복수의 메모리 접근 명령어의 효율적인 이용을 통한 코드 크기의 감소
(Code Size Reduction Through Efficient use of Multiple Load/Store Instructions)

안 민 욱 † 조 두 산 † 백 윤 흥 †† 조 정 훈 †††
(Minwook Ahn) (Doosan Cho) (Yunheung Paek) (Jeonghun Cho)

요 약 하나의 instruction으로 여러 메모리 블록을 읽거나 쓰는 MLS(Multiple Load/Store) 명령어를 사용하면 전체 코드에서 메모리 명령어의 수를 최소화해서 코드 사이즈를 축소할 수 있다. 이러한 장점 때문에 많은 마이크로 프로세서에서 이 명령어를 지원하고 있으나 현재까지 개발되어 있는 컴파일러들은 MLS 명령어의 장점을 효과적으로 이용하고 있지 못하고 있고 오직 제한적인 용도로 MLS 명령어를 사용하고 있다. 기존의 컴파일러에서 MLS 명령어를 효율적으로 지원하지 못하는 것은 일반적으로 MLS 명령어를 효과적으로 이용하기 위해서 해결해야 할 문제가 NP-hard의 범주에 속하기 때문이다. 이것은 stack frame에서 변수들에 대한 최적의 메모리 옵셋을 찾는 문제와 레지스터 할당에 관련된 복합적인 문제이다. 본 논문에서는 heuristic 기법을 효율적으로 이용하여 위에 언급된 문제를 polynomial time bound에 해결할 수 있는 기법을 제안한다.
키워드 : 컴파일러, 전산기구조, 복수 로드/스토어 명령어, 코드 사이즈, 임베디드 시스템

Abstract Code size reduction is ever becoming more important for compilers targeting embedded processors because these processors are often severely limited by storage constraints and thus the reduced code size can have a positively significant impact on their performance. Various code size reduction techniques have different motivations and a variety of application contexts utilizing special hardware features of their target processors. In this work, we propose a novel technique that fully utilizes a set of hardware instructions, called the *multiple load/store* (MLS), that are specially featured for reducing code size by minimizing the number of memory operations in the code. To take advantage of this feature, many microprocessors support the MLS instructions, whereas no existing compilers fully exploit the potential benefit of these instructions but only use them for some limited cases. This is mainly because optimizing memory accesses with MLS instructions for general cases is an NP-hard problem that necessitates complex *assignments of registers and memory off-sets* for variables in a stack frame. Our technique uses a couple of heuristics to efficiently handle this problem in a polynomial time bound.
Key words : compiler, computer architecture, multiple load/store instructions, code size, embedded system

· This research was supported in part by ITRC, Young Investigator Research Project, and MPSoC project.
† 비 회 원 : 서울대학교 전기컴퓨터공학부
mwahn@compiler.snu.ac.kr
dscho@compiler.snu.ac.kr
†† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
(Corresponding author임)
ypaek@ee.snu.ac.kr
††† 비 회 원 : 경북대학교 전자전기컴퓨터공학부 교수
jcho@ee.knu.ac.kr
논문접수 : 2004년 8월 27일
심사완료 : 2005년 6월 2일

1. INTRODUCTION

In general, microprocessors contained in an embedded system are heavily limited by resource constraints such as size, power and cost. In particular, as embedded software grows rapidly complex and large to satisfy diverse demand from the market, efficient use of limited storage resources are ever becoming more important for compilers targeting these processors. To attain a

desired performance goal even with such limited storage, the processors are designed assuming software that runs on them would make heavy use of their various special hardware instructions and addressing modes [1, 2]. One such example is the MLS instructions, which are often encountered in modern processors such as Motorola Mcore, ARM 7/9/10, Fujitsu FR30 and IBM R6000. The following shows an example of the MLS instructions in an ARM processor [3]:

$$ldmia\,/\,stmia \quad r_{base}, \{r_1, r_2, \cdots, r_m\}$$

where m≤16 and all the operands $(r_{base}, r_1, r_2, \cdots, r_m)$ can be any of the ARM general-purpose registers: $r0, r1, \cdots, r15$.

These instructions allow large quantities of data to be transferred more efficiently in a single operation 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 instance, the instruction, $ldmia \quad r1, \{r3, r4, r8\}$, loads a block of three words

$Mem[r1], Mem[r1+4]$ and $Mem[r1+8]$

respectively into the registers in an increasing order of their numbers, that is, $r3$, $r4$ and $r8$. The order of registers appearing inside the braces does not affect the data transfer result. The main advantages of MLS instructions are two fold:

- code size reduction through compaction of a number of memory operations into one instruction word[1], and
- running time reduction through pipelining of memory accesses.

To illustrate this, consider the example in Figure 1, which shows the C source code and its assembly translated by a commercial ARM native compiler. The figure also shows the memory offsets the compiler assigns the variables in the stack. From this memory layout, we can see that the ARM compiler, like many other conventional ones, performs memory assign-ment simply in a declaration or lexicographic order of variables. This naive approach does not satisfy the above requirements for *registers* and *memory offsets* in an MLS instruction and so impedes the chance to use it in the code. In consequence, even after traditional optimizations such as redundant load and store elimination, the ARM compiler output in Figure 1(c) still has 17 loads/stores in total, including just a single multiple store generated from *double stores* (i.e., two stores). Furthermore, even this single use of the MLS may not bring any advantage in the code size, because one ALU instruction, *add* $r1$, *sp*, #8, was inserted to initialize the base register $r1$ for the MLS. This fact implies that the ARM compiler needs a much more aggressive approach for *memory* and *register assignment* to take full advantage of hardware support for the MLS.

```
   ...
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) A fragment of C source code

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

(b) The original memory layout in the stack frame represented in relative offsets from the stack pointer

```
         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 |11.48|
         mov r1,r2
         bl ||qstring||
         ldr r0, [sp,#4]
         ldr r1, [sp,#0xc]
         sub r0,r0,r1
         str r0, [sp,#4]
         b |11.88|
|11.48|  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}
|11.88|  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) ARM native compiler output

Figure 1 Benchmark code, and its assembly with the memory layout generated by the ARM native compiler with a-O2 optimization option

In this paper, we report our recent study on a problem, which we name the *MLS problem*, whose goal is to find an optimal register and memory

1) In an MLS instruction, each register operand is encoded as a single bit.

assignment that leads us to fully capitalize on the advantages of MLS instructions. In our example, an ideal memory assignment for this problem would be the one in Figure 2(a).

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

(a) Optimal memory assignment for MLSs

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

(b) Modified ARM code output with the memory assignment in (a)

```
        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}
```

(c) Optimized code output both after memory and register assignment

Figure 2 Improving the original code by using MLSs with better memory & register assignments for local variables: To obtain (c), local instruction scheduling and other traditional optimizations were applied

Reflecting this new memory layout, we may change the original code as listed in Figure 2(b), where we notice that more neighboring loads/stores are accessing contiguous addresses under this layout. But we also notice that despite this improved memory assignment, they still cannot be converted to MLS instructions. This is of course because without a proper register assignment, a smart memory assignment alone cannot enable neighboring loads/stores to aggregate into an MLS. For instance, although the last two stores in Figure 2(b) access contiguous addresses (0×10, 0×14), an MLS instruction cannot be generated for them since it stipulates that a lower address be assigned a lower register number in its operand. Thus, in this case, if the address 0×14 is assigned the register r1, then the other one 0×10 must be re-assigned the register r0.

This example convinces us that optimizing a program with MLS instructions is quite a complex memory assignment problem tangled with a register assignment (or we may call *register renaming*) problem. On top of this, when we solve this problem, we must also consider *instruction rescheduling* simultaneously because a single large memory transfer formed together from many small ones scattered in the code mostly results in better performance.

Our final code optimized with ldmia/stmia is shown in Figure 2(c). As compared to the original code, the code size is reduced about 20%. If we only consider memory operations, the number of loads/stores are reduced about 60%. Since memory accesses in an MLS can be overlapped by pipelining when data are actually transferred, we may also expect some significant amount of reduction in the total memory access time.

In the optimized code, there are three MLS instructions, each of which is combined from double loads/stores. At first glance, they seem unprofitable in terms of code size, as we already stated, due to additional ALU operations for base register initialization. But we sometimes found in real cases that these ALU operations are often exposed to conventional optimiz-ations such as common subexpression elimination(CSE) and redundancy elimination, and thereby some of them were removed. In this example, one add was removed since the base address in the register r10 is reused by two of the MLS instructions. In fact, note in the code that one more add was removed for another MLS (converted from triple stores) with r10 as its base register. Due partially to temporal locality embedded in real code, we observed this reuse of base addresses was made possible in our experiments.

To summarize, finding an optimal solution to the MLS problem is an extremely difficult task complicated with several NP-complete optimization subproblems. Therefore, it is not surprising to discover that all the compilers we tested in our

experiments fail to utilize the MLS instructions to such a degree as we demonstrated in Figure 2, and only use them for special occasions, such as exception handlers, function prologues/epilogues and context switches [4], where recognizing block memory copies for MLS instructions are rather trivial. All this inevitably implies that to utilize MLS instructions, the users should hand-optimize their code in assembly, making programming complex and time consuming.

The purpose of this paper is to discuss our heuristic-based algorithm that solves the MLS problem efficiently in a polynomial time bound. Although our algorithm does not guarantee to find an optimal solution to the MLS problem for all cases, our experiment with real benchmark programs exhibits its effectiveness on most cases.

In Section 2, we start our discussion by relating our study with previous ones that worked on broadly similar but completely different problems from ours. In Section 3, we describe our algorithm, and in Section 4, present our experimental results and compare ours with other compiler results. In Section 5, we conclude.

2. RELATED WORK

Memory assignment of 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 work was done by Bartley [5] who first addressed the *simple offset assignment* (SOA) problem, the problem of assigning scalar variables to memory such that the number of explicit address arithmetic instructions are minimized by using auto-increment/decrement addressing modes. Later, Liao et al. [1] formally proved that the SOA problem can be reduced to *the maximum-weighted path cover* (MWPC) problem, and thus that it is NP-complete. Hence, to cope with the SOA problem fast in polynomial time, they proposed a heuristic based on Kruskal's *maximum spanning tree* (MST) algorithm.

Inspired by the previous work, many researchers extended the work in various aspects. Leupers and David [6] developed a genetic algorithm-based technique to solve the SOA 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 problem. Choi and Kim [7] proposed an integrated approach where the SOA problem is coupled with instruction scheduling. Zhuang et al. [8] discussed an approach of variable coalescence which enables both code and data size reduction. A similar study was also conducted at the same time by Ottoni et al. [9] who discussed an approach based on Coalescing SOA algorithm that performs variable coalescing and offset assignment simultaneously.

All these previous studies are in some sense related to ours in that they aim at finding optimal memory (offset) assignment for certain hardware instructions or addressing modes. But they are also clearly different from ours in that they all center around only the SOA problem, which varies from our MLS problem in many ways. In fact, our problem subsumes the SOA problem because as discussed in Section 1, ours must find not only optimal memory assignment but at the same time optimal register assignment along with the load/store scheduling that facilitates maximum utilization of MLS instructions. This means that we need a more aggressive approach to handle our problem than previous studies.

To our best knowledge, the only study on code optimization with MLS instructions was published most recently by Nandivada and Palsbergby at Purdue [10]. In their study independent from ours, they investigated the use of SDRAM for optimization of spill code. The core of their problem is to arrange the variables in the spill area such that loading to and storing from the SDRAM is optimized with MLS instructions. Their work differs from ours in two key points.

First, their technique focuses on running time not code size in the sense that, as explained in Section 1, it generates MLS instructions only from double

loads/stores. For this reason, we deem their problem to be a special MLS problem for double loads/stores, which is simpler than our general problem. Second, their algorithm is based on *integer linear programming* (ILP). As mentioned above, the MLS problem is composed of several optimization subproblems tightly coupled to each other. So, the ideal strategy for this would be to solve the whole problem in a single, combined phase, where all subproblems are simultaneously considered. In their approach, therefore, they converted their problem to an ILP problem so that they solve it in a *coupled*, single phase. Obviously one critical drawback of such a coupled approach is that it uses an exponential time algorithm as they also did in their work. To avoid this excessively high time complexity, we take a more *decoupled* approach where we apply fast polynomial-time algorithms to solve each subproblem in a sequential, step-by-step manner. In Section 3, we present our algorithm based on this decoupled approach.

3. SOLVING THE MLS PROBLEM

In our approach, the information about memory access patterns is first culled from the code and summarized in a graph form. In the next phase, this form is used to find an optimal schedule for MLSs. Then, this load/store scheduling result is in turn used to determine the best possible offsets of variables in memory. Only after code has been scheduled and compacted by MLSs with all their variables fixed to memory, comes the assignment of physical registers to the variables. In this section, we discuss how the original MLS problem is divided in these phases and how each phase is structured in a decoupled fashion.

3.1 Dividing the Problem in Three Phases

To more formally describe the MLS problem, we define *parallel loads* and *parallel stores* to be respectively a block of loads and stores that can be executed simultaneously. In our approach, they are identified from the code as the first step of generating MLSs. Ideally, each parallel load/store block can be scheduled together and converted directly to an MLS instruction. However, the conversion is not always so straightforward because it stipulates beforehand that all the three constraints below be satisfied.

For the definition of the constraints, we follow the convention of the ARM architectures (see Section 1) to assume our MLS instructions is of the form:

$\{r_1, r_2, ..., r_m\}$ = Mem[r_{base}] // *multiple load*

Mem[r_{base}] = $\{r_1, r_2, ..., r_m\}$ // *multiple store*

where $m \leq n$ and n is the number of general-purpose registers on the target architecture.

RF-size constraint: This is enforced because the MLS problem involves the instruction scheduling issue. When loads/stores are moved together to form a single MLS, it normally increases the life span of each value associated with them and so the overall register pressure in the code. Therefore, if there exists a point where the register pressure becomes higher than the actual *register file size,* then some of those loads/stores responsible for it may not be executed in the same MLS.

M-sequence constraint: The sequence of the memory locations where the m words, $m \leq n$, are fetched must be contiguous starting from the address specified by the content of r_{base}: $Mem[r_{base}], Mem[r_{base}+4], ..., Mem[r_{base}+4m-4]$.

R-sequence constraint: The number sequence of the m registers where the memory data are transferred may not be contiguous, but the sequence must be strictly increasing.

Simply stated, the MLS problem is of finding optimal blocks of parallel loads/stores subject to these constraints. In our decoupled approach, it is divided and conquered individually in separate three phases, as portrayed in Figure 3. Starting from the graph called *LS-regions*, the initial information about parallel loads/stores is gradually shaped into the final form, called an *mPLS graph*, as each constraint is applied in the subsequent phases. At the end of Phase 3, the parallel loads/stores remaining in an mPLS graph satisfy all the constraints; thus each block of them can now be converted safely to an MLS.

In the next subsections, we will detail each phase of Figure 3. In our algorithm, we assume that conventional code generation with register allocation has already been performed for our input code. For

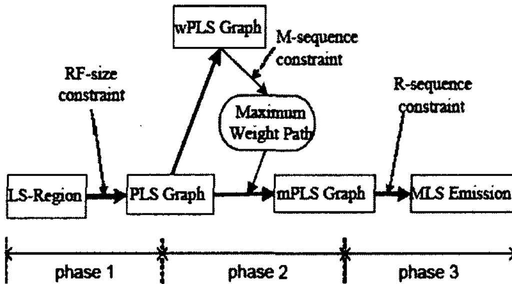


Figure 3 Gradually shaping up the parallel load/store blocks with each constraint in three for MLS instruction generation

simplicity, we also assume no memory address aliasing.

3.2 Computing Parallel Loads and Stores

Figure 4 shows the subroutine **build_PLS** for Phase 1 whose task is to identify all parallel loads and stores in each basic block of the input code. To describe this task, we first define two types of regions in Definitions 1 and 2, where given a block B with size $|B|$, we assume the cycles in B count from 0 up to $|B|$-1.

DEFINITION 1. Suppose 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/upper* bounds *lb* and *ub* are defined as follows.

- If there occurs the last store into 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 2. Suppose 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/upper bounds *lb* and *ub* are 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 at the cycle t″ in B after the store, then $int_v.ub = t''$.

```
build_PLS(P):
  G_PLS ← ∅;   // PLS graph
   for each basic block B in P do
   R_L ← compute_L_regions(B);   // according to Definition 1
   R_S ← compute_S_regions(B);   // according to Definition 2
   R ← R_L ∪ R_S;   // a unified set of L/S-regions
   while R ≠ ∅ do
     select int_v ∈ R such that ∀int_u ∈ R, int_v.ub ≤ int_u.ub;
     I ← {int_v};   // We call int_v the seed interval of I
     if int_v ∈ R_L then   // loadable region
       add to I all the L-regions in R overlapping with int_v;
     else   // storable region
       add to I all the S-regions in R overlapping with int_v;
     // check if register pressure > RF_size at t_vio
     t_vio ← check_RFsize(I, B);
     while (t_vio > 0) do   // RF-size constraint violated
       remove int_w from I such that ∀int_u ∈ I, int_w.ub ≥ int_u.ub;
       int_w.lb ← t_vio + 1;
       t_vio ← check_RFsize(I, B);
     od
     // construct a complete graph with all elements of I as its nodes
     C ←build_complete_graph(I);
     if int_v ∈ R_L then   // loadable region
       C.gen_time ← max_{int_u ∈ I} int_u.lb;
       C.type ← load;
     else   // storable region
       C.gen_time ← min_{int_u ∈ I} int_u.ub;
       C.type ← store;
     fi
     add C to G_PLS;
     R ← R − I;
   od
  od
  return G_PLS;
end
```

Figure 4 Greedy algorithm designed to find parallel loads/stores

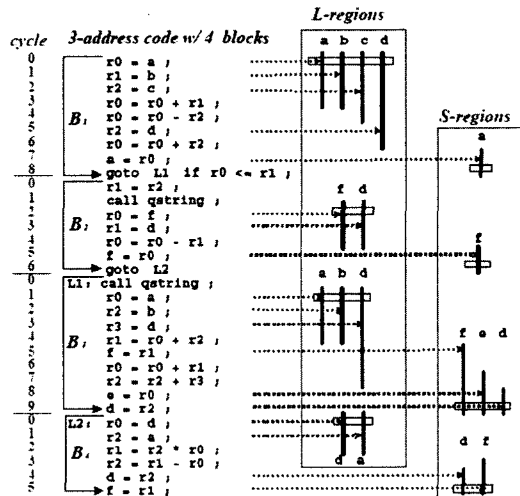


Figure 5 4 basis blocks (B_1,B_2,B_3,B_4) of the 3-addess code rewritten from Figure 1(c) and the L/S-regions for loads/stores in each block

Otherwise, $int_v.ub = |B|-1$.

The L- and S-regions respectively represent the maximum ranges within B where the load r = v

and the store v = r can move without violating data dependences on the variable v. Examples of the L/S-regions are shown in Figure 5 where the original ARM code (see Figure 1) is translated into 3-address form for better readability. The vertical bars in the figure stand for the L/S-regions of each load/store. For instance, the load from the variable *d* in the basic block B_1 has the L-region stretching from the cycle 0 to 6 because by definition, there is no store before the load in B_1 and the value loaded to *r*2 is first used at the cycle 6. Similarly, the S-region for the store into the variable *f* in the block B_3 stretches from the cycle 5 to 9 since the stored value is defined at the cycle 4 and there is no load from *f* in B_3 after the store.

The routine build_PLS uses the L/S-regions to identify parallel loads/stores. For this, it first divides the input procedure P into basic blocks, and for each block, computes the L/S-regions R_L and R_S according to Definitions 1 and 2. In principle, any loads/stores are parallel as long as their L/S-regions are overlapped. So in build_PLS, these parallel loads/stores are initially all gathered into the same block I of parallel loads/stores. However, this simple gathering may cause many new register spills in the final code. To explain this with an example, suppose in Figure 5 that we combine a load for *d* in B_1 to the same I with those for *a*,*b* and *c*. Then, when we generate MLS instructions, we will schedule these four loads into the same MLS so that by definition they can be executed *simultaneously*. This inevitably means that the load for *d* should move up from the cycle 5 to 2 or even earlier. This movement would prolong the life time of the value in the register *r*2, possibly increasing the register pressure as well. This can be a major drawback for us because we might have to generate MLSs with extra spills due to the increased register pressure, and very likely, these spills would offset the gains from our MLS uses in terms of code size and running time. To prevent this potential problem, the subroutine **check_RFsize** invoked inside build_PLS enforces the RF-size constraint when parallel loads/stores are collected to I.

When check_RFsize reports that the current configuration of I violates the RF-size constraint, some L/S-regions (or we may say simply *intervals* by their definitions) are removed from I until the constraint is satisfied. To explain this, consider Figure 6 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 code. In figure 6 (a), the block *I* of L-regions are first formed with four intervals $(\mathrm{int}_x, \mathrm{int}_y, \mathrm{int}_z, \mathrm{int}_v)$, beginning with int_z as the *seed interval.*[2] However, under the register file size limit (= 4), moving up the two loads for v and y to join I before the cycle 2 would cause the resulting pressure to violate the RF_size constraint by exceeding the limit at the cycle 4. When this violation is reported, build_PLS forbids the load for v to move up before the cycle 5 by eliminating it from I and adjusting its lower bound $int_v.lb = 5$, as shown in Figure 6 (b), where we now can 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 overlapping with other intervals.

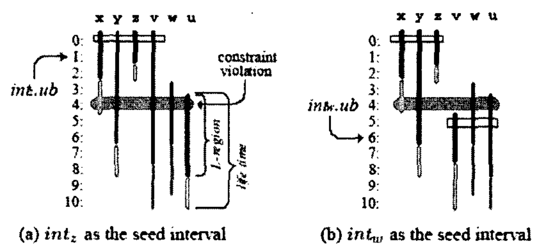


Figure 6 RF-size constraint validity check for register pressure=4

After int_v is removed, the three intervals remaining in I will form a block of parallel loads, as shown in Figure 6(b). Then, by definition, the interval int_w will be selected as the next seed, and clustered with the other two intervals, int_v and int_u.

The output of build_PLS is an undirected graph

2) the interval whose upper bound is the lowest of all in the same block

G_{PLS}, called the *parallel load/store* (PLS) graph, which is a collection of disconnected subgraphs. Each subgraph is constructed from a block of parallel loads/stores by turning an interval in the block to one node in the subgraph. Figure 7 shows the PLS graph built from the intervals in Figure 5. Each subgraph forms a complete graph because the edge is designed to represent the parallelism between two loads in the code and parallelism is an equivalence relation.

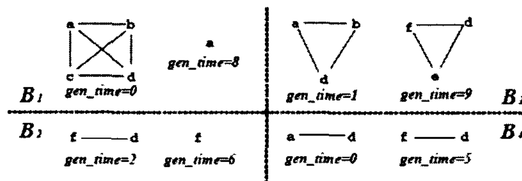


Figure 7 The PLS graph built from the code in Figure 5: Each node v in the graph represents the interval of a load/store for variable v.

Later in Section 3.4, we will see that each subgraph in the PLS graph is given the code generator to emit MLS instructions in the final code. When an instruction is emitted, the generator needs to know where in the code it must be scheduled. To supply this information, every subgraph is associated with an integer *gen_time* that records the time just before which an MLS instruction for the subgraph is inserted in the code. To minimize pipeline hazards, gen_time is set as early as possible to schedule loads and as late as possible to schedule stores. These times are denoted by the horizontal white bars in Figures 5 and 6.

When we build a PLS graph in the routine build_PLS, we follow a greedy approach by choosing as a seed interval an interval with the lowest upper bound among all remaining ones. The number of disconnected subgraphs in the PLS graph is proportional to that of loads/stores emitted in the final code. This means that the less subgraphs we produce in build_PLS, the more likely we will have an optimal code in the end. Luckily, the following theorem proves that our greedy approach reaches an optimal solution.

THEOREM 1. *Given a set of loads/stores in the basic block B, the routine* **build_PLS** *finds the minimum number of parallel loads and stores for B.*

PROOF: The proof is straightforward. Suppose build_PLS produces k blocks of parallel loads/stores. Let S be the set of all seed intervals. The seed intervals are all disjoint since otherwise some of them would be included in the same block of parallel loads/stores. Since |S|=k, k is the minimum number of parallel loads/stores. ■

3.3 Memory Assignment

In Phase 1, we imposed the RF-size constraint on the initial parallel load/store blocks when we summarized them as the disconnected subgraphs in a PLS graph. The task of Phase 2 is to impose the M-sequence constraint on them in an attempt to find an optimal memory layout for variables that can minimize the number of loads/stores emitted in the final code, as depicted in Figure 8.

```
solve_MAM(P, G_PLS)
   G_wPLS ← build_wPLSG(G_PLS);   // according to Definition 5
   mwp ← solve_SOA(G_wPLS);
  // Maximum-weight path algorithm from [1]
   E_np ← get_non_path_edges(G_wPLS, mwp);
   for all complete graphs C = (V_C, E_C) ∈ G_PLS do
      if ∃(u, v) ∈ E_np such that (int_u, int_v) ∈ C then
         remove (int_u, int_v) from E_C;
   od
 ○ for every variable v ∈ P do
      v.offset ← assign_offsets_in_memory(v, mwp);
   return G_PLS;   // return PLS graph as mPLS graph
end
```

Figure 8 Phase 2 algorithm to solve the MAM problem

In principle, we can simultaneously run any parallel loads/stores in the same block by emitting them in one MLS instruction. However, the M-sequence constraint tells us that the memory locations accessed in an MLS should be contiguous. This means that if parallel loads/stores are referencing variables with non-contiguous memory offsets, they should be scheduled to more than one different MLS instructions, which will certainly result in an increase in code size. So, the ultimate problem we need to solve in Phase 2 is how to find such memory offsets for variables that satisfy the M-sequence constraint and at the same time, minimize the number of parallel loads/stores that

must be scheduled to different MLS instructions. For clarity, this problem, which we call the *memory assignment for* ***MLS*** (MAM), can be stated more formally below.

DEFINITION 3. Given a set of local variables S, let [v] denote a memory offset of $v \in S$ in a local stack. We define $<$ to denote a relation between u and v in S such that $u < v$ iff u immediately precedes v in the stack, that is, [v] = [u] + 4. From the relation $<$, we define a reflexive transitive closure $<^*$ on S as follows:

- $\forall v \in S : v \prec^* v$;
- $\forall u, v \in S : \exists w \in S : u \neq v, u \prec^* w$ and $w \prec v$ iff $u \prec^* v$

DEFINITION 4. Given a PLS graph $G_{PLS} = (N,E)$, let U be a set of all possible partitions Ψ of N such that $\forall \psi \in \Psi$, the block ψ satisfies the following conditions:

1. $\forall \mathrm{int}_u, \mathrm{int}_v \in \psi : (\mathrm{int}_u, \mathrm{int}_v) \in E$;
2. $\forall \mathrm{int}_u, \mathrm{int}_v \in \psi :$ either $v \prec^* u$ or $u \prec^* v$.

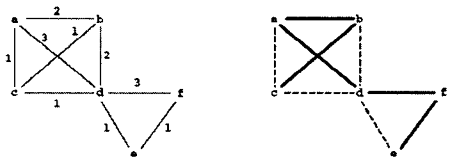
Then, the MAM is a problem that finds a partition $\Psi \in U$ such that $\forall \Phi \in U, |\Psi| \leq |\Phi|$.

The first condition in Definition 4 implies that a partition Ψ consists of disjoint subsets of N each of which corresponds to a block of the parallel loads/stores summarized in G_{PLS}. On such a partition Ψ, the second condition imposes the M-sequence constraint. Since the possible number of all partitions of N is $O(2^{|N|\log|N|} / \sqrt{\log |N|})$[11] bound by an exponential in |N|, it requires an exponential-time algorithm to optimally solve the MAM problem, like other memory assignment problems listed in Section 2. Considering |N| can be several orders of 10 in real code, therefore, we devise a heuristic that solves the problem fast in polynomial time. To attain this goal, we first transform the MAM problem to an MWPC problem even though the MWPC problem is NP-complete. This is because as discussed in Section 2, the MWPC problem is already well-understood and solved with many powerful algorithms thanks to numerous previous studies. For this transformation, we build a weighted graph, called the *wPLS* graph, as described in Definition 5.

DEFINITION 5. Let V be the set of all variables declared in a procedure P. Let $G_{PLS} = (N',E')$ be the PLS graph built for P by the routine build_PLS. Then, the wPLS graph $G_{wPLS} = (N, E)$ is defined as follows.

- $N = V$
- $\exists e = (u,v) \in E$ if $\exists (\mathrm{int}_u, \mathrm{int}_v) \in E'$.
- For $e = (u,v) \in E, e.weight,$ the weight e, is the total number of edge $e' \in E'$ such that $e' = (\mathrm{int}_u, \mathrm{int}_v)$.

Figure 9(a) shows the wPLS graph generated from the PLS graph in Figure 7. The weight on an edge (u, v) equals to the number of times the variables u and v can be either loaded or stored in parallel.



(a) wPLS built from Figure 7

(b) Maximum-weight path in solid lines and non-paths in dotted lines

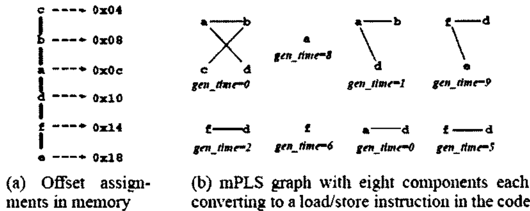
Figure 9 wMPLS graph and the maximum-weight path on it

After a wPLS graph being constructed, the MAM problem in effect reduces to a simpler, yet still exponential-time MWPC problem. To efficiently solve this problem, therefore, we resort to a widely-known heuristic based on an MST algorithm that has also been used to solve the SOA problem [1].

Figure 9(b) shows the resulting *maximum-weight path* (MWP) computed by this heuristic-based algorithm. The MWP is constituted by a set of edges in thick solid lines, and all the other edges in dotted lines form what we call a set of *non-paths* (= E_{np} in Figure 8). The rationale for this use of a MST-based heuristic relies on our speculation that assigning contiguous addresses to

the variables that can be more often accessed together is more likely to increase the chance better utilizing MLS instructions, thus reducing the overall cost of memory accesses.

The resultant MWP is now used to determine the offsets of variables in memory as shown in Figure 10(a). It is also used to measure the total number of memory operations required for this program by removing the non-path edges of the wPLS graph from the original PLS graph. We abbreviate this modified PLS graph as an *mPLS* graph. Figure 10(b) shows that after non-paths are removed, the mPLS graph is no longer a collection of complete graphs, but instead that of connected components. In most cases, the number of those components will become the number of required memory operations in the final code. From Figure 10(b), we can estimate that the program will probably need eight load/store instructions in total, including both single and multiple loads/stores.



(a) Offset assignments in memory (b) mPLS graph with eight components each converting to a load/store instruction in the code

Figure 10 Momery offset assignment for local variables and the modified PLS graph both determine according to the MWP in Figure 9

Although in this example the number of connected graphs does not increase when we modify the PLS graph, in reality we have seen several cases where it actually does. This is of course in part because our heuristic cannot always find an optimal solution. For instance, suppose our algorithm finds the path c-b-a-d-e-f as the result instead of the optimal path c-b-a-d-f-e. Then, the complete graph f-d in Figure 7 would be split into two separate graphs f and d since the edge (f, d) belongs to the non-paths of the wPLS graph. This would result in 10 connected components in the mPLS graph, producing more loads/stores in the code.

3.4 Register Assignment

Every connected component in the mPLS graph corresponds to 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 is now satisfied, each component with k nodes can be converted to a sequence of code either for a multiple load

$$r_{base} = sp + \# m_{base}$$

$$\{r_1, r_2, \ldots, r_k\} = Mem[r_{base}].$$

or for a multiple store

$$r_{base} = sp + \# m_{base}$$

$$Mem[r_{base}] = \{r_1, r_2, \ldots, r_k\}$$

Of course, if the component has only one node, as in the cases of a and f in Figure 10(b), then it will be converted to an ordinary single load/store.

As displayed in the algorithm of Figure 11, this whole conversion process is completed in Phase 3 after a valid order of the registers referenced in each MLS instruction is determined by enforcing the R-sequence constraint. The definition of the R-sequence constraint in Section 3.1 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 an increasing order of the register numbers.

Given a connected component with k nodes, the first part of the constraint implies that at least k registers must be available for the MLS instruction. For instance, in Figure 2(c), the first MLS instruction generated for the four loads from the variables a, b, c and d requires four registers. But as can be seen from the original code in Figure 1 (b), c and d were assigned to the same register r2. Therefore, when we generate an MLS for them, we need to allocate one more register (r3 in this case).

Fortunately, the first part of the R-sequence constraint is trivially met in Phase 3 since the RF-size constraint ensures that the register pressure always stays within the register file size. So, as long as we handle load instructions, we will

always find as many registers as we need to generate them. To the contrary, when we handle store instructions, we may not find, although not common in practice for several reasons, an enough number of registers that we need to generate them because we need one more register as the base register r_{base} for each multiple store. In case none is available for the base register, we generate an extra store for one of the registers included in the original multiple store, as shown in Figure 11. Then, this register becomes free and can be provided as the base register for the rest of the registers. In the case of a multiple load, we can avoid this additional work by arbitrarily designating the base register to be the one from the registers already allocated for the load.

```
emit_MLSs(P, G_mPLS)
  for each connected component C ∈ G_mPLS do
    Bind ← ∅;  // set of tuples <r,v>: binding register r to variable v
    t ← C.gen_time;  // the cycle just before which an MLS is inserted
    k ← # of nodes in C;  // = # of registers for this MLS instruction
    L_var ← a list of variables v such that int_v ∈ C;
    L_reg ← get_free_registers(k);  // a list of registers
    for all r ∈ L_reg selected in an increasing order of their numbers &
        all v ∈ L_var selected in an increasing order of their offsets do
      insert <r,v> into Bind;
    if C.type = load then
      if k = 1 then  // generate a single load at t in P
        emit_load(P, t, Bind);  // a load: r = v
      else  // k > 1: generate a multiple load at t in P
        r_base ← select_first_reg(Bind);  // select 1st register ∈ Bind
        m_base ← find_base_offset(L_var);
        emit_base_register(P, t, r_base, m_base);  // r_base=sp+#m_base
        emit_mload(P, t, Bind);  // {r1,r2,··· ,rk}=Mem[r_base]
      fi
    else  // C.type = store
      if k = 1 then  // generate a single store at t in P
        emit_store(P, t, Bind);  // a store: v = r
      else  // k > 1: generate a multiple store at t in P
        if free register available for base register then
          r_base ← select_free_reg();  // return a free register
          m_base ← find_base_offset(L_var);
          emit_base_register(P, t, r_base, m_base);  // r_base=sp+#m_base
          emit_mstore(P, t + 1, Bind);  // Mem[r_base]={r1,··· ,rk}
        else  // make free the 1st register r1 in Bind
          remove <r1,v1> from Bind;
          Bind' ← {<r1,v1>};  // spill r1 to use it as r_base
          remove v1 from L_var;
          emit_store(P, t, Bind');  // a store: v1 = r1
          m_base ← find_base_offset(L_var);
          emit_base_register(P, t, r1, m_base);  // r1=sp+#m_base
          emit_mstore(P, t + 1, Bind);  // Mem[r1]={r2,··· ,rk}
          emit_load(P, t + 2, Bind');  // reloading: r1 = v1
        fi
    fi
    rename_registers_in(P, Bind);
  od
  eliminate_redundancy_in(P);  // CSE and redundancy elimination
end
```

Figure 11 Phase 3 algorithm for register (re-)assignment

Once an enough number of registers are allocated for an MLS instruction, the second part of the R-sequence constraint is enforced when these registers are bound to each variable in the instruction. To explain this, consider the example in Figure 1(b) where the first three loads are accessing contiguous addresses: 0×10 for c, 0×14 for b and 0x18 for a. Even if the loads are satisfying all other constraints, the ARM compiler could not convert them to a multiple load because they still do not satisfy the R-sequence constraint; that is, a was assigned to r2, b to r1 and c to r0.

Since the first part of the R-sequence constraint is already satisfied, finding an binding between variables and registers that satisfies the second part is straightforward in our algorithm. However, after a load/store instruction is inserted in the code, we may need do an extra chore for appropriate register renaming in the code so as to reflect the new binding made in the instruction.

4. EXPERIMENT

The effectiveness of our 3-phase algorithm on the MLS problem has been evaluated with a set of benchmarks from the *DSPStone* [12] and *MediaBench* [13] suites. The evaluation was conducted on an ARM 7 processor in two different experiments. In the first, our compiler [14, 15] was used to compare the size of both versions of the code output generated before and after the algorithm is applied to the compiler. In the second, our technique was tested with other existing compilers that have already been targeted to the ARM processor. In this section, we report our empirical results.

4.1 Comparison between Before and After

In the first experiment, we generated two versions of the ARM assembly code from a set of DSPstone benchmarks. The first version was generated mostly by using single load/store instructions, with the exception of procedure boundaries where MLS instructions were limitedly used to save a few status registers. The second was generated after applying our algorithm to the first version. Figure 12 compares the code size of the two versions, each respectively denoted by *Before* and *After* in the legends. For each benchmark, the upper bar stands for the code size of the first version, and the lower for that of the second one.

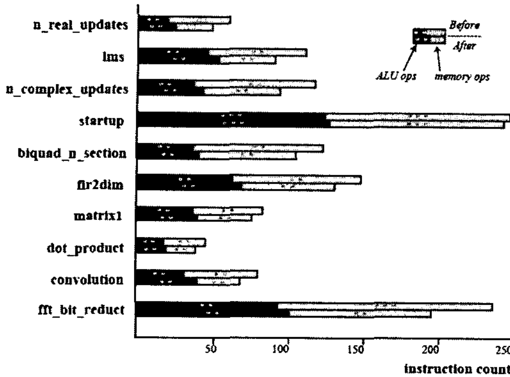


Figure 12 ARM code size before and after our technique is applied

Each bar is halved in two sections. The dark section represents ALU operations, and the light section represents memory operations (i.e., loads/stores).

The amount of code size reduction gained from our MLS generation ranges from 1% to 23%. On average, the overall code size reduction rate is roughly 10%. In the figure, we see that there is a slight increase in the number of ALU instructions for every program. This is certainly because base registers should be initialized before MLS instructions.

For a couple of reasons, we argued in Section 3 that the performance gain with a reduction of loads/stores often outweighs the performance loss with an equal amount of the increase of integer adds for base register initialization. To support this argument, in Figure 13, we single out the effect of load/store reduction and evince the performance benefits we achieve in memory accesses. In the figure, the vertical dashed line denotes the original number of loads/stores in the first version, normalized to one. The horizontal bar represents a reduced code size ratio of loads/stores in the second version against those in the first version. This performance figures reveal to us that the average reduction ratio is approximately 30%, and thus prove that given a set of loads/stores, our technique can reduce the number of loads/stores substantially.

Often many embedded system designers are only allowed limited storage space for their hardware

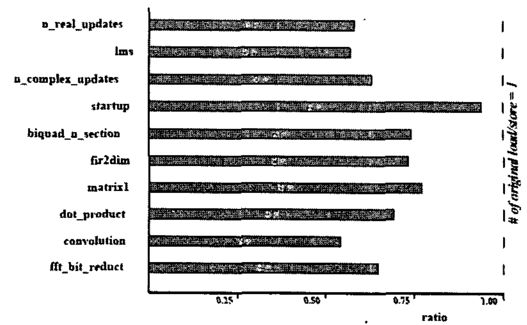


Figure 13 Ratio = $\frac{\#\ of\ loads/stores \in 2^{nd} version}{\#\ of\ loads/stores \in 1^{st} version}$

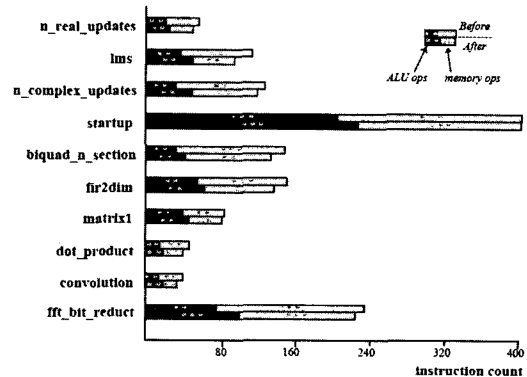


Figure 14 Code size reduction on a modified ARM with 8 registers

due to stringent resource constraints. Thus, we investigate the effectiveness of our technique on an architecture with a less number of registers than the original ARM processor. For this experiment, the compiler is retargeted to the same ARM architecture but with the register file size reduced by half, that is, 8 registers in total. Figure 14 compares the code size of the two versions on this new target machine.

As can be expected, the reduced file size increases register spills, producing more loads/stores in the code. However, except for the code startup, the code size has been just slightly increased for all the others. So, for these codes, we achieve almost identical code size reduction ratios in Figure 14 as we did in Figure 12.

One interesting code we concentrate on here is startup where we have about twice more loads/stores due to new spills generated after the

register file size is reduced. Notice in Figure 14 that our technique does not perform better even with these more loads/stores in the code. According to our hand analysis, the major reason is that traditional register allocation algorithms [16] generate spills without considering its effect on MLS operations. This analysis provides us with an insight that the register allocator may reduce spill costs if it is able to identify a set of the registers to spill which will be more likely spilled and reloaded together with a single MLS operation. As our future study, we are currently working on this issue to improve our register allocator for the MLS instructions.

4.2 Evaluation on Existing Compilers

More recently, we conducted another experiment where we evaluated our technique in comparison with two existing compilers:

- the *GNU C Compiler* (GCC) 3.3, a public domain compiler ported to the ARM processor [17]
- a production-quality ARM native compiler from the *ARM Developer Suite* (ADS) 1.2 [18]

In general, it is difficult, often impossible, to make a fair comparison between two different compilers and pinpoint the precise effect of a single technique on each one because a compiler usually has its own unique infrastructure with a different mixture of compilation techniques. Therefore, instead of directly comparing our compiler with others, we applied our technique to the assembly output of each compiler and optimized it with MLS instructions. Then, we measured the amount of code size reduction in each compiler. Table 1 lists the measurement results for individual procedures taken from the JPEC, MPEG2 and RASTA packages in the MediaBench suite. The numbers in the table are the instruction counts for each benchmark. They are measured in three categories. Firstly, the total instruction count is measured. Then, the number of memory instructions in the code is measured. Lastly, the MLS instructions among the memory instructions are counted.

The benchmark code is compiled with the full optimization level of each compiler so that we can allow them to maximize the chance of applying all their optimization techniques for code size reduction before the code is further reduced by our technique. Even with these full optimization levels, our experiment reveals us that both the compilers fail to fully exploit MLS instructions for reduction of code size. Due to lack of a powerful technique for the MLS problem, the compilers only use them mainly for restricted purposes.

Table 1(a) exhibits that on average our technique discovers about five times more MLS operations from the code than the GCC compiler. According to our hand analysis, our polynomial-time algorithm arrives at the solutions very close to optimal in all cases. This efficiency of our algorithm helps us to reduce the number of loads/stores by 28% on average. In all, despite the addition of ALU operations for base register initialization, we achieve almost 6% of the average reduction ratio for the total code size.

We achieve slightly worse performance with the ADS compiler than we do with the GCC compiler. This is apparently because the ADS compiler is commercialized to produce higher-quality code. So in Table 1(b), we see that our technique identifies *only* about three times more MLS operations than the ADS compiler. As a result, we reduce the number of loads/stores by 17%, a little more than a

benchmark code	full optimization			our technique added		
	total	memory	MLS	total	memory	MLS
fullsearch	189	74	9	180	51	23
dpfield_estimate	134	57	3	126	40	12
ford2	238	104	2	230	75	23
pass2_fs_dither	200	117	4	181	83	19
quantize_fs_dither	136	76	3	128	58	13
field_estimate	305	176	13	291	127	48

(a) Code size reduction with the GCC compiler

benchmark code	full optimization			our technique added		
	total	memory	MLS	total	memory	MLS
fullsearch	176	58	5	172	45	14
dpfield_estimate	127	37	8	125	33	10
ford2	145	96	3	138	82	10
pass2_fs_dither	184	107	2	172	83	14
quantize_fs_dither	106	51	2	101	44	5
field_estimate	265	120	20	259	101	33

(b) Code size reduction with the ADS compiler

Table 1 Reduction of instruction counts when our technique is applied to the GCC and ADS compilers, respectively

half of the percentage we achieve with the GCC compiler. Overall, we achieve approximately 4% ratio of the total code size reduction on average.

Not surprisingly, exploiting MLS instructions does not lead to a dramatic decrease in the total code size. However, considering that we successfully further reduce the code size even after every effort was already made by both compilers for code optimizations, we believe these results are of some meaning. Besides, although we did not present empirical evidence in this paper, we also believe that several tenfold percent (about 10 to 30%) reduction in loads and stores would bring about a tangible reduction in running time and energy consumption[3], which are also equally important performance metrics in embedded processors.

5. CONCLUSION & FUTURE STUDY

The work reported here has been motivated by our on-going project to build an optimizing compiler for a commercial media processor under development. In the processor, we found a variety of instructions specifically designed to accelerate media applications, and among them there were MLS instructions. In our efforts to optimize the code with these instructions, we found that no previous compilers had addressed this optimization problem seriously before. For this reason, we opted for pursuing our research to devise a cost-effective algorithm that tackles this exponential-time problem fast and efficiently.

In this paper, we analyze that the MLS problem is an enormously complex problem tangled with several NP-complete subproblems. We, therefore, circumvent this complexity by applying heuristics. That is, we first divide the original problem in three subproblems each defined by a constraint, and then enforce the constraints one-by-one in three different phases. Since each phase is implemented by a polynomial time algorithm, the overall complexity of our algorithm still remains polynomial in the number of input loads/stores.

Our heuristic-based algorithm does not always guarantee an optimal solution to the MLS problem. However, we demonstrated through experiments that it exploits MLS instructions effectively to further reduce the size of code even after the code is fully optimized by existing production-quality compilers. Although our technique cannot reduce the total code size on a dramatic scale, it has been proven to be effective to some extent for most cases after all.

There are still remaining several research topics for our future study. For instance, in Figure 14, we explained the importance of register spill decision on our technique. We are currently developing a new register allocation algorithm that can minimize spill costs by carefully selecting the registers to spill so that MLS operations can be maximally utilized in the memory accesses for spill and reloading.

3) Many studies indicate that energy consumption as well as running time is more dominated by memory operations than ALU operations [19, 20].

REFERENCES

[1] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Storage Assignment to Decrease Code Size. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 186-195, 1995.

[2] A. Rao and S. Pande. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs. In *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation*, pages 128-138, May 1999.

[3] ARM, www.arm.com. ARM *Instruction Set Quick Reference Card.*

[4] ARM, www.arm.com. *ARM Developer Suite - Version* 1.2, Nov. 2001.

[5] D. Bartley. Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes. *Software Practice & Experience*, 22(2), 1992.

[6] R. Leupers and F. David. A Uniform Optimization Technique for Offset Assignment Problems. In *International Symposium on Systems Synthesis*, pages 3-8, 1998.

[7] Yoonseo Choi and Taewhan Kim. Address Assignment Combined with Scheduling in DSP Code Generation. In *Design Automation Conference*, 2002.

[8] X. Zhuang, C. Lau, and S. Pande. Storage Assignment Optimizations through Variable Coalescence for Embedded Processors. In *Proceedings of the SIGPLAN Conference on Languages, Com-*

piler and Tools for Embedded Systems, pages 220-231, June 2003.

[9] D. Ottoni, G. Ottoni, G. Araujo, and R. Leupers. Improving Offset Assignment through Simultaneous Variable Coalescing. In *Workshop on Software and Compilers for Embedded Systems*, Sep. 2003.

[10] V. Nandivada and J. Palsberg. Efficient Spill Code for SDRAM. *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Nov. 2003.

[11] A. Buchsbaum, R. Giancarlo, and J. Westbrook. On Reduction via Determinization of speech Recognition Lattices. Technical report, AT&T Bell Labs, 1997.

[12] V. Zivojinovic, J.M. Velarde, C. Schager, and H. Meyr. DSPStone - A DSP oriented Benchmarking Methodology. In *Proceedings of International Conference on Signal Processing Applications and Technology*, 1994.

[13] C. Lee, M. Potkonjak, and W Mangione-smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330-335, Nov. 1997.

[14] J.Kim, S. Jung, Y. Paek, and G. Uh. Experience with a Retargetable Compiler for a Commercial Network Processor. *In International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Oct. 2002.

[15] Y. Paek, M. Ahn, and S. Lee. Case Studies on Automatic Extraction of Target-specific Architectural Parameters in Complex Code Generation. In *Workshop on Software and Compilers for Embedded Systems*, Sep. 2003.

[16] G. Chaitan. Register Allocation and Spilling via Graph Coloring. In *Proceedings of the SIGPLAN symposium on Compiler Construction*, pages 201-207, June 1982.

[17] R. Stallman. *Using the GNU Compiler Collection.* Free Software Foundations. Dec. 2002.

[18] Embedded Concepts & Solutions, Inc., www.go-ecs.com. *ARM Technical Tidbits*, 2002.

[19] M. Franklin and T. Wolf. Power Considerations in Network Processor Design. In *Network Processor Design - Issues & Practices: Volume II*. Morgan Kaufmann Pub., Sep. 2003.

[20] M. Sanchez-Elez, M. Fernandez, M. Anido, H. Du, N. Bagherzadeh, and R. Hermida. Low Energy Data Management for Different On-Chip Memory Levels in Multi-Context Reconfigurable Architectures. In *Design Automation Conference*, June 2003.

안 민 욱

2003년 서울대학교 전기공학과(학사). 서울대학교 전기컴퓨터공학부 석박사통합과정(현재). 관심분야는 ASIP 설계방법론, 컴파일러 설계, HW/SW 동시설계

조 두 산

2001년 한국외국어대학교 정보공학과(학사). 2003년 고려대학교 전기공학과(석사). 서울대학교 전기컴퓨터공학부 박사과정(현재). 관심분야는 임베디드 시스템 설계, 최적화 컴파일러, HW/SW 동시설계

백 윤 흥

1988년 서울대학교 컴퓨터공학과(학사) 1990년 서울대학교 컴퓨터공학과(석사) 1997년 University of Illinois Urbana Champaign Computer Science(박사) 2002년 KAIST 전자전산학과 부교수. 현재 서울대학교 전기컴퓨터공학부 부교수 관심분야는 고속 ASIP, 프로토타이핑을 위한 EDA툴, 재겨냥성 최적화 컴파일러, 임베디드 프로세서 설계

조 정 훈

1996년 KAIST 전자전산학과(학사). 1998년 KAIST 전자전산학과(석사). 2003년 KAIST 전자전산학과(박사). 현재 경북대학교 전자전기컴퓨터공학부 전임강사 관심분야는 임베디드 시스템, 소프트웨어 최적화, EDA툴, 임베디드 시스템을 위한 재겨냥성 최적화 C 컴파일러