Offset Assignment (SOA) algorithm [13] rearranges local variables within a function's stack frame in order to minimize address computations. The target architecture is assumed to have a single address register with word-oriented auto-increment/decrement addressing modes.

[3] We showed previously [8] that the special case of a VSDG with *enough* serializing edges to enforce a linear order corresponds to a CFG.

The input to the algorithm is an instruction-scheduled and register-allocated program, *i.e.*, a Control Flow Graph (CFG) [3], with a fixed memory access sequence. The SOLVESOA algorithm constructs an *Access Graph* (V, E), an undirected graph with vertices V and edges E. Vertices correspond to variables, and there is an edge $e = (p, q)$ between vertices p and q with weight $w(e)$ if there are $w(e)$ adjacent accesses to variables p and q.

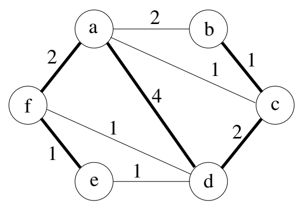
For our purposes we say that a *covering* of a graph is a subset of its edges, and recall that a path is an alternating sequence of vertices and edges, with each edge connecting its adjacent vertices, and there are no cycles in the path.

The algorithm then *covers* the Access Graph with one or more maximally-weighted disjoint *paths*. Each path specifies an ordering of the variables in the path on the stack, thereby minimizing the number of address computations through the use of auto-increment/decrement addressing. The number of address computations is then given by the sum of the weights of the uncovered edges.

Finding an optimal path covering is a formulation of the Maximum Weight Path Covering (MWPC) problem, which has been shown [13] to be in NP. Liao *et al* propose a greedy algorithm (similar to Kruskal's maximum spanning tree algorithm [2]) which iteratively chooses the edge with the greatest weight to add to the path while preserving the properties of the path (if two or more edges have the same weight the algorithm non-deterministically chooses one of them). It terminates when either no further edges can be added to the solution, or there are no more edges. An example of SOA is shown in Figure 1.

```
c = a + b;
f = d + e;
a = a + d;
c = d + a;
b = d + f + a;
```

(a)



(c)

```
a b c d e f a d a d a c d f a b
```

(b)

	b	c	d	a	f	e	

(d)

Fig. 1. Example illustrating SOA. In (a) is a short code sequence, accessing variables `a`–`f`. The access sequence (b) describes the sequence of read or write accesses to the variables (for illustration we assume variables in expressions are accessed in left-to-right order). This leads to the Access Graph shown in (c), where the edge weights correspond to the number of times a given sequence occurs in (b). We then cover the graph using the MWPC algorithm, with covered edges shown in bold. The result is the stack frame layout shown in (d).

Computing this approximate MWPC can be done in $O(|E| \log |E| + |L|)$ time, where $|E|$ is the number of edges in the Access Graph, and $|L|$ the number of variable accesses. Liao *et al* showed that for large programs the Access Graphs are generally quite sparse.

The General Offset Assignment (GOA) algorithm extends SOA to consider k address registers (consider SOA as a special case of GOA, with $k = 1$). The SolveGOA algorithm grows an Access Graph, adding vertices to it if adding an additional address register to the graph would likely contribute the greatest reduction in cost. The decision of *which* variables to address is left as a heuristic.

3 Multiple Memory Access on the Control Flow Graph

In this section we describe our SolveMMA algorithm in the context of the Control Flow Graph (CFG) [3]. Similar to Liao *et al*'s SolveSOA algorithm, we construct an *Access Graph* from the input program and then find a covering which maximizes provisional MMA instructions and biases register assignment to enable as many of these as possible to become real.

The SolveMMA algorithm is different to SolveSOA in two distinct ways. Firstly, the goal of SolveMMA is to identify groups of loads or stores that can be profitably combined into MMA instructions. And secondly, SolveMMA is applied *before* instruction scheduling, when it is easier to combine loads and stores into provisional MMA instructions and to give hints to the register assignment phase.

3.1 Generic MMA Instructions

For discussion purposes we use two generic MMA instructions—`LDM` and `STM`—for load-multiple and store-multiple respectively. The format of the `LDM` (and `STM`) instruction is:

```
LDM   Addr, { reglist }
```

where `Addr` specifies some address computation (*e.g.*, register or register+offset) containing the address from which the first word is read, and the registers to be loaded are specified in `reglist`. As in the ARM instructions, the only constraint on the list of registers is that they must be sorted in increasing numerical order. See Section 5 for specific details of addressing modes and other target-specific details.

The SolveMMA algorithm is applied twice to the flow graph. The first pass transforms global variables whose address and access ordering is fixed in the CFG[4], while the second pass transforms local and spill variables, whose addresses are not fixed, and can in more cases be re-ordered.

[4] But note Section 4 where using the VSDG instead allows only essential (programmer-required) ordering to be fixed.

3.2 Access Graph and Access Paths

We define *Access Graph* and *Access Paths* as a framework in which to formulate the problem and the SolveMMA algorithm.

Definition 1 *Let $\alpha(p)$ be the set of instructions which access variable p and $op(i)$ be the operation (load or store) of instruction i. Then the* Access Graph *is a weighted* directed *graph $AG = (V, E)$, consisting of vertices V, one for each variable, and edges $E \subseteq V \times V$ such that there exists an edge $(p, q) \in E$ between vertices p and q iff (1) $i \in \alpha(p)$ and $j \in \alpha(q)$ are in the same basic block; (2) $op(i) = op(j)$; (3) j is scheduled after i; and (4) j is not data-dependent on i.*

Vertices in V are tagged with the direction property $dir \in \{UND, HEAD, TAIL\}$ to mark *undecided*, *head* and *tail* vertices respectively. Initially, all nodes are marked UND. For some access sequences the direction of accesses must be enforced (*e.g.*, accesses to memory-mapped hardware registers), while in others the sequence is equally valid if traversed in either direction. The direction property marks explicit directions, while deferring the final direction of undefined access sequences to the covering algorithm. Our use of a directed Access Graph is in contrast to SOA's undirected Access Graph.

Definition 2 *The* Weight *$w(e)$ of an edge $e = (p, q) \in E$ is given by $w(e) = |\{(i, j) | i \in \alpha(p), j \in \alpha(q), i \text{ is before } j \text{ in the same basic block, and only arithmetic operations between them }\}|$.*

It is possible that some pairs of vertices are accessed in both combinations (p, q) and (q, p). Some of these edge pairs are likely to have differing weights, due to explicit preferences in the program (*e.g.*, a pair of stores). Such edges are called *unbalanced* edges:

Definition 3 *An* Unbalanced Edge *is an edge $e = (p, q) \in E$ where either there is an opposite edge $e' = (q, p) \in E$ such that $w(e) \neq w(e')$, or where such an opposite edge does not exist.*

The definition of the access path follows from the above definition of the Access Graph.

Definition 4 *An* Access Path *$C = (V_C \subseteq V, E_C \subseteq E)$ where $|E_C| = |V_C| - 1$, is a sequence of vertices $\{v_1, v_2, \ldots, v_m\}$ where $(v_i, v_{i+1}) \in E_C$ and no v_i appears more than once in the sequence.*

An Access Graph can be covered by two or more *disjoint* access paths:

Definition 5 *Two paths $C_A = (V_A, E_A)$ and $C_B = (V_B, E_B)$ are* disjoint *if $V_A \cap V_B = \emptyset$.*

Note that SolveSOA does not consider the type of access *operation*. In contrast, SolveMMA combines instructions of the same operation, so path covering must be sensitive to access operations. Consider the access sequence of three local variables:

$$a_r b_r c_w b_r a_r b_w$$

where the subscript r denotes read and w denotes write. It may be that a, b and c are placed in contiguous locations, but the write to c prevents the construction of a single MMA access path $\{a, b, c\}$, whereas SOA would place all three in a path. When SolveMMA is applied to the VSDG (Section 4), which would not specify the order of the reads of a and b, then either $\{a, b\}$ or $\{b, a\}$ can be normalised to the other, subject to any restrictions imposed by target-specific MMA instructions.

3.3 Construction of the Access Graph

The Access Graph is constructed in a single pass over the input program, as directed by Definition 1. The first two criteria restrict merging to access instructions of the same operation (load or store) and that both are within the same basic block (this maintains the ordering of memory accesses in the CFG). The third criterion ensures that there are no intervening load or store instructions that might interfere with one or the other instructions (*e.g.*, for two loads, an intervening store to the same address as the second load).

Vertices in the Access Graph represent variables whose address can be statically determinable and are guaranteed not to alias with any other vertex[5]. While this restriction may seem harsh, it supports global variables, local (stack-based) variables, including compiler-generated temporaries, and register-plus-constant-offset (indexed array) addressing. In these cases we can statically determine if two memory accesses do not alias to the same address.

3.4 SolveMMA and Maximum Weight Path Covering

We follow the approach taken in SolveSOA of a greedy MWPC algorithm (Section 2). The algorithm (Figure 2) is greedy in that on each iteration it selects the edge with the greatest weight (if two or more edges have the same weight the algorithm non-deterministically chooses one of them).

In the following definitions let AG be an Access Graph (Definition 1) with vertices V and edges E.

Definition 6 *A* partial disjoint path cover *of a weighted Access Graph AG is a subgraph $C = (V' \subseteq V, E' \subseteq E)$ of AG such that $\forall v \in V', degree(v) \leq 2$ and there are no cycles in C; an* orphan vertex *is a vertex $v \in V \backslash V'$.*

[5] Aliasing may produce loops in the access path. For instance, consider the path *a-b-c-d*; if c is an alias for b, after addresses have been assigned the resulting offset sequence would be, say, 0-4-4-8. MMA instructions can only access contiguous memory locations, neither skipping nor repeating addresses, and thus this sequence of accesses is not possible.

Applying the SolveMMA algorithm to the Access Graph produces a partial covering of the graph. It is partial in that some vertices in the graph may not be covered by an access path. An orphan vertex identifies a variable that cannot be profitably accessed with an MMA instruction.

```
      // Take program $P$, construct its Access Graph $(V, E)$ return a covering
      // Access Graph $(V', E')$.
1.    procedure SolveMMA ( $P$:CFG ): AccessGraph
2.        $(V, E) \leftarrow$ ConstructAccessGraph($P$);
3.        $E_{sort} \leftarrow$ SortDescendingOrder($E$);      // Sort $E$ by weight.
4.        $V' \leftarrow V, E' \leftarrow \emptyset$;
5.        while $|E'| < |V| - 1$ and $E_{sort} \neq \emptyset$ do
6.            $e \leftarrow$ greatest edge in $E_{sort}$;
7.            $E_{sort} \leftarrow E_{sort} - \{e\}$;
8.            if $e$ does not cause any vertex in $V'$ to have degree $> 2$ and
9.              $e$ does not cause a cycle in $E'$ and
10.+            $head(e).dir \in \{UND, TAIL\}$ and
11.+            $tail(e).dir \in \{UND, HEAD\}$ then
12.               $E' \leftarrow E' + \{e\}$;
13.+              $e' \leftarrow$ reverse-edge $\in E$ of $e$;
14.+              if $e' = \emptyset$ or $weight(e') \neq weight(e)$ then
15.+                  walk from $head(e) \in V'$ marking $UND$ vertices as $HEAD$;
16.+                  walk from $tail(e) \in V'$ marking $UND$ vertices as $TAIL$;
17.+              endif
18.           else
19.               discard $e$;
20.           endif
21.       endwhile
22.       return $(V', E')$;
23.   endproc
```

Fig. 2. The SolveMMA algorithm. The steps additional to the SolveSOA algorithm (marked with a '+') are lines 10 and 11, which ensure that we never add edges that violate the direction of directed paths, and lines 13–17 which convert undirected paths into directed paths if e is an unbalanced edge (Definition 3).

The first step of the SolveMMA algorithm (Figure 2) constructs the Access Graph (V, E) from the input program P. The set of edges E is then sorted in descending order by weight, a step which simplifies the operation of the algorithm. We initialise the output Access Graph (V', E') with $V' = V$, and no edges.

The main body of the algorithm (lines 5–21) processes each edge, e, in E_{sort} in turn until either there are just enough edges in the solution to form one long path (at which point we can add no further edges to E' that would satisfy the criteria on lines 8 or 9), or there are no more edges in E_{sort}. We remove e from

E_{sort} and then decide whether it should be added to C. To do so, it must meet the following four criteria:

1. the edge must not cause a cycle in E';
2. the edge must not cause any vertex in V' to have degree > 2, *i.e.*, no edge can connect to an internal vertex within a path;
3. the head of the edge can only connect to a vertex that is either the tail of a directed path, or the end of an undirected path; and
4. the tail of the edge can only connect to a vertex that is either the head of a directed path, or the end of an undirected path.

If all four criteria are satisfied we add e to E'.

Initially, all vertices (and hence paths constructed from them) are marked *UNDecided*, reflecting no preference in the access order. However, it is very likely that in (V, E) there will be some sequences of access that are more favourable than others (*e.g.*, if there was one instance of (p, q) and two instances of (q, p)). This is reflected in a difference between the weights of the edges (p, q) and (q, p). Indeed, there may not even be a matching reverse edge (line 13).

The remaining lines, 15 and 16, handle the case of e being unbalanced, marking all the vertices from the head of e with *HEAD*, and all the vertices from the tail of e with *TAIL*. Note that this can happen at most once per path, as any subsequent addition to the path must respect its direction (lines 10 and 11). This simple heuristic works well in practice and has low runtime cost.

3.5 The Phase Order Problem

An important problem encountered by compiler designers is the *phase ordering* problem, which can be phrased as *"in which order does one schedule two (or more) phases to give the best target code?"*. Many phases are very antagonistic towards each other; two examples being code motion (which may increase register pressure) and register allocation (which places additional dependencies between instructions, artificially constraining code motion).

If SolveMMA is applied before register allocation then any subsequent spill code generated by the register allocator would not be considered by SolveMMA, and additional constraints specified by the semantics of a given target's MMA instructions would be imposed on the operation of the register allocator. If SolveMMA is applied after the instruction scheduling phase then the scheduler might construct a sequence that prevents several memory access instructions becoming a single MMA instruction.

3.6 Scheduling SolveMMA Within A Compiler

We now describe the order in which MMA optimizations are applied to a CFG in our implementation. While this order does not produce optimal code for all programs, early indications are that this order is generally good.

The first pass of *MMA Optimization* takes in a scheduled CFG, where the order of instructions (especially loads and stores) is fixed. At this stage the only

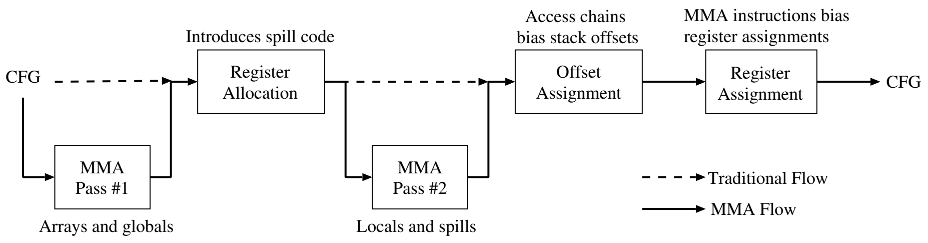


Fig. 3. Scheduling MMA optimization with other compiler phases.

memory accesses are to global variables (fixed addresses) and local arrays and structs (fixed offsets from a base address)[6]. The access graph constructed during this pass identifies *provisional* MMA instructions, albeit in a fashion which we can undo later.

The second phase is *Register Allocation.* This inserts spill code (compiler-generated loads and stores to temporaries placed on the stack) into the CFG where there are insufficient physical registers to be able to colour a register.

The third phase is another *MMA Optimization* pass, but now concerned with loads and stores introduced by register spilling. We add the spill temporaries and their access sequences to the the Access Graph of the first pass.

Phase four—*Offset Assignment*—assigns stack offsets to spilt locals and temporaries, guided by the access paths generated in the previous phase.

Finally, *Register Assignment* maps the virtual registers onto the physical registers of the target architecture. Again, we use the access paths to guide the assignment of registers to the MMA instructions, since most target MMA instructions enforce some order on the register list.

The output of this chain of phases is a CFG from which target code can be emitted. Where the register assigner has been unable to comply with the constraints of the given target's MMA instruction register ordering, we decompose a single provisional MMA instruction back into a number of smaller MMA instructions, or even single loads or stores.

3.7 Complexity of Heuristic Algorithm

SolveMMA processes each edge in $E \subseteq V \times V$, which is potentially quadratic in the number of variables in the program. Thus we require SolveMMA to be an efficient algorithm else it will be too costly for all but trivial programs. Here, we show that SolveMMA has similar complexity to SolveSOA.

Lemma 1 *The running time of* SolveMMA *is* $O(|E|\log|E| + |L|)$*, where* E *is the number of edges in the Access Graph, and* L *is the number of variable accesses in the given program.*

[6] We assume that local user variables have been mapped to virtual registers, except in the case of address-taken variables, which are turned into one-element arrays.

Proof SolveMMA is derived from SolveSOA, whose complexity has been shown to be $O(|E|\log|E|+|L|)$. Thus we only consider the additional complexity introduced in the construction of the MMA Access Graph and in lines 10–11 and 13–16 of Figure 2. For every variable access $l \in L$ there can be at most 1 adjacent variable access. Lines 10, 11 and 14 incur a constant cost per edge, as does line 13 in a well-implemented program. Lines 15–16 together walk paths to convert them into directed paths. A path can be converted from undirected to directed at most once. The total complexity is then $O(|E|\log|E|+|E|+|E|+|E|+|L|)$, *i.e.*, $O(|E|\log|E|+|L|)$. □

Using path elements [13] the test for whether an edge e forms a cycle in C is reduced to testing whether the head and tail vertices of e are common to a single path element. This test can be performed in constant time.

4 Multiple Memory Access on the VSDG

The previous section applied MMA optimization to a program in CFG form. The transformations that are possible are constrained by the precise ordering specified by the CFG. In this section we apply MMA optimization to the Value State Dependence Graph, which has fewer constraints on the ordering of instructions.

4.1 The Value State Dependence Graph

The Value State Dependence Graph (VSDG) [8] $G = (N, E_V, E_S, \ell, N_0, N_\infty)$ is a directed graph of a single procedure (as for the CFG) consisting of operation, loop (θ) and merge (γ) nodes N, special entry (N_∞) and exit (N_0) nodes, node labelling function ℓ, and value (E_V)- and state (E_S)-dependency edges.

Value dependency edges indicate the flow of values between nodes, and must be preserved during register allocation and code motion.

State dependency edges represent the *essential* sequential dependency required by the original program. These edges both fix an issue in the Value Dependence Graph (VDG) [17], and are useful for other purposes (*e.g.*, our Combined Register Allocation and Code Motion (RACM) algorithm [8]).

The VSDG inherits from the VDG the property that a program is implicitly represented in Static Single Assignment (SSA) form [5]: a given operator node, n, will have zero or more value-dependent-successors using its value.

Figure 4 shows the VSDG of the example program from Figure 1, showing memory load, memory store and add nodes, with value- and state-dependency edges, together with the resulting Access Graph and memory layout.

4.2 Modifying SolveMMA for the VSDG

The VSDG under-specifies the ordering of instructions. Thus we can reformulate the four criteria of Definition 1 to *“(1) $i \in \alpha(p)$ and $j \in \alpha(q)$ are in the same*

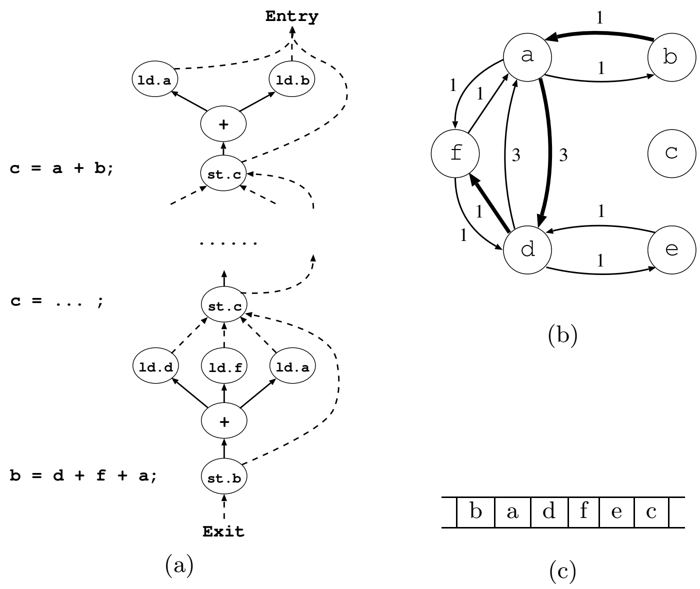


Fig. 4. VSDG of Figure 1. The solid lines in (a) are value dependencies, and the dashed lines state dependencies (the middle portion of the graph is not shown, but can be readily constructed). The labels of the memory load '`ld`' and memory store '`st`' nodes include the variable name after the period. The Access Graph of (a) is shown in (b), with the covered edges in bold, resulting in memory layout (c).

γ- and θ-dominated regions[7]*; (2) $op(i) = op(j)$; (3) for loads, j has the same state-dependent node as i; for stores, j state-depends on i; and (4) j is not value-dependent on i."*

For example, in "`x = v[a + b]`" there are three loads—one each for `a`, `b` and the indexed access into array `v[]`. But the order of the first two loads is unspecified, and so either can precede the other. This is represented in the Access Graph by two edges (a, b) and (b, a) of equal weight.

We gain two benefits from using the VSDG. The first is due to the state dependency edges defining the necessary ordering of loads and stores. For example, if a group of loads all depend on the same state, then those loads can be scheduled in any order; the VSDG *under*specifies the order of the loads, allowing MMA optimization to find an order that benefits code size.

The second benefit is due to the separation of value dependency and state dependency. Such a separation facilitates a simple method of code motion by inserting additional serializing edges into the VSDG. For example, we can hoist an expression from between two stores, allowing them to be combined into a single MMA store.

The VSDG does not change the SolveMMA algorithm itself, but we argue that it is a better data structure, and allows greater flexibility in the mixing of

[7] These are equivalent to basic blocks in the VSDG.

phases within the compiler. For example, the register allocation phase [8] can favour an instruction ordering which allows provisional MMA instructions to become actual MMA instructions.

5 Target-Specific MMA Instructions

The generic MMA instructions (§3.1) do not specify any addressing mode, since the underlying algorithm is neutral in this respect. Its purpose is to combine multiple instructions into a single instruction to reduce code size.

Tailoring the algorithm for a given architecture requires adding constraints to the code generator and register assigner. For example, the PowerPC's `lmw` and `stmw` instructions require a contiguous block of registers ending in `R31`.

The ARM Thumb `LDM` and `STM` instructions use a post-incremented register. Thus we only consider access paths where the address is increasing. `Raddr` is updated with the address of the word *after* the last word accessed by the MMA instruction. This gives another form of register re-use similar to GOA which we resolve opportunistically: after an MMA instruction the base register may be available for re-use later, saving an address computation.

6 Experimental Results

The SolveMMA algorithm has been implemented as a transformation tool within our VSDG tool chain. The tool chain consists of an ANSI C compiler (based on LCC [7]), a multi-pass optimizer, an ARM Thumb code generator, and a combined register allocator and code motion pass (an implementation of our earlier RACM algorithm [8]).

We chose the ARM Thumb because its MMA instructions (`LDMIA` and `STMIA`) are a subset of those in the full ARM instruction set, so any implementation will be applicable to both instruction sets, and because there is greater interest in producing compact code for the Thumb than for the ARM.

6.1 Motivating Example

Our first experiment is shown in Figure 5. It is a function which performs some operation on a global array, as might be found in network packet processing. SolveMMA combined all four loads into a provisional `LDM` instruction, and similarly for the four stores. However, the code generator found it cheaper (avoiding `mov` instructions) to undo the provisional `LDM` and emit four separate load instructions. Using classical optimizations not yet implemented in our experimental compiler we can remove the four instructions highlighted in Figure 5(b) by modifying the final `add` instruction.

```
void bufswap(int * array) {
  int i;
  for ( i = 0; i < 256; i += 4 ) {
    int t1, t2, t3, t4;

    t1 = array[i];
    t2 = array[i+1];
    t3 = array[i+2];
    t4 = array[i+3];

    array[i]   = t4;
    array[i+1] = t3;
    array[i+2] = t2;
    array[i+3] = t1;
  }
}
```

(a) C source

```
bufswap PROC
      push    {r4-r6,LR}
      mov     r1, #0
tn_2  cmp     r1, #255         **
      bgt     tn_1             **
      lsl     r2, r1, #2       **
      add     r6, r0, r2       **
      ldr     r5, [r6, #0]
      ldr     r4, [r6, #4]
      ldr     r3, [r6, #8]
      ldr     r2, [r6, #12]
      stmia   r6!, {r2-r5}
      add     r1, r1, #4
      b       tn_2
tn_1  pop     {r4-r6,PC}
```

(b) Thumb code (14 instructions)

Fig. 5. An illustrative example of MMA optimization of the function in (a). The output of our compiler (b) shows that we combine the four stores into a single `STMIA` instruction. This code compares favourably with that produced by the ARM commercial compiler, which required 17 instructions and one extra register; using classical optimizations our code gives 10 instructions.

6.2 Discussion of Benchmark Cases

The SolveMMA algorithm performs best on spill-code and global-variable-intensive code, where a significant proportion of the instructions can be combined into MMA instructions. Of note is that SolveMMA degrades gracefully—in the worst case a provisional MMA instruction is decomposed into a corresponding number of load or store instructions.

We applied our compiler to a number of test cases from MediaBench [11], which provide opportunities of our algorithm to reduce code size. The results are shown in Table 1 for individual functions where there was a reduction in code size. In all other cases, while SolveMMA identified many provisional MMA instructions, the limitations of the ARM Thumb addressing modes generally need to combine three instructions to save one instruction. At worst, the code generator emits a group of single loads or stores which, while not reducing code size, neither does it increase.

In the first case, `Gsm_Coder()`, the saving was through the use of a multiple push (a special form of `STM` using the stack pointer as address register with pre-decrement) to place two function arguments onto the stack prior to a function call.

The remaining cases utilise `LDM` or `STM` instructions to reduce code size. Further examination of the generated ARM code indicates places where provisional MMA instructions have been decomposed into single instructions during register allocation.

Source	Without MMA	With MMA	ARM	GCC
`code.c::Gsm_Coder()`	93	92	91	105
`decode.c::Gsm_Decoder()`	86	81	66	82
`gsm_encode.c::gsm_encode()`	561	559	430	552
`lpc.c::Reflection_coeffs()`	193	191	211	234

Table 1. Measured behaviour of MMA optimization on benchmark functions. Numbers indicate instruction counts; both ARM and GCC compilers were run with space optimization selected.

We also compared our compiler to both GCC and the ARM Ltd. commercial Thumb compiler on the same benchmarks. We achieve almost 13% *smaller* code than the GCC compiler, by spilling fewer callee-save registers and using MMA optimization. In contrast, we produce around 9% more instructions than the ARM compiler, which uses peephole optimization, a carefully designed register allocator, narrow sub-word optimizations, and other optimizations which are not the focus of this work.

7 Conclusions and Further Work

This paper introduces the SolveMMA algorithm as a tool for combining several memory access instructions into a single MMA instruction. Using a technique similar to that of Liao's SolveSOA algorithm, we both identify loads or stores that can be combined into single MMA instructions and guide stack frame layout. Implemented as a transformation within our VSDG tool chain targeting the ARM Thumb processor, we achieve up to 6% code size reduction, and never increase code size.

One question that remains unanswered is which of the currently available MMA instructions offers the best results for code compaction. In our choice of the ARM Thumb we have a single address register and a bitmap to specify the desired data registers. In contrast the MMA instructions of the PowerPC support register+displacement addressing and specify the start index of a block of contiguous registers ending in `R31`.

The current implementation of SolveMMA takes a simple approach to alias analysis, considering only loads or stores where we can directly infer their address relationship from the VSDG. More aggressive alias analysis should identify yet more opportunities for combining loads and stores into MMA instructions.

Acknowledgements

The research was supported with a grant from ARM Ltd. Thanks are due to Lee Smith for reminding us of the importance of the problem, to the Cambridge Programming Research Group for helpful discussions on the SolveMMA algorithm, to Kate Taylor for reviewing an early draft of this paper, and to the CC'04 reviewers for their helpful remarks.

References

1. *i486 Processor Programmer's Reference Manual.* Intel Corp./Osborne McGraw-Hill, San Francisco, CA, 1990.
2. Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.
3. Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.
4. Bartley, D. H. Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes. *Software—Practice and Experience 22*, 2 (February 1992), 101–110.
5. Cytron, R. K., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently Computing the Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Programming Languages and Systems 12*, 4 (October 1991), 451–490.
6. Fisher, R. J., and Dietz, H. G. Compiling For SIMD Within A Register. In *Proc. 11th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'98)* (August 1998), vol. 1656 of *LNCS (Springer-Verlag)*, pp. 290–304.
7. Fraser, C. W., and Hanson, D. R. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, 1995.
8. Johnson, N., and Mycroft, A. Combined Code Motion and Register Allocation using the Value State Dependence Graph. In *Proc. 12th International Conference on Compiler Construction (CC'03)* (April 2003), vol. 2622 of *LNCS (Springer-Verlag)*, pp. 1–16.
9. Kissell, K. MIPS16: High-density MIPS for the Embedded Market. In *Proc. Conf. Real Time Systems (RTS'97)* (1997).
10. Koseki, A., Komatsu, H., and Nakatani, T. Preference-Directed Graph Coloring. In *Proc. ACM SIGPLAN 2002 Conference on Prog. Lang. Design and Implementation* (June 2002), ACM Press, pp. 33–44.
11. Lee, C., Potkonjak, M., and Mangione-Smith, W. H. Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. 30th Intl. Symp. Microarchitectures (MICRO'30)* (December 1997), ACM/IEEE, pp. 330–335.
12. Leupers, R., and Marwedel, P. Algorithms for Address Assignment in DSP Code Generation. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design* (1996), IEEE Computer Society Press, pp. 109–112.
13. Liao, S., Devadas, S., Keutzer, K., Tjiang, S., and Wang, A. Storage Assignment to Decrease Code Size. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming Language Design and Implementation* (June 1995), ACM Press, pp. 186–195.
14. Rao, A., and Pande, S. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation* (1999), ACM Press, pp. 128–138.
15. Seal, D. *ARM Architecture Reference Manual.* Addison-Wesley, UK, 2000.
16. Udayanarayanan, S., and Chakrabarti, C. Address Code Generation for Digital Signal Processors. In *Proceedings of the 38th Conference on Design Automation* (2001), ACM Press, pp. 353–358.
17. Weise, D., Crew, R. F., Ernst, M., and Steensgaard, B. Value Dependence Graphs: Representation Without Taxation. In *ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Langs (POPL)* (January 1994), ACM.