SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (in press)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.801

SP&E

Efficient embedded code generation with multiple load/store instructions

Yunheung Paek, Minwook Ahn, Doosan Cho*,† and Taehwan Kim

School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-744, Korea

SUMMARY

In a recent study, we discovered that many single load/store operations in embedded applications can be parallelized and thus encoded simultaneously in a single-instruction multiple-data instruction, called the multiple load/store (MLS) instruction. In this work, we investigate the problem of utilizing MLS instructions to produce optimized machine code, and propose an effective approach to the problem. Specifically, we formalize the MLS problem, that is, the problem of maximizing the use of MLS instructions with an unlimited register file size. Based on this analysis, we show that we can solve the problem efficiently by translating it into a variant of the problem finding a maximum weighted path cover in a dynamic weighted graph. To handle a more realistic case of the finite size of the register file, our solution is then extended to take into account the constraints of register sequencing in MLS instructions and the limited register resource available in the target processor. We demonstrate the effectiveness of our approach experimentally by using a set of benchmark programs. In summary, our approach can reduce the number of loads/stores by 13.3% on average, compared with the code generated from existing compilers. The total code size reduction is 3.6%. This code size reduction comes at almost no cost because the overall increase in compilation time as a result of our technique remains quite minimal. Copyright © 2007 John Wiley & Sons, Ltd.

Received 21 March 2006; Revised 6 November 2006; Accepted 8 November 2006

KEY WORDS: compiler; SIMD; parallelism; code optimization; processors; embedded systems

*Correspondence to: Doosan Cho, School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-744, Korea.
†E-mail: dscho@snu.ac.kr

Contract/grant sponsor: Institute of Information Technology Assessment; contract/grant number: IITA-2005-C1090-0502-0031
Contract/grant sponsor: Korean Research Foundation (KRF); contract/grant number: D00191
Contract/grant sponsor: Ministry of Information and Communication, Korea; contract/grant number: A1100-0501-0004
Contract/grant sponsor: Korea Ministry of Science and Technology; contract/grant number: M103BY010004-05B2501-00411
Contract/grant sponsor: Nano IP/SoC Promotion Group of the Seoul R&BD program in 2006

1. INTRODUCTION

As a result of the increasing size and complexity of embedded systems, driven by the need to satisfy increasingly diverse market demands, the task of reducing code size is becoming an ever more important issue for compilers targeting embedded processors. Often the reduced code size leads to an exceptionally large positive impact on the performance of these processors because many of them are severely limited by storage constraints. To attain a desired performance goal with such limited storage, embedded processors are designed on the assumption that the software that runs on them would make heavy use of their various special hardware instructions and addressing modes [1,2].

The existing code size reduction techniques are designed to utilize the special hardware features of their target processors to satisfy a range of demands in a variety of different contexts. One notable feature of hardware instructions in modern architectures that exposes the potential for code reduction is the *multiple load/store* (MLS) instructions, which are often encountered in existing processors such as Motorola Mcore, ARM 7/9/10, Fujitsu FR30 and IBM R6000. As an example, the MLS instructions in an ARM processor [3] are of the form `ldm/stm` $\mathtt{r}_{base}, \{\mathtt{r}_1, \mathtt{r}_2, \ldots, \mathtt{r}_m\}$ where $m \leq 16$ and all the operands ($\mathtt{r}_{base}, \mathtt{r}_1, \mathtt{r}_2, \ldots, \mathtt{r}_m$) can be any of the ARM general-purpose registers `r0`, `r1`, ..., `r15`. These instructions allow, in a single operation, a large quantity of data to be transferred efficiently between any subset (or all) of the 16 registers and the memory locations starting at the address designated by the register content of r_{base}. For example, the instruction

```
ldmr1, {r3, r4, r8}
```

loads a block of three words `Mem[r1]`, `Mem[r1+4]`, and `Mem[r1+8]` into the registers in increasing order of their numbers, that is, `r3`, `r4`, and `r8`, respectively. In the instruction, the order of registers appearing inside the braces does not affect the data transfer result.

To demonstrate the benefits of utilizing MLS instructions in code generation, consider the example in Figure 1(a) which shows a fragment of C source code with six variables a, b, c, d, e, and f. Note that since a multiple load instruction allows more than one data value in *contiguous memory locations* to be transferred to registers, whereas a multiple store instruction allows data values in more than one register to be transferred to contiguous memory locations, the generation of MLS instructions is tightly related to the *memory layout* of the variables in the code. Figure 1(c) then shows the assembly of the code in Figure 1(a) translated by a commercial ARM native compiler. Figure 1(b) shows the corresponding memory layout that the compiler assigns to the variables[‡], represented in relative offsets from the stack pointers. The assembly code in Figure 1(c) has 17 load/store instructions in total, in which there is only one multiple store instruction (i.e. `stmia`), and the rest are single load and single store instructions. The `stmia` instruction in the code transfers the values of e and d that are stored in registers `r0` and `r2`, respectively, to the two (contiguous) memory locations, the lower address of which is pointed by the content of `r1`. On the other hand, Figure 1(d) shows another assembly for the source code in Figure 1(a). The corresponding memory layout is shown in Figure 1(e), which consequently enables the assembly to use only eight load/store instructions including five MLS instructions (i.e. three `ldmia` instructions and two `stmia` instructions).

[‡]The ARM compiler, as with many other conventional compilers, determine a memory layout for stack variables simply in a declaration or lexicographic order of the variables.

DOI: 10.1002/spe

```
a = a + b - c + d;
if (a > b) { qstring(a, d);
             f = f - d;
} else {  qstring(a, b);
          f = a + b;
          d = b + d;
          e = a + f;
}
f = d * a;
d = f - d;
   ...
```

(a)

```
 [sp + #0x04] : f
 [sp + #0x08] : e
 [sp + #0x0c] : d
 [sp + #0x10] : c
 [sp + #0x14] : b
 [sp + #0x18] : a
```

(b)

```
        ldr r0,[sp,#0x18]
        ldr r1,[sp,#0x14]
        ldr r2,[sp,#0x10]
        add r0,r0,r1
        sub r0,r0,r2
        ldr r2,[sp,#0xc]
        add r0,r0,r2
        str r0,[sp,#0x18]
        cmp r0,r1
        ble |l1.48|
        mov r1,r2
        bl ||qstring||
        ldr r0,[sp,#4]
        ldr r1,[sp,#0xc]
        sub r0,r0,r1
        str r0,[sp,#4]
        b |l1.88|
|l1.48| bl ||qstring||
        ldr r0,[sp,#0x18]
        ldr r2,[sp,#0x14]
        ldr r3,[sp,#0xc]
        add r1,r0,r2
        str r1,[sp,#4]
        add r0,r0,r1
        add r1,sp,#8
        add r2,r2,r3
        stmia r1,{r0,r2}
|l1.88| ldr r0,[sp,#0xc]
        ldr r2,[sp,#0x18]
        mul r1,r2,r0
        sub r2,r1,r0
        str r2,[sp,#0xc]
        str r1,[sp,#4]
```

(c)

```
        add r10,sp,#0x10
        add r9,sp,#4
        ldmia r9,{r0,r1,r2,r3}
        add r2,r2,r1
        sub r2,r2,r0
        add r2,r2,r3
        str r2,[sp,#0xc]
        cmp r2,r1
        ble |l1.48|
        mov r0,r2
        bl ||qstring||
        ldmia r10,{r0,r1}
        sub r1,r1,r0
        str r1,[sp,#0x14]
        b |l1.88|
|l1.48| bl ||qstring||
        add r9,sp,#8
        ldmia r9,{r0,r2,r3}
        add r1,r2,r0
        add r2,r2,r1
        add r0,r0,r3
        stmia r10,{r0,r1,r2}
|l1.88| add r9,sp,#0xc
        ldmia r9,{r0,r2}
        mul r1,r0,r2
        sub r0,r1,r2
        stmia r10,{r0,r1}
```

(d)

```
[sp + #0x04] : c
[sp + #0x08] : b
[sp + #0x0c] : a
[sp + #0x10] : d
[sp + #0x14] : f
[sp + #0x18] : e
```

(e)

Figure 1. Benchmark C code, its assembly with the memory layout generated by the ARM compiler with a 'O2' optimization option, and an optimized assembly resulting from an MLS-aware memory layout. (a) A fragment of C source code; (b) a memory layout of the variables in (a) determined by the ARM compiler; (c) the assembly generated by the ARM compiler for (a) resulting from the memory layout decision in (b); (d) an optimal assembly for (a) resulting from the MLS-aware memory layout in (e); (e) an MLS-aware memory layout for the variables in (a).

DOI: 10.1002/spe

Comparing the code in Figure 1(d) with that in Figure 1(c), we can see that the number of load/store instructions is reduced by 53% (from 17 to eight). The reduction of loads/stores through converting single load/store instructions to MLS instructions normally leads to a total code size reduction (in this example, by 18%) since each MLS instruction takes up only a single instruction word by encoding each register operand as a single bit. Furthermore, there will also be a certain amount of reduction in the total memory access time since the hardware can schedule memory accesses in an MLS instruction to be overlapped through *pipelining* when data are actually transferred[§].

The comparison clearly reveals that an MLS-aware compilation technique may significantly reduce the code size as well as the execution time, and is thus particularly useful in embedded system design where many applications are memory access intensive. However, the problem of utilizing MLS instructions has not been fully addressed in the literature despite the fact that finding an effective solution to the problem can be critically important to the class of embedded system designs with severely limited memory resource constraints for software storage. This is mainly the result of the potentially high degree of complexity of identifying an optimal memory layout for stack variables which is the most critical issue for achieving a maximal use of MLS instructions, as revealed in our early example.

Therefore, not surprisingly, existing compilers (mostly designed for conventional general-purpose processors) merely use MLS instructions for special occasions [4], such as exception handlers, function prologues/epilogues, and context switches, where recognizing block memory copies for MLS instructions are rather trivial. Inevitably, this implies that, to utilize MLS instructions, the users should hand-optimize their code in assembly, making programming a complex and time-consuming process.

In this paper, we present an approach to automate the process of a maximal generation of MLS instructions such that the total code size is minimized. In Section 2, we start our presentation with a formal definition of the MLS problem (MLSP), i.e. the problem of maximizing the use of MLS instructions for code size reduction with unlimited register file size. In fact, our analysis in Section 2 indicates that the MLSP with unlimited register file size, denoted as MLSP-u, is similar in several aspects to the well-known simple offset assignment (SOA) problem [1,5], the problem of assigning scalar variables to memory such that the number of explicit address arithmetic instructions is minimized by using auto-increment/decrement addressing modes. In Section 6, we will continue this analysis and argue how our MLSP is related to others mostly centering around the SOA problem by showing in more detail that these two problems are broadly similar but completely different in nature.

In Section 3, we propose a solution to a core part of the MLSP-u problem. Then in Section 4, we provide an extended solution to the general MLSP, where we allow for the constraints of register sequencing in MLS instructions and for the limited register resources available to the hardware.

We have tested our techniques experimentally by using a set of benchmark suites and in Section 5 we provide our empirical results and compare them against other compiler results.

[§]However, for some processors, such as ARM, we cannot gain a visible execution time reduction because these processors do not support pipelining for their MLS instructions. Therefore, in the ARM example of Figure 1, the optimized code would not reduce the execution time even though it achieves a 53% load/store reduction.

DOI: 10.1002/spe

2. PROBLEM FORMULATION

We first define the MLS instructions used in our technique, and list a set of constraints that should be satisfied in the process of generating MLS instructions. Then, we formally define the MLSP.

Definition 1. We define MLS instructions, following the convention of the ARM architecture, to be of the forms:

$$\{\mathtt{r}_1, \mathtt{r}_2, \ldots, \mathtt{r}_m\} = \mathtt{Mem}[\mathtt{r}_{base}]; \text{ // multiple load}$$
$$\mathtt{Mem}[\mathtt{r}_{base}] = \{\mathtt{r}_1, \mathtt{r}_2, \ldots, \mathtt{r}_m\}; \text{// multiple store}$$

where n is the number of general-purpose registers on the target architecture and $m \leq n$. The multiple load instruction transfers the data values in m memory locations,

$$\mathtt{Mem}[\mathtt{r}_{base}],\ \mathtt{Mem}[\mathtt{r}_{base}]+4,\ \ldots,\ \mathtt{Mem}[\mathtt{r}_{base}]+4m-4$$

to registers $\mathtt{r}_1,\ \mathtt{r}_2,\ \ldots,\ \mathtt{r}_m$, respectively. Conversely, the multiple store instruction transfers the data values in the m registers $\mathtt{r}_1,\ \mathtt{r}_2, \ldots,\ \mathtt{r}_m$ to the corresponding memory locations.

There are a few variants of MLS instructions that are different in meaning from those in Definition 1. Our proposed technique is designed to be flexible enough to take care of most of the variants.

Definition 2. The M-sequence constraint ensures that, in an MLS instruction, the sequence of memory locations from/to which m data values, $m \leq n$, are fetched/stored must be *contiguous*, starting from the address specified by the content of $\mathtt{r}_{base}$, i.e. $\mathtt{Mem}[\mathtt{r}_{base}], \mathtt{Mem}[\mathtt{r}_{base}+4], \ldots, \mathtt{Mem}[\mathtt{r}_{base}+4m-4]$.

Definition 3. The R-sequence constraint ensures that, in an MLS instruction, the number sequence of the m registers from/to which m data values are transferred may not be contiguous, but the sequence must be *strictly increasing*.

From the above definitions, we can see that when an MLS instruction is to be formed from loads or stores, the task of satisfying the M-sequence constraint is closely related to that of assigning variables to memory (i.e. memory layout), while the task of satisfying the R-sequence constraint is closely related to the task of register allocation.

Definition 4. The RF-size constraint ensures that the maximum number of registers to be used simultaneously is limited by the register file (RF) size constraint of the corresponding processor.

Notice in Definition 4 that when a few loads or stores are grouped to form an MLS instruction, it normally increases the life span of each value associated with them, and consequently increases the overall register pressure in the code as well. This means that, if there exists a point where the register pressure comes to be higher than the available RF size in the formation of an MLS instruction, some of those loads/stores involved in the formation need to be excluded from the instruction to reduce the resulting pressure. Otherwise, the increased pressure would very likely cause more register spills, which in turn normally offset the gains from our MLS uses in terms of code size and run time. To prevent this potential problem the RF-size constraint is enforced in our algorithm, and will be discussed in more detail in Section 4.1.

DOI: 10.1002/spe

Definition 5. Given a set S_{init} of load/store instructions in assembly code $\mathcal{C}$, the problem MLSP is to generate a set S_{MLS} of MLS instruction from S_{init} while satisfying the M-sequence, R-sequence and RF-size constraints as well as the data dependency constraint of $\mathcal{C}$ with the objective of minimizing the following factors:

(1) the total sum of the number of MLS instructions, i.e. the quantity of $|S_{\text{MLS}}|$; and

(2) the number of instructions (denoted as $|S_{\text{rest}}|$) that have not been involved in the formation of the MLS instructions in S_{init}, i.e. minimizing the quantity of

$$N_{\text{ld/st}} = |S_{\text{MLS}}| + |S_{\text{rest}}| \tag{1}$$

As stated earlier, in our approach, we first consider MLSP-u as a restricted case of MLSP in which we ignore the RF-size constraint by assuming that there is an infinite number of registers available. Then, we can easily see that the assumption of a sufficiently large number of registers makes the satisfaction of R-sequence and RF-size constraints trivial.

Definition 6. Given a set S_{init} of load/store instructions in assembly code $\mathcal{C}$, the problem MLSP-u is to generate a set S_{MLS} of MLS instructions from S_{init} while satisfying the M-sequence and dependency constraints in $\mathcal{C}$ with the objective of minimizing the quantity of Equation (1).

We describe the MLSP-u problem and our proposed solution to the problem using the example shown in Figure 2 where the original ARM code (see Figure 1) is translated into 3-address form for improved readability. Loadable regions (L-regions) and storable regions (S-regions), marked with vertical bars on the right-hand side of Figure 2, represent the maximum ranges of cycle steps in code, in which the corresponding load and store instructions can be executed without violating the data dependency specified in the code. For example, the load from the variable d in the basic block B_1 of the code has an L-region stretching from cycle 0 to cycle 6 in B_1 because there is no store before the load in B_1 and the value loaded to `r2` is first used at cycle 6. Similarly, the S-region for the store into the variable f in the block B_3 stretches from cycle 5 to cycle 9 in B_3 since the stored value is defined at cycle 4 and there is no load from f in B_3 after the store.

Then, the problem is to generate MLS instructions by grouping the load/store instructions whose L/S-regions are overlapped, so that the value in Equation (1) is as small as possible while still satisfying the M-sequence constraint. By a simple check we can see that the results of the memory layout for the variables would critically affect the ability to satisfy the M-sequence constraint for each MLS instruction, and thus affect the quality of the solution. The MLSP-u problem is obviously an NP-complete problem since a similar but even simpler problem has already been proven NP-complete by Nandivada and Palsberg [6] who use a reduction from the Hamiltonian path problem to their problem. According to their proof, even the MLSP with only double loads and stores (i.e. $m = 2$) is NP-complete. Therefore, it is clear from their proof that our general MLSP with $m \geq 2$ should be no easier than NP-complete.

3. SOLVING MLSP-u

Since MLSP-u is intractably complex as proven in [6], we attempt to circumvent the complexity using an efficient algorithm, called Solve_MLSP-u, to solve the MLSP-u problem quickly in a polynomial time bound. The Solve_MLSP-u algorithm consists of three steps.

DOI: 10.1002/spe

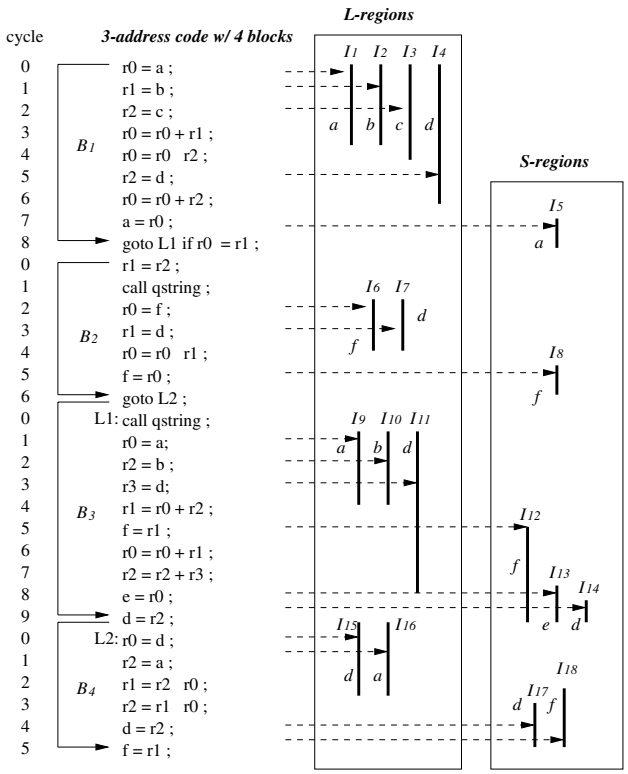


Figure 2. Four basic blocks (B_1, B_2, B_3, B_4) of the 3-address code rewritten from Figure 1(a) and the *L/S-regions* for loads/stores of each block.

- *Step 1*. From $S_{\text{init}} = \{I_1, I_2, I_3, I_4, \ldots, I_{m-1}, I_m\}$, where all instructions I_j in S_{init} are loads and stores, a group of instructions can be merged into an MLS instruction only if it satisfies the M-sequence constraint. To this end, we build a graph, called the *multiple load/store graph*, G_{MLS}, which fully describes relations between load/store instructions when forming MLS instructions.
- *Step 2*. We propose a memory layout technique and apply it to the G_{MLS} obtained in Step 1, to generate MLS instructions that minimize the value of $N_{\text{ld/st}}$ ($= |S_{\text{MLS}}| + |S_{\text{rest}}|$) in Equation (1).
- *Step 3*. Finally, registers in MLS instruction are assigned, which is a rather trivial task because it is assumed that there is a sufficiently large number of registers in MLSP-u.

3.1. Building the G_{MLS} graph

Nodes of the G_{MLS} graph represent the variables used in the load/store instructions in S_{init}. The G_{MLS} graph is a multi-graph, and there is a unique edge between two nodes n_{i_1} and n_{i_2} if there is a pair of load or store instructions I_{j_1} and I_{j_2} in S_{init} that satisfy the following two conditions.

DOI: 10.1002/spe

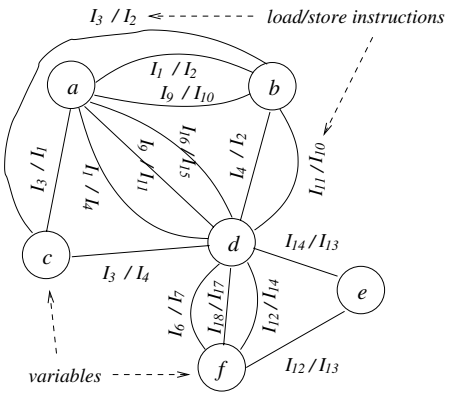


Figure 3. The G_{MLS} graph derived from the code in Figure 2.

- *Condition 1*. $\{\mathrm{var}(n_{i_1}), \mathrm{var}(n_{i_2})\} = \{\mathrm{var}(I_{j_1}), \mathrm{var}(I_{j_2})\}$, where $\mathrm{var}(x)$ indicates the variable loaded or stored by x if x is a load or store instruction, or indicates the corresponding variable if x is a node in G_{MLS}.
- *Condition 2*. The lifetimes of $\mathrm{var}(I_{j_1})$ and $\mathrm{var}(I_{j_2})$ overlap.

Figure 3 shows the G_{MLS} graph of the code in Figure 2. The graph consists of nodes, each of which represent a distinct variable used in the loads/stores. Note that each edge of the G_{MLS} graph is labeled with a pair of instructions. These instructions are a pair of loads or stores for the variables corresponding to the end nodes of the edge. For example, 'I_3/I_2' is labeled on the edge (c, b) because I_3 and I_2 respectively perform loads for c and b, satisfying Condition 1, and the lifetimes of $\mathrm{var}(I_3)$ and $\mathrm{var}(I_2)$ overlap, as shown at the L-regions of I_3 and I_2 in Figure 2, meeting Condition 2.

3.2. Generating MLS instructions

Step 2 groups load/store instructions to form MLS instructions using the G_{MLS} graph obtained in Step 1. Since the formation of MLS instructions should satisfy the M-sequence constraint, and the M-sequence constraint is closely related to the result of the memory layout of the variables, our key algorithm in this step is to find a memory layout for the variables that leads to the formation of MLS instructions with a minimum value of $N_{\mathrm{ls/st}}$ in Equation (1). Our approach in Step 2 is to formulate the problem of finding such a memory layout into a problem of finding a path cover in G_{MLS}. The rationale of this approach and the algorithm based on this approach are respectively presented as follows.

- *Rationale of our approach for Step 2*. As mentioned in Section 1, MLSP-u seems to be to some extent similar to the SOA problem in that the optimal solutions of both the problems are among all possible memory layouts of variables (i.e. all path covers in the *access graph* for the SOA problem and in the G_{MLS} graph for the MLSP) that lead to a minimum code size.

DOI: 10.1002/spe

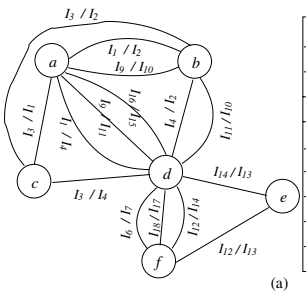
For the SOA problem, the minimum code size is achieved by maximizing the use of auto-increment/decrement mode accesses, whereas in the MLSP, the minimum code size is achieved by merging as many single loads/stores into MLS instructions as possible.
However, our MLSP is in fact more complex than the SOA problem, mainly because the SOA problem deals with an ordinary *static* weighted graph while ours must deal with a *dynamic* weighted graph where edge weights are dynamically changed as the problem is being solved. In other words, in the SOA problem, the cost (i.e. code size) of a path cover in the access graph is simply obtained by summing the weights of the edges on the path cover, but in the MLS problem the cost of a path cover in the G_{MLS} graph cannot be obtained in a straightforward manner because there might be multiple ways of merging the load/store instructions in the edges of the path cover. This means that we need a careful merging process to consider the merging conflicts among the instructions. Unlike solving the SOA problem, our version includes a new feature for dynamically updating nodes and edges in the G_{MLS} graph during the iteration process of edge selections to resolve the merging conflicts among the instructions.

- *Algorithm for finding memory layouts*. Based on the above analysis, the proposed strategy for Step 2 is an iterative one. Initially, we have an empty path P and the G_{MLS} graph. At each iteration, we select an edge in the G_{MLS} among the 'candidate' edges and expand P by including the edge to P and update the G_{MLS} graph by merging the two end nodes of the edge into a new node. More precisely, in the ith iteration of the algorithm, for every edge (v_i, v_j) in the G_{MLS} graph such that the inclusion of (v_i, v_j) to P does not cause a cycle in the initial G_{MLS} graph, the instructions corresponding to v_i and v_j are merged into an MLS instruction, and we compute the cost

$$\Delta N_{\mathrm{ls/st}} = \Delta|S_{\mathrm{MLS}}| + \Delta|S_{\mathrm{rest}}| \tag{2}$$

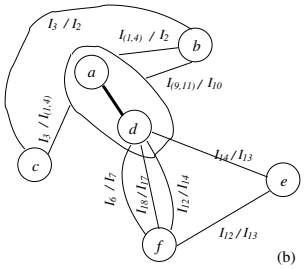
where $\Delta|S_{\mathrm{MLS}}|$ and $\Delta|S_{\mathrm{rest}}|$ are the decreased numbers of MLS instructions and non-MLS instructions for the merge, respectively. We then select, among the candidate edges, the edge (v_i, v_j) with the largest value of $\Delta N_{\mathrm{ls/st}}$, and merge the instructions of v_i and v_j into MLS instruction(s). The empty path P is then updated by including (v_i, v_j), and the G_{MLS} graph is also updated by merging the nodes v_i and v_j and updating the connected edges accordingly. The process repeats until P becomes a path that covers all the nodes in the initial G_{MLS}.
Figure 4 shows a step-by-step procedure for the generation of MLS instructions by our proposed algorithm from the G_{MLS} graph in Figure 3. The table in Figure 4(a) summarizes the values of $\Delta N_{\mathrm{ls/st}}$ of edges in the initial G_{MLS} graph. Edge (a, d) is selected because its $\Delta N_{\mathrm{ls/st}}$ is the largest. In consequence, three new MLS instructions, $I_{(1,4)}$, $I_{(9,11)}$, and $I_{(16,15)}$, which respectively come from the results of merging instructions I_1 and I_4, I_9 and I_{11}, and I_{16} and I_{15}, are produced. The left-hand side of Figure 4(b) shows the updated G_{MLS} graph where the thick edge indicates a (partial) path cover (i.e. P). The table in Figure 4(b) shows the $\Delta N_{\mathrm{ls/st}}$ values of edges. Edge (d, f) is then selected, and thus three new MLS instructions are generated accordingly. Figures 4(c)–(e) show the updated G_{MLS} graph and computations of $\Delta N_{\mathrm{ls/st}}$ values in the third, fourth, and fifth iterations in Step 2 of Solve_MLSP-u, respectively. Finally, Figure 4(f) shows the final path cover $c - b - a - d - f - e$, which becomes exactly the memory layout of variables. From the results, we can see that the total reduction of instructions is 10, which is from 3, 3, 2, 1, and 1 reductions in the first, second, third, fourth, and fifth iterations. We summarize our algorithm for finding memory layout in Figure 5.

DOI: 10.1002/spe



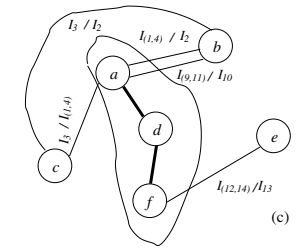
paths	new merged MLS	$\triangle N_{l/s}$
a b	$I_{(1,2)}$ $I_{(9,10)}$	2
a c	$I_{(1,3)}$	1
a–d	$I_{(1,4)}$ $I_{(9,11)}$ $I_{(16,15)}$	**3**
b c	$I_{(2,3)}$	1
b d	$I_{(2,4)}$ $I_{(10,11)}$	2
c d	$I_{(3,4)}$	1
d e	$I_{(13,14)}$	1
d f	$I_{(6,7)}$ $I_{(12,14)}$ $I_{(17,18)}$	**3**
e f	$I_{(12,13)}$	1

(a)



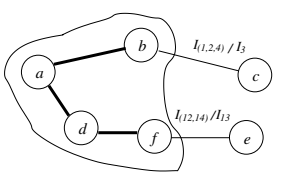
paths	new merged MLS	$\triangle N_{l/s}$
a d b	$I_{(1,2,4)}$ $I_{(9,10,11)}$	2
a d c	$I_{(1,3,4)}$	1
a d e	$I_{(13,14)}$	1
a–d–f	$I_{(6,7)}$ $I_{(12,14)}$ $I_{(17,18)}$	**3**
a d, b c	$I_{(2,3)}$	1
a d, e f	$I_{(12,13)}$	1

(b)

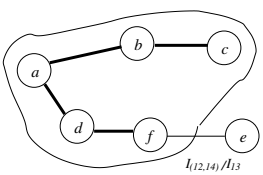


paths	new merged MLS	$\triangle N_{l/s}$
c a d f	$I_{(1,3,4)}$	1
b–a–d–f	$I_{(1,2,4)}$ $I_{(9,10,11)}$	**2**
a d f e	$I_{(12,13,14)}$	1
a d f, b c	$I_{(2,3)}$	1

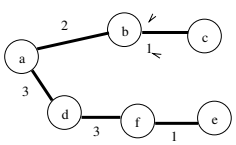
(c)



(d)



(e)



Final path cover: c –– b –– a –– d –– f –– e
#instruction_reduction at each iteration
a –– d : 3
(a––d) –– f : 3
b –– ((a––d)––f) : 2
c– (b––(a–d)––f): 1
(c–(b–((a–d)––f))–e: 1
Total reduction of instructions: 3+3+2+1+1 = 10

(f)

Figure 4. Example to illustrate the detailed procedure of Step 2, generation of MLS instructions from G_{MLS}. (a) Initial G_{MLS} and the cost computation for each pair of nodes in the first iteration; (b) the updated G_{MLS} and the cost computation for each pair of nodes in the second iteration; (c) the updated G_{MLS} and the cost computation for each pair of nodes in the third iteration; (d) the updated G_{MLS} in the fourth iteration (node c is selected and merged); (e) the updated G_{MLS} in the fifth iteration (node e is selected and merged); (f) final results of MLS instructions and memory layout (path cover).

DOI: 10.1002/spe

```
MLSGen(Program P)
  G_MLS ← Construct_MLS_Graph(P);
  for each edge e in G_MLS
    e.gain = ComputeDelta(e);
  Q ← priority queue of edges using gain as key;
  Path ← ∅;
  while Q is not empty do
    e ← Pop(Q);
    if (Path ∪ {e}) results in a cycle then continue;
    else add e to Path;
  od
  Assign memory offsets to memory variables
    according to their position in the Path;
end
```

Figure 5. Greedy algorithm designed to satisfy the M-sequence constraint.

3.3. Assigning registers

In Step 2, we satisfied the memory sequence constraint of the MLS instruction. To complete the work, register reassignment is necessary to satisfy the R-sequence constraint. Although we have made the assumption of infinite registers for MLSP-u, register assignment is not trivial since there can be conflicts between R-sequences imposed by MLS instructions from Step 2. For example, in the code

$$\{sr_1,\ sr_2,\ sr_3\} = \texttt{Mem}[sr_{addr}] \text{ // multiple load}$$
$$\ldots$$
$$\texttt{Mem}[sr_{addr}] = \{sr_2,\ sr_1,\ sr_3\} \text{ // multiple store}$$

the multiple load instruction requires that sr_1 should be assigned a smaller physical register number than sr_2, while the multiple store instruction requires the opposite. To solve this problem, we check conflicts between all MLS candidates before register reassignment. If any conflict exists, the conflicting MLS instructions are split one by one until all the conflicts are resolved. This procedure is described in greater detail later in Section 5.2.

The time complexity of the Solve_MLSP-u is dominated by that of Step 2, which is bounded by $O(N^3)$ where N is the number of variables, because the number of iterations in Step 2 is exactly N and each iteration considers at most N^2 pairs of nodes in graph G_{MLS}.

4. SOLVING MLSP

In this section, we describe our algorithm, called Solve_MLSP, which is an extension of Solve_MLSP-u, to the general MLSP. Our approach to solving MLSP extends Solve_MLSP-u by satisfying two additional constraints: R-sequence and RF-size. Figure 6 shows the flow of our integrated solution to MLSP where Step 1 of Solve_MLSP-u is extended to support the RF-size constraint while Step 3 is carefully designed to support the R-size constraint.

DOI: 10.1002/spe

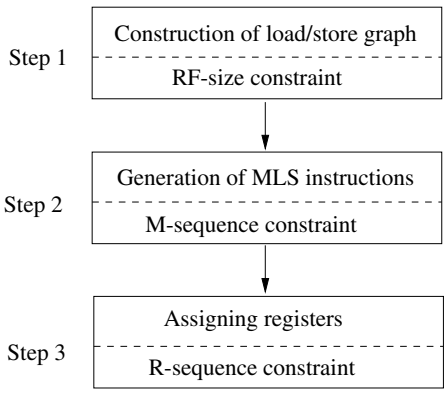


Figure 6. The flow of Solve_MLSP.

4.1. Supporting the RF-size constraint

One central task of Step 1 in Solve_MLSP is to use the L/S-regions to identify parallel loads/stores. For this, it first divides the input procedure into basic blocks, and, for each block, computes the L/S-regions as shown in Figure 2. We define the L-Region and S-Region as follows.

Definition 7. (Loadable region) Suppose a basic block B contains a load $r = v$ at the cycle t that loads the value into the register r from the memory location denoted by the variable v. Then, the Loadable region (L-region) of the load is the time interval $int_v = [lb,ub]$ where its lower and upper bounds lb and ub are respectively defined as follows.

- If there occurs the last store into v or use of v at some cycle t' in B before the load, then $int_v.lb = t' + 1$. Otherwise, $int_v.lb = 0$.
- If the value loaded at t is first used at some cycle t'' in B, then $int_v.ub = t''$. Otherwise, $int_v.ub = |B| - 1$.

Definition 8. (Storable region) Suppose a basic block B contains a store $v = r$ at the cycle t that stores the value from the register r into the memory location denoted by the variable v. Then, the Storable region (S-region) of the store is the time interval $int_v = [lb, ub]$ where its lower and upper bounds lb and ub are respectively defined as follows.

- If the register value stored at t was last defined at the cycle t' in B, then $int_v.lb = t' + 1$. Otherwise, $int_v.lb = 0$.
- If there is the first load from v or defined v at the cycle t'' in B after the store, then $int_v.ub = t''$. Otherwise, $int_v.ub = |B| - 1$.

DOI: 10.1002/spe

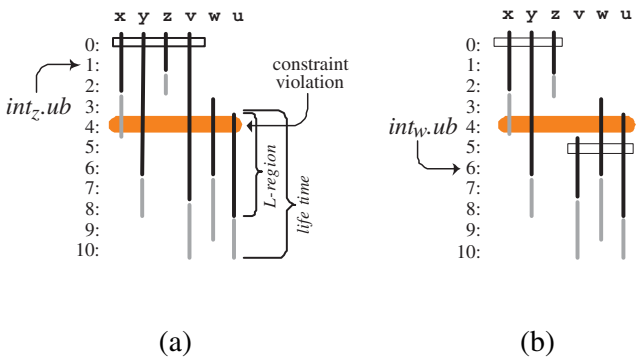


Figure 7. RF-size constraint test for four available registers: (a) int_z as the seed interval; (b) int_w as the seed interval.

In principle, any loads/stores are parallel as long as their L/S-regions are overlapped. So, in forming MLS instructions, these parallel loads and stores may initially all be gathered to form an MLS instruction. However, this naive gathering may cause many new register spills in the final code. To explain this with an example, suppose in Figure 2 that we combine a load for d in block B_1 with those for a, b, and c to form an MLS instruction. The generation of an MLS instruction requires that the load for d should move up from cycle 5 to 2 or even earlier. This movement would prolong the lifetime of the value in the register `r2`, possibly also increasing the register pressure. As discussed in Section 2, to prevent extra spills resulting from the increased register pressure, the RF-size constraint is enforced during Step 1 when parallel loads/stores are collected from the top of the code.

If the current configuration of collection of loads/stores violates the RF-size constraint, some L/S-regions (i.e. *intervals*) are removed from the configuration until the constraint is satisfied. To explain this more precisely, consider Figure 7 where each L-region is extended with a gray line to represent the whole life span of the value loaded from a memory location. Assume that the target machine currently has only four registers available for loads/stores in this part of the code. In Figure 7(a), a collection of L-regions is first formed with four intervals (int_x, int_y, int_z, int_v), beginning with int_z as the *seed interval*, the interval whose upper bound is the lowest in the block. However, under the register file size limit ($=4$), moving up the two loads for v and y to join I before cycle 2 would cause the resulting pressure to violate the RF-size constraint by exceeding the limit at cycle 4. When this violation is reported, the load for v to move up before cycle 5 is not considered by eliminating it from the collection of L-regions and adjusting its lower bound $int_v.lb$ to 5, as shown in Figure 7(b), where we can now see that the constraint is no longer violated. Although we could also prevent the violation by choosing int_y instead of int_v, we choose the interval with the longest tail since its life span stretches longest having more chance of overlapping with other intervals. After int_v is removed, the three intervals remaining in I will form a block of parallel loads, as shown in Figure 7(b). Then, by the definition of a seed interval, the interval int_w will be selected as the next seed, and clustered with the other two intervals, int_v and int_u.

DOI: 10.1002/spe

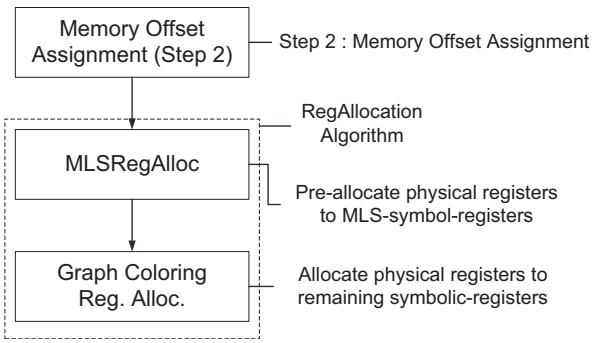


Figure 8. MLS register allocation flow.

4.2. Supporting the R-sequence constraint

Each of the MLS instructions produced in Step 2 should be a block of parallel loads/stores accessing contiguous memory locations starting at their base offset m_{base} from the stack pointer. Since the M-sequence constraint was satisfied during the process of Step 2, each MLS with k loads or stores can, for a multiple load, be of the form

$$r_{base} = \texttt{sp+\#}m_{base}$$
$$\{r_1, r_2, \ldots, r_k\} = \texttt{Mem[}r_{base}\texttt{]}$$

or for a multiple store

$$r_{base} = \texttt{sp+\#}m_{base}$$
$$\texttt{Mem[}r_{base}\texttt{]} = \{r_1, r_2, \ldots, r_k\}$$

The definition of the R-sequence constraint can be divided in two parts:

(1) all register operands in an MLS instruction must be distinct;
(2) the memory words are transferred from/to the registers in increasing order of the register numbers.

The first part of the R-sequence constraint is trivially met in Step 3 since the RF-size constraint, enforced in Step 1, ensures that the register pressure always stays within the register file size. However, the second part is not easy to satisfy since it requires register reallocation.

Figure 8 shows the flow of the register allocation algorithm. The work is done in two steps. In the *MLSRegAlloc* step, we pre-allocate physical registers to the operands of MLS instructions such that they satisfy the R-sequence constraint. Then, a general graph coloring register allocation algorithm is invoked to assign registers to the remaining symbolic registers. Since the second step is the same as a traditional graph coloring register allocation, we concentrate on the *MLSRegAlloc* step in this section.

In *MLSRegAlloc*, we first extract the R-sequence constraint of each MLS instruction and resolve conflicts between them. Then, all the R-sequence constraints are summarized in graph form, which, in turn, is used to guide the register allocation to satisfy the R-sequence constraint.

DOI: 10.1002/spe

```
        ...
add     sr15,SP,#0x10
ldmia   sr15,{sr0,sr1,sr2,sr3}
add     sr11,sr1,sr2
add     sr4,sr11,sr3
sub     sr5,sr3,sr4
stmia   sr15,{sr0,sr4,sr5,sr3}
add     sr12,sr3,#0x10
mult    sr6,sr7,sr0,sr12
add     sr16,SP,#0x20
stima   sr16,{sr6,sr7,sr0}
add     sr17,SP,#0x10
ldmia   sr17,{sr8,sr9,sr10}
stmia   sr17,{sr9,sr8,sr10}
        ...
```

(a)

```
order1: sr0<sr1<sr2<sr3
order2: sr0<sr4<sr5<sr3
order3: sr6<sr7<sr0
order4: sr8<sr9<sr10
order5: sr9<sr8<sr10
```

(b)

```
order1: sr0<sr1<sr2<sr3
order2: sr0<sr4<sr5<sr3
order3: sr6<sr7<sr0
order4: sr8<sr9<sr10
order5: sr9
order6: sr8<sr10
```

(c)

Figure 9. An example assembly code generated after satisfying M-sequence constraint. (a) Example assembly code generated from Step 2 (MLS Generation); (b) R-sequence constraints of code (a); (c) R-sequence constraints of code (b) after resolving conflicts.

4.2.1. Conflict resolution

The R-sequence constraint of each MLS instruction can be represented as an *ordered set O* which is defined as $(X, <)$, where X is the set of symbolic registers sr_i and $<$ is the total order between two symbolic registers sr_i, sr_j, such that sr_i should be assigned a smaller physical register number than sr_j. In the example in Figure 9, we can observe three ordered sets `order4`, `order5`, and `order6` in Figure 9(c) instead of two ordered set `order4` and `order5` in Figure 9(b), because `order4` requires $sr8 < sr9$ while `order5` requires $sr9 < sr8$ after register allocation and then we have to split `order5` into $sr9$ and $sr8 < sr10$ resulting in two memory access operations to resolve this conflict.

To resolve the conflicts, we first check conflicts between all pairs of MLS instructions and summarize the result in a graph called a *conflict graph*. In the *conflict graph*, each node denotes an MLS instruction and each edge denotes a conflict between two corresponding MLS instructions. The conflicts are resolved by splitting the MLS instruction corresponding to the maximum-degree node one by one until all the conflicts are eliminated. In our example code, `order5` is split into $sr9$ and $sr8 < sr10$ resulting in two memory access operations.

4.2.2. ROG/MIG construction

To satisfy the R-sequence constraint, we need to express both the R-sequence constraint and any interference between symbolic registers in a single data structure. For this, we first combine all the R-sequence constraints and represent it as a directed acyclic graph (DAG) called register order graph (ROG). The ROG summarizes the whole order of symbolic registers related to MLS instructions.

DOI: 10.1002/spe

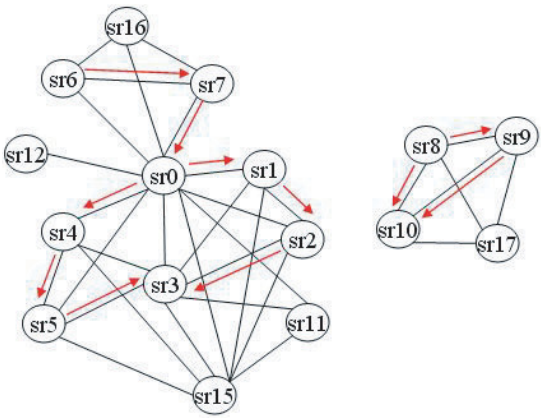


Figure 10. Modified interference graph (MIG).

Then we merge the ROG and the conventional interference graph (IG) to generate a multi-graph called a modified interference graph (MIG). The MIG contains both the R-sequence and interference information, and is thus used to guide register reassignment.

Definition 9. (ROG) Let V be the set of all symbolic registers related to MLS instructions. Then a ROG $= (N, E)$ is defined as follows:

(1) $N = V$;
(2) for any edge $u \rightarrow v$ in E, there exists an order set O such that u and v are adjacent in O and $u < v$, where the adjacency in the ordered set O means that there is no elements between nodes u and v in sorted manner.

Definition 10. (MIG) For a ROG $R = (N_R, E_R)$ and IG $I = (N_I, E_I)$, let MIG $M = (N_M, E_M)$. Then M is a multi-graph and is defined as follows:

(1) $N_M = N_I$;
(2) $E_M = E_R U E_I$.

Figure 10 shows the resulting MIG for the code in Figure 9. The directed edges denote the edges of ROG and the undirected edges represent those of the IG.

After constructing ROG, we can see that the *height*[¶] of the ROG indicates the minimum number of physical registers required to satisfy the R-sequence constraint. Therefore, when the height of the ROG is greater than the number of physical registers ($= N$), the condition should be relaxed by splitting one or more MLS instructions. The relaxation is accomplished by a MinCostCut routine in

[¶]Defined as the longest path from the top node to the leaf node.

DOI: 10.1002/spe

```
// OrderSet : set of R-sequence constraints.
// G_IG : original interference graph.
// N : number of physical registers.
build_MIG(OrderSet, G_IG, N)
   G_ROG ← ConstructROG(OrderSet);
   while the height of G_ROG > N do   // Relax height
      E_split ← MinCostCut(G_ROG);   // edges to cut
      G_ROG ← G_ROG − E_split;   // Update G_ROG
      od
   G_MIG ← G_ROG ∪ G_IG;   // Construct G_MIG
   return G_MIG;

// G_ROG : Register Order Graph.
// G_MIG : Modified Interference Graph.
MLS_RegAlloc(G_MIG, G_ROG, N)
   remove non-MLS nodes from G_MIG;
   for each connected component C of G_ROG do
      while DepthFirstAlloc(G_ROG, N) is failed do
         E_split ← MinCostCut(C);
         C ← C − E_split;
      od
   od

DepthFirstAlloc(G_ROG, N)
   DFO ← GetDepthFirstOrder(G_ROG);
   for each node n in DFO in sequential order do
      flag ← AssignReg(G_ROG, N, n);
      if flag == failure then return failure;
   od

// n : Target node for register allocation
AssignReg(G_ROG, N, n)
   for(r = N − 1; r ≥ 0; r − −) do
      if r violates R-sequence constraint or r is already
      used by one of its neighbors then
         continue;   // try next register
      else do
         assign r to node n;
         return success;
   od
   return failure; end
```

Figure 11. Algorithm to assign physical registers to MLS register operands.

DOI: 10.1002/spe

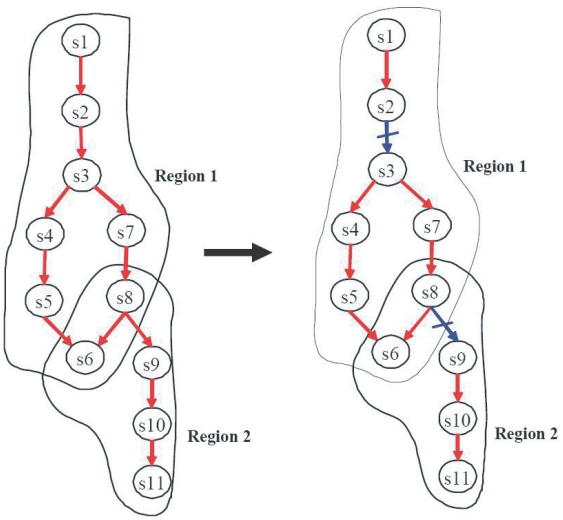


Figure 12. A MIG example where register allocation is still impossible even after adjusting the height of ROG: (a) before the R-sequence constraint is relaxed; (b) after the R-sequence constraint is relaxed.

the following manner. For every edge in the ROG, it computes the expected gain using

$$Gain = (\Delta\ Height\ of\ ROG) / (\Delta\ CodeSize) \tag{3}$$

and cuts the edge with maximum gain. The corresponding MLS instructions are split accordingly. The MinCostCut routine is invoked repeatedly until the height becomes less than or equal to N.

4.2.3. Register allocation

At the final step, physical registers are allocated to register operands of MLS instructions guided by the ROG and MIG. Since we consider only MLS operands, non-MLS nodes are removed from the MIG and physical registers are sequentially assigned to MLS nodes in depth first traversal order, starting from the largest register number available. Figure 11 lists the MLSRegAlloc algorithm.

Although the height of the ROG is adjusted in the previous step, there can still be cases where register allocation is impossible as a result of the R-sequence constraint. In the case shown in Figure 12, even with height 8, more than eight registers are required to meet the R-sequence constraint. The smallest register number that can be assigned to symbolic register $s8$ is $r3$ since four symbolic registers $s6$, $s9$, $s10$, $s11$ should be assigned unique register numbers satisfying the R-sequence constraint at the same time. However, the remaining three registers ($r0$, $r1$, $r2$) are insufficient to satisfy the R-sequence constraint of $s1$, $s2$, $s3$, $s7$. We solve this problem by using the same MinCostCut routine as before. Whenever such cases are met, the MinCostCut is invoked to relax the R-sequence constraint. Figure 12(b) shows the relaxed version of the MIG in Figure 12(a).

DOI: 10.1002/spe

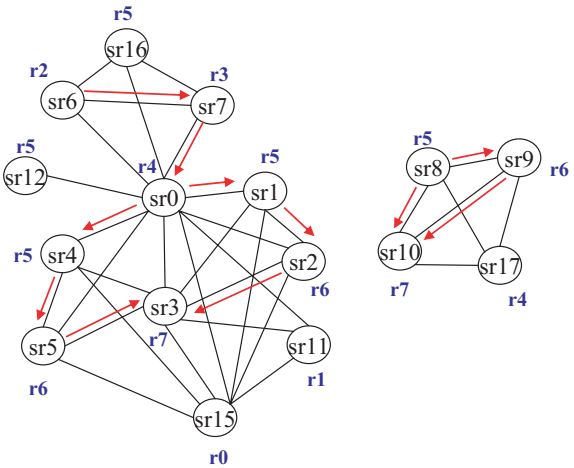


Figure 13. The final result of register allocation on the MIG of Figure 10.

After assigning registers to MLS operands, we assign registers to the remaining non-MLS nodes using a traditional graph coloring register allocation algorithm. The MLS nodes are regarded as pre-assigned nodes. Figure 13 shows the final result on the MIG of Figure 10 assuming eight physical registers.

5. EXPERIMENTAL RESULTS

We implemented the proposed algorithm Solve_MLSP in C++ and evaluated it on a set of two embedded benchmark suites in *DSPStone* [7] and *MediaBench* [8]. *DSPStone* consists of well-known digital signal processing (DSP) benchmark kernels. Unlike *DSPStone*, *MediaBench* consists of full codes for complete media applications. Solve_MLSP is attached to the GCC-ARM compiler as a post-optimization phase and optimizes only local stack variables; arrays or global variables are not touched. The evaluation was conducted on an ARM 7 processor. For each benchmark, the GCC-ARM compiler is used to generate the initial assembly code and then our technique is applied to obtain an MLS-optimized version of the code. We measured the compile time cost on an Intel machine with one 3.4 GHz Pentium 4 processor and 1 GB RAM and report the result for *MediaBench*.

There are several issues to address before going into the experimental results.

(1) The Solve_MLSP algorithm is designed to be applied to the assembly code, so before applying Solve_MLSP we computed a web of physical registers and converted them into symbolic registers except for special purpose registers.
(2) When computing L-regions and S-regions, the results depend on the computation order of candidate load/store instructions. This is because the extended lifetime of precomputed L-regions (or S-regions) affects the RF-size constraint of the remaining (unprocessed) load/store operations. In our implementation, we first computed L-regions for all load operations in order, and then computed S-regions for all store operations in order.

DOI: 10.1002/spe

Table I. Comparisons of code size before and after the application of Solve_MLSP to the unoptimized codes by the GCC compiler for *DSPStone* benchmarks.

	Original		MLS_Opt		Difference (%)
Benchmark code	*Total size*	*Mem size*	*Total size*	*Mem size*	*Total size/mem size*
DOT_PRODUCT	37	16	37	16	0.0/0.0
CONVOLUTION	50	18	50	17	0.0/5.6
N_REAL_UPDATES	76	36	74	33	2.6/8.3
FFT_BIT_REDUCT	112	38	112	38	0.0/0.0
LMS	115	51	110	43	4.3/15.7
MATRIX1	117	44	115	39	1.7/11.4
BIQUAD_N_SECTIONS	124	58	122	53	1.6/8.6
N_COMPLEX_UPDATES	128	60	126	57	1.6/5.0
FIR2DIM	229	85	227	82	0.8/3.5
Average	—	—	—	—	1.4/6.5

5.1. Results on two embedded benchmarks

5.1.1. DSPStone benchmark

In this experiment, two versions of the codes are generated and compared. The first version is an unoptimized ARM assembly code generated by GCC-ARM Compiler 3.3. The second version is generated after applying Solve_MLSP to the first version. Table I summarizes and compares the two code versions. Original and MLS_Opt indicate the first and the second version respectively, and *total size* and *mem size* indicate the total code size and the number of memory access instructions, respectively. *Mem size* and *total size* may be expressed as

$$mem\ size = loads/stores + MLSs$$

$$total\ size = mem\ size + BRSs + other\ instructions$$

where BRSs represents the number of base register setting instructions which are additional address computation instructions for initializing the base register for each MLS.

The table shows that *Solve_MLSP* is able to reduce the memory access code size by 6.5% on average and the total code size by 1.4% when a normal ARM 7 processor is used. For the *DSPStone* benchmark suite, *Solve_MLSP* is not very effective as a result of the small code size of *DSPStone* code. Since *DSPStone* benchmark is a set of synthesized programs of small kernel routines, there is little opportunity for MLS optimization.

5.1.2. MediaBench benchmark

To evaluate the effect of Solve_MLSP on practical cases, we performed a second experiment with larger codes than those from *DSPStone*. For this, we chose benchmark programs from *MediaBench*,

DOI: 10.1002/spe

Table II. Comparisons of code size before and after the application of Solve_MLSP to each function body optimized by the GCC 3.3 compiler for *MediaBench* benchmarks.

	Original		MLS_Opt		Difference (%)
Benchmark code	*Total size*	*Mem size*	*Total size*	*Mem size*	*Total size/mem size*
COMPUTE_COLOR	119	66	113	47	5.0/29.0
QUANTIZE_FS_DITHER	137	78	129	58	5.8/25.6
DPFIELD_ESTIMATE	147	73	136	54	7.5/26.0
FULL_SEARCH	193	69	189	56	2.1/18.8
FRAME_ESTIMATE	200	142	195	130	2.5/8.5
PASS2_FS_DITHER	202	116	182	79	9.9/31.9
FIELD_ESTIMATE	317	198	317	190	0.0/4.0
DPFRAME_ESTIMATE	369	161	343	115	7.0/28.6
FIELD_ME	1008	458	972	380	3.6/17.0
FRAME_ME	1039	528	1008	446	3.0/15.5
Average	—	—	—	—	4.6/20.5

which have already been optimized with a 'O2' option, and, as in the first experiment, we generate two versions of the machine codes from them.

The ARM native compiler we tested in this experiment utilizes MLSs instructions that are highly efficiently for function calling conventions where various parameters and pointers are saved and loaded during function calls. Its function call optimizations use MLSs to reduce memory instructions needed to implement calling conventions. However, the compiler does not utilize MLSs instructions for ordinary loads/stores within a function body. To isolate these *interprocedural* optimization effects of the ARM compiler from its *intraprocedural* effects and compare them with our algorithm, we conducted two different levels of experiments.

- *Intraprocedural level*. We exclude the interprocedural effect by comparing the output results only for each individual function body without function prologues/epilogues for calling convention.
- *Interprocedural level*. We include the effects of functional call optimizations by comparing the output results for the full codes of the benchmarks containing multiple functions. It also shows the overall effectiveness of our algorithm.

Tables II and III respectively show the intraprocedural and interprocedural experimental results. As expected above, owing to the limited intraprocedural utilization of MLSs instructions of the ARM compiler, Solve_MLSP outperforms the compiler in Table II more than in Table III.

In an experiment using full codes, Solve_MLSP achieved a 1.8 to 23.6% reduction of memory code size, and up to a 5.6% reduction to the total code size. On average, the total code size is decreased by 3.6%, and memory code size is decreased by 13.3%. For *MediaBench*, Solve_MLSP achieves much better improvement than it does for the *DSPStone* benchmark.

The main reason for this difference is the fact that *MediaBench* has a larger code size and accesses memory variables more frequently. In fact, observations on the *MediaBench* suite reveals that memory

DOI: 10.1002/spe

Table III. Comparisons of code size before and after the application of Solve_MLSP to each full code optimized by the GCC 3.3 compiler for *MediaBench* benchmarks.

	Original		MLS_Opt		Difference (%)
Benchmark code	*Total size*	*Mem size*	*Total size*	*Mem size*	*Total size/mem size*
G721	748	279	736	248	1.6/11.1
GSM	3097	1231	2952	1060	4.7/13.9
RASTA	4487	2159	4447	2119	0.9/1.8
MPEG2_DECODER	4372	2015	4160	1813	4.8/10.0
EPIC	5066	1974	4871	1587	3.8/19.6
MPEG2_ENCODER	9204	4268	8684	3259	5.6/23.6
Average	—	—	—	—	3.6/13.3

access instructions (such as load/store) occupy a significant portion of the program. For instance, 46.4% of MPEG2_ENCODER, 38.9% of EPIC, and 39.7% GSM are memory access instructions. A second reason is that the GCC-ARM compiler lacks the optimization techniques to exploit MLS instructions. By examining the assembly code generated by existing compilers, we found that, even at the full optimization level, the GCC-ARM compiler as well as the ARM native compiler [9] generate MLS instructions in a very limited way such as register–save–restore instructions at function prologue and epilogue.

Note that, as we would expect, the exploitation of MLS instructions by Solve_MLSP does not lead to a dramatic decrease in the total code size. However, considering that we can successfully further reduce the code size even after every effort has been made by both compilers for code optimizations, we believe these results are significant. Besides, although it is not shown explicitly in the experimental results, we believe that the percentage reduction in loads and stores (about 10–20%) would bring about a tangible reduction in execution time and energy consumption[‖], which are also equally important performance metrics in embedded processors.

5.1.3. Compile time cost

To see the compilation overhead resulting from the use of our technique, we measured the time taken to perform the Solve_MLSP algorithm as illustrated in Figure 14. The figure shows that the compile time is given as a ratio, in percent, of *MLSGen* and *MLSRegAlloc* with respect to the total compile time; *MLSGen* denotes the time taken at Step 1 (G_{MLS}) + Step 2 (*MLSGen*), and *MLSRegAlloc* denotes the time taken for allocating registers to MLS operands. As you can see in the figure, *MLSRegAlloc* is the most dominant factor since it involves many complex graph data structures, such as ROG, IG, and MIG. Figure 15 shows the relationship between the input code size and the compile time cost.

[‖] Many studies report that in most embedded system applications a large portion of energy consumption and execution time is due to memory access operations [10,11].

DOI: 10.1002/spe

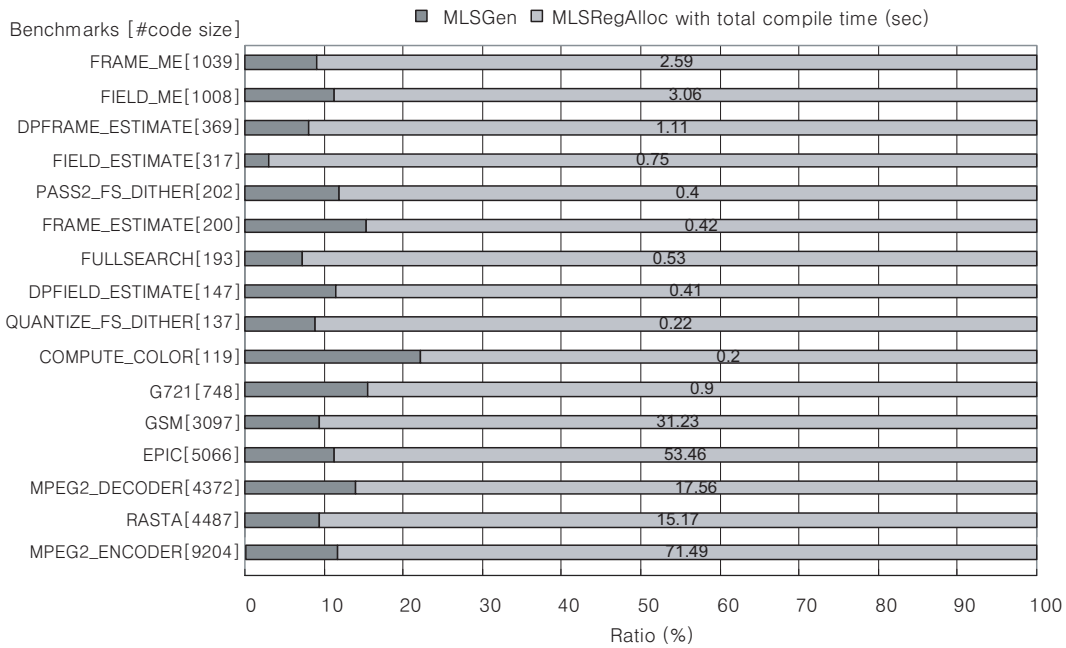


Figure 14. Compile time cost taken at each step of the Solve_MLSP algorithm.

At the end of Section 3, we stated that the time would increase as $O(N^3)$ for N variables in the input code. Figure 15 roughly confirms our time analysis. This empirical result ensures that the Solve_MLSP does not add a heavy burden to existing compilers.

5.2. Statistics of generated MLS instructions

To analyze the effect of Solve_MLSP, we classified the resulting MLS instructions with respect to the number of associated memory access operations. Figure 16 shows the statistics for the *MediaBench* suite. As you can see in the figure, more than 90% of MLS instructions are composed of less than six memory access operations. Specifically, 93% of MLS instructions consists of two to five loads (or stores) and MLS instructions with more than 12 loads (or stores) are not generated at all. This can be explained by the following reasons.

(1) Most of the basic blocks contain small number of instructions; around 10 instruction on average. This limits the size of parallel loads/stores, which in turn leads to dominance of small-size MLS instructions.
(2) The RF-size constraint limits the size of possible parallel loads/stores.
(3) Even if the code contains large-size parallel loads/stores, most of them are likely to split in the course of applying the M-sequence and R-sequence constraints.

DOI: 10.1002/spe

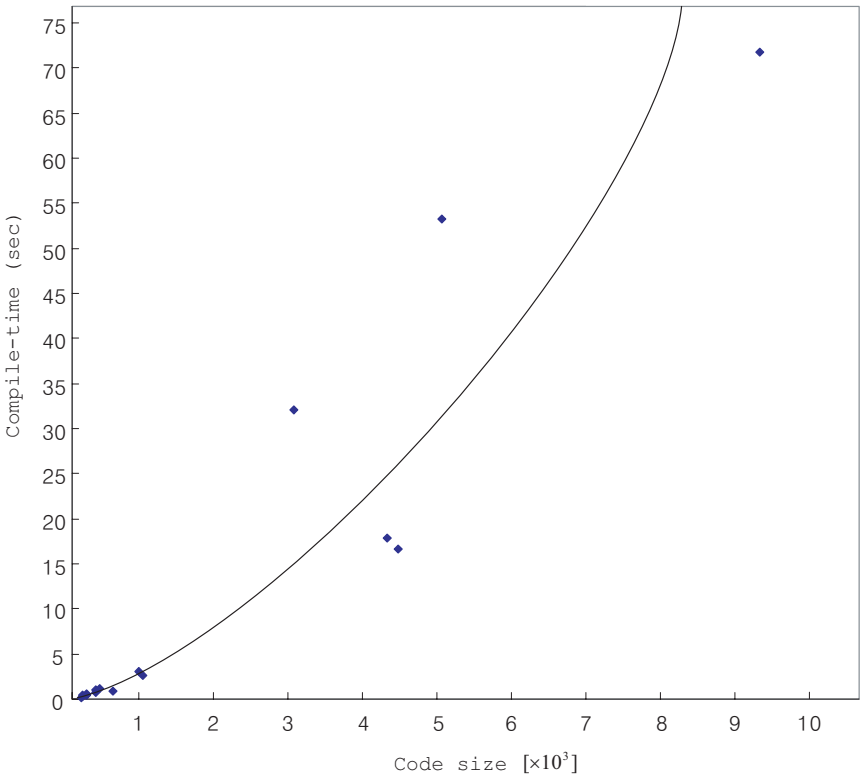


Figure 15. The relationship between compile time cost and input code size.

We can also observe that *double load/store* (MLS instructions with two loads/stores) occupies approximately 50% of the resulting MLS instructions. Note that double load/store does not affect the code size since almost all MLS instructions require one additional arithmetic operation for address computation. Thus, from this observation, we have found that only half of the total MLS instructions found by our technique contribute to the reduction of our code size.

One potential concern for an MLS instruction is that, in most architectures, interrupts are disabled while all data values in the instruction are completely transferred. For some hard real-time applications, this may lead to undesirable system behavior because the handling of interrupts may become too delayed for an MLS instruction with a long list of data values. This is, in fact, one reason why the ARM native compiler limitedly uses MLS instructions. Nonetheless, Figure 16 reveals that the adverse effect of the Solve_MLSP algorithm on the interrupt response time may be safely ignored or minimized in practice owing to the dominance of small-size MLS instructions.

DOI: 10.1002/spe

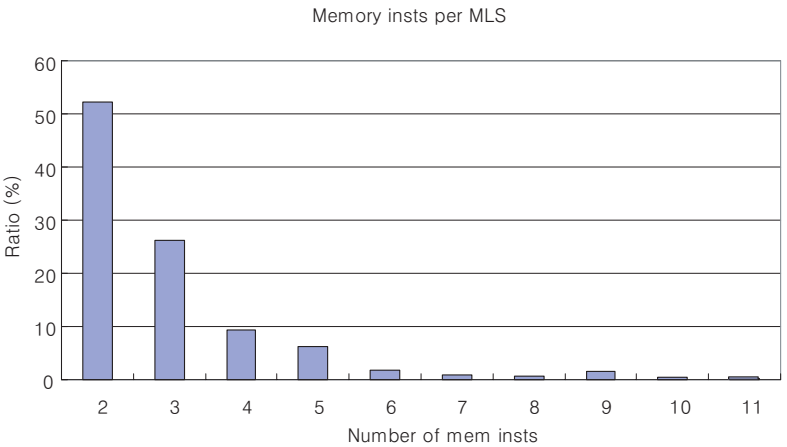


Figure 16. Ratio of MLS instructions with respect to MLS size.

6. COMPARISON WITH PREVIOUS WORK

Finding an optimal memory layout for scalar variables in a stack frame had hardly been a crucial issue in compiler research until about a decade ago when the utilization of special addressing modes became important for typical embedded processors. Since then, there has been much work on code size reduction through optimal memory assignment for such addressing modes. One of the earliest works was undertaken by Bartley [5] who first addressed the SOA problem. Later, Liao *et al.* [1] formally proved that the SOA problem can be reduced to the MWPC problem, and thus that it is NP-complete. Hence, to cope with the SOA problem quickly in polynomial time, they, in common with our approach, proposed a heuristic based on the Kruskal's MST algorithm. They also extended the SOA problem to the GOA problem that handles multiple index registers.

Inspired by the previous work, many researchers extended the work in various aspects. Leupers and David [12] developed a genetic-algorithm-based technique to solve the SOA/GOA problem by simultaneously handling arbitrary register file sizes and auto-increment ranges. Rao and Pande [2] optimized the access sequence of variables by applying algebraic transformations on expression trees to obtain the least cost offset assignment for the SOA/GOA problem. Zhuang *et al.* [13] discussed an approach of variable coalescence which enables both code and data size reduction, and simplifies the access graph yielding better SOA solutions. A similar study was also conducted simultaneously by Ottoni *et al.* [14] who proposed a coalescing SOA-based algorithm that performs variable coalescing and offset assignment simultaneously.

All of these previous studies are in some sense related to our work in that they are aimed at finding an 'optimal memory (offset) assignment', and so are mostly based on the MST algorithm. However, as discussed in Section 3, our work is fundamentally different from previous research as our approach is concerned with solving the MLSP, rather than the SOA/GOA problem. Consequently, while previous techniques are only able to handle static weighted graphs, our approach must be capable of handling

DOI: 10.1002/spe

dynamic weighted graphs, which makes the algorithm more complicated. To the best of our knowledge, two works that addressed the optimization with MLS instructions are [6] and [15]. Nandivada [6] investigated the use of synchronous dynamic random access memory (SDRAM) for the optimization of spill code, and proposed an approach that arranges the variables in the spill area, so that loading to and storing from the SDRAM is optimized by utilizing MLS instructions. This work is different from ours in two respects. First, their technique focuses on improving the execution speed of the code rather than reducing the code size in the sense that it generates MLS instructions only from 'double' load/store. For this reason, we deem their problem to be a special case of the MLSP for double load/store instructions, which is much simpler than the problem we want to solve. Second, their optimization algorithm is based on *integer linear programming* (ILP). In contrast, our approach is based on an efficient graph-based heuristic, which guarantees a polynomial-time bound in run time while producing global solutions.

The work reported in this paper is an extension of some of our authors' early work. In [16], we proposed an integrated approach where the SOA/GOA problem is coupled with instruction scheduling to more efficiently exploit scheduling by minimizing addressing instructions. This differs from our current work as, in common with the majority of the previous work, it targets the SOA/GOA problem. In the other study [17], we targeted the MLSP and proposed a heuristic algorithm to tackle the problem. Although that work offered us a glimpse of the benefit of exploiting MLS instructions in our code generation [18], it was somewhat premature since it has one fundamental shortcoming: the proposed algorithm was not specifically targeted to the MLS problem, owing mainly to an insufficient formal analysis of the problem itself. In addition, the problem MLSP-u was not separately addressed in this preliminary study. This is also the case with another related work by Johnson and Mycroft [15] who conducted a similar experiment independently, and almost simultaneously published a paper that happens to be quite close to our work [17].

7. CONCLUSIONS

This work was motivated by our previous project to build an optimizing compiler for a commercial embedded media processor under development. In the processor, we found a variety of instructions specifically designed to accelerate media applications, and among them there was a class of single-instruction multiple-data (SIMD) instructions, called MLS instructions. In our efforts to exploit a SIMD-style parallelism in memory operations for code optimization, we found that no previous compilers had made a serious attempt to address this optimization problem. For this reason, in our research we chose to devise an effective algorithm that tackles this exponential-time problem quickly. More precisely, in this work we rigorously analyzed the MLSP, from which we proved that it is intractable, and proposed a polynomial-time bounded algorithm, Solve_MLSP, to solve the problem efficiently. Solve_MLSP attacks the problems in three steps: (1) constructing a graph with parallel load/store relations among instructions; (2) formulating the problem into a problem of finding a path cover in the graph; and (3) assigning registers.

From experimental results on a set of benchmark suites in multimedia applications, it is shown that Solve_MLSP was able to reduce the load/store codes by on average 6.5–13.3%, indicating that it exploits MLS instructions successfully to further reduce the size of code even after the code has been fully optimized by existing production-quality compilers. Although our technique cannot reduce the total code size on a dramatic scale (i.e. 3.6%), we believe that it has been proven effective and valuable to the applications of memory-resource-limited embedded system designs.

DOI: 10.1002/spe

REFERENCES

1. Liao S, Devadas S, Keutzer K, Tjiang S. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems* 1996; **18**(3):235–253.
2. Rao A, Pande S. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. ACM Press: New York, 1999; 128–138.
3. ARM. ARM Instruction Set Quick Reference Card. http://www.arm.com [2005].
4. Embedded Concepts and Solutions, Inc. ARM Technical Tidbits. http://www.go-ecs.com [2005].
5. Bartley B. Optimizing stack frame accesses for processors with restricted addressing modes. *Software—Practice and Experience* 1992; **22**(2):101–110.
6. Nandivada V, Palsberg J. Efficient spill code for SDRAM. *Proceedings of the 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM Press: New York, 2003; 24–31.
7. Zivojnovic V, Velarde JM, Schager C, Meyr H. DSPStone—A DSP oriented benchmarking methodology. *Proceedings of the International Conference on Signal Processing and Technology*, Dallas, TX, October 1994; 715–720.
8. Lee C, Potkonjak M, Mangione-Smith W. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society Press: Washington, DC, 1997; 330–335.
9. ARM. ARM Developer Suite—Version 1.2. http://www.arm.com [2005].
10. Franklin M, Wolf T. Power considerations in network processor design. *Network Processor Design–Issues and Practices*, vol. II. Morgan Kaufmann: San Francisco, CA, 2003.
11. Sanchez-Elez M, Fernandez M, Anido M, Du H, Bagherzadeh N, Hermida R. Low energy data management for different on-chip memory levels in multi-context reconfigurable architectures. *Proceedings of the Conference on Design, Automation and Test in Europe*, vol. 1. IEEE Computer Society Press: Washington, DC, 2003.
12. Leupers R, David F. A uniform optimization technique for offset assignment problems. *Proceedings of the 11th International Symposium on System Synthesis*. IEEE Computer Society Press: Washington, DC, 1998; 3–8.
13. Zhuang X, Lau C, Pande S. Storage assignment optimizations through variable coalescence for embedded processors. *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compilers, and Tools for Embedded Systems*. ACM Press: New York, 2003; 220–231.
14. Ottoni D, Ottoni G, Araujo G, Leupers R. Improving offset assignment through simultaneous variable coalescing. *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems*, September 2003 (*Lecture Notes in Computer Science*, vol. 2826). Springer: Berlin, 2003; 285–297.
15. Johnson N, Mycroft A. Using multiple memory access instructions for reducing code size. *Proceedings of the International SIGPLAN Symposium on Compiler Construction* (*Lecture Notes in Computer Science*, vol. 2985). Springer: Berlin, 2004; 265–280.
16. Choi Y, Kim T. Address assignment combined with scheduling in DSP code generation. *Proceedings of the 39th Conference on Design Automation*. ACM Press: New York, 2002; 225–230.
17. Paek Y, Choi J, Joung J, Lee J, Kim S. Exploiting parallelism in memory operations for code optimizations. *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, September 2004 (*Lecture Notes in Computer Science*, vol. 3602). Springer: Berlin, 2005.
18. Paek Y, Ahn M, Lee S. Case studies on automatic extraction of target-specific architectural parameters in complex code generation. *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, September 2003 (*Lecture Notes in Computer Science*, vol. 2826). Springer: Berlin, 2003; 151–166.
19. Buchsbaum A, Giancarlo R, Westbrook J. On reduction via determinization of speech recognition lattices. *Technical Report*, AT&T Bell Labs, 1997.
20. Chaitan G. Register allocation and spilling via graph coloring. *Proceedings of the 1982 International SIGPLAN Symposium on Compiler Construction*. ACM Press: New York, 1982; 98–105.
21. Stallman R. Using the GNU Compiler Collections. http://gcc.gnu.org [2005].

DOI: 10.1002/spe