Efficient Spill Code for SDRAM

V Krishna Nandivada
Purdue University
Dept. of Computer Science
West Lafayette, IN 47907
nvk@cs.purdue.edu

Jens Palsberg
Purdue University
Dept. of Computer Science
West Lafayette, IN 47907
palsberg@cs.purdue.edu

ABSTRACT

Processors such as StrongARM and memory such as SDRAM enable efficient execution of multiple loads and stores in a single instruction. This is particularly useful in connection with register allocation where spill code may need to save and restore multiple registers. Until now, there has been no effective strategy for utilizing this to its full potential. In this paper we investigate the use of SDRAM for optimization of spill code. The core of the problem is to arrange the variables in the spill area such that loading to and storing from the SDRAM is optimally efficient. We show that the problem is NP-complete and present a method based on integer linear programming (ILP) to solve the problem. We have implemented our approach as an additional phase in a gcc-based compiler for the StrongARM core of Intel's IXP–1200 network processor. Our optimizer, SLA (stack location allocator), rearranges the scalar variables so that memory accesses can be made cheaper. Our experimental results show that our ILP-based method is efficient and that the code generated for our benchmarks runs 0.8–15.1% faster than the code produced by the original compiler with –O2 optimization. Our SLA phase is guaranteed to not deteriorate the execution-time performance and can be configured such as not to increase the code size.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation*

General Terms

Algorithms, Measurement, Performance, Experimentation

Keywords

Memory layout, integer linear programming, optimization, SDRAM

CASES'03, Oct. 30–Nov. 2, 2003, San Jose, California, USA.

1. INTRODUCTION

1.1 Background

The widening gap between processor speed and memory speed motivates better compiler techniques for reducing the number of memory accesses. The idea is that even a small reduction in memory accesses can give a significant improvement in execution time. One opportunity is given by a commonly-used memory technology, namely memory with a 64-bit bus, including RAMBUS QRSL and SDRAM. An increasing number of processors exploit this memory technology by allowing loading and storing of multiple registers in one instruction. Processors in this category include the Sun MAJC 5200 [17] and several network processors, such as the IBM PowerPC405 in the PowerNP NP4GS3 [19] and the Intel StrongARM in the IXP–1200 [2]. Until now, little has been published work on how a compiler can take advantage of the capabilities for multiple load and store. In this paper we present a compiler technique for maximizing the number of multiple load/store instructions, in the context of the Intel StrongARM in the IXP–1200 network processor.

The Intel IXP–1200 contains a StrongARM processor and a SDRAM unit along with many other units. On the StrongARM, the register size is 32 bits and the basic load/store operations (called LDR and STR) operate on one register at a time. However, the SDRAM has a 64 bit bus, so if we are using a LDR instruction to load a 32 bit register, we are wasting half of the bandwidth of the bus. Fortunately, the StrongARM also allows efficient execution of multiple loads and stores to and from the SDRAM in a single instruction (we refer to them as LDM and STM) [24]. If we use a LDM/STM instruction with two registers, then we save one full load/store instruction, which is equivalent to saving around 40/50 cycles [26]. The formats of the LDM/STM instructions are:

LDM baseRegister, bitVector
STM baseRegister, bitVector

where baseRegister holds a memory address, called the *base address*, which we write as [baseRegister], and bitVector denotes a subset (possibly all) of the general-purpose registers. In the first instruction, LDM stands for "load-multiple," and the idea is to load several words, starting from [baseRegister], into the registers denoted by bitVector. In the second instruction, STM stands for "store-multiple," and the idea is to store the registers denoted by bitVector into the memory, starting from [baseRegister]. A load-multiple instruction loads the lowest-numbered register from [baseRegister], the second-lowest register from [baseRegister] + 4, and so

on. A store-multiple instruction works similarly. For example, let us consider loading four items. Loading them individually, with four LDR instructions, will take $4 \times (1 + 40) = 164$ cycles (one cycle of processor time and 40 cycles for the memory access, for each load). Loading them all together, with one LDM instruction, will take $1 + (2 \times 40) = 81$ cycles (the memory access is done in two steps because the total number of words accessed is twice the bus width). There is a middle ground here: we can load the four items in pairs, with two LDM instructions, which will take $2 \times (1 + 40) = 82$ cycles. As the example indicates, most of the benefit of multiple loads and stores can be achieved using double loads and double stores. Hence, in this paper we will concentrate on double loads and double stores only. Investigation of triple loads and stores, and beyond, is left to future work.

Suppose we want to replace the following two LDR instructions with one LDM.

LDR $addr_1$ r_i
LDR $addr_2$ r_j

Let us assume $i \neq j$. The two base addresses $addr_1$ and $addr_2$ must be contiguous at 4 byte boundaries, that is, $addr_2 - addr_1 = 4$. (If $addr_1 - addr_2 = 4$, then swap the two instructions.) There are two cases depending on whether $i < j$ or $i > j$. If $i < j$, then we replace the two LDRs by the following code, in which r is a free register:

MOV r $addr_1$
LDM [r] {r_i,r_j}

In the binary format of the LDM instruction, $\{r_i, r_j\}$ is represented by a bit map of 16 bits with one bit for each of r_1 through r_{16}. Thus, $\{r_i, r_j\}$ and $\{r_j, r_i\}$ denote the same bit map.

If $i > j$, then we have an *inversion*: if we replace the two loads with LDM like above, then the value from $[addr_1]$ would be loaded into r_j and the value from $[addr_2]$ would be loaded into r_i. We handle inversions by swapping the contents of r_i and r_j, using the standard trick involving three exclusive-or instructions (called eor on the StrongARM):

eor r_i, r_i, r_j
eor r_j, r_i, r_j
eor r_i, r_i, r_j.

The advantage of using exclusive-or instructions in this fashion is that no extra temporary register is needed. Note that the exclusive-or instructions operate on registers only, and hence are much faster than load and store instructions. Thus, even with three extra exclusive-or instructions, the resulting code is faster than two single loads.

The case of replacing two store instructions with a store-multiple instruction is similar to that of load, except for two differences in the case of an inversion. First, the swapping of registers is done *before* the MOV instruction. Second, if both registers are live after the stores, then, additionally, we need to swap the contents of the registers *after* the store-multiple instruction, resulting in a total of six exclusive-or instructions.

In this paper we investigate how a compiler can maximize the number of double loads and double stores, while minimizing the number of inversions. While this problem can be tackled at various stages of a compiler, we focus on the late stages that follow register allocation and spilling, in the context of the gcc compiler for the StrongARM. We do that because, once spill code is inserted, the locations of all the loads and stores in the code are known. The register allocator will assign some variables to registers and other variables to stack locations. On the IXP–1200, the gcc compiler represents those stack locations on the SDRAM. When a variable on the stack needs to be loaded and stored, the gcc compiler first generates only individual load and store instructions. Only during the peephole optimization phase, the compiler attempts to combine the load/store instructions. But the peephole optimizer does these replacements only if the two loads/stores are accessing consecutive locations and the registers being loaded/stored are in the same order (ascending or descending) as that of the memory addresses that are being accessed. The compiler does not attempt to rearrange the variables, and in case of inversion, no load-multiple or store-multiple instructions are introduced. We can do better.

To generate double-load and double-store instructions from individual-load and individual-store instructions, we need to (1) be able to move them next to each other and (2) have them access consecutive memory addresses. The first of these tasks is a standard code motion problem which must be done without changing the program behavior. The second task is a memory layout problem. To solve it, we can change the stack layout for the local variables of each procedure. It is now of paramount importance to note how much we can change the stack layout without changing the behavior of the code. When compiling C programs, we make changes to two parts of the stack layout, namely (1) the stack area for scalar variables (int, float, double, char, enum) and (2) the stack area for arguments that are passed in registers and are saved by the callee (at most 4 in case of the StrongARM). The reason for this choice is that only in these two cases, the C language standard does not specify the stack layout [5, Section 6.9.1#9], while it prohibits rearrangement of the fields inside aggregate types [5, Section 6.7.2.1]. The good news is that the gcc compiler stores all of the variables listed in (1) and (2) in one contiguous memory area, irrespective of the order in which they are declared. We will refer to this memory area as the *scalar* memory area, and we will use *placement function* to refer to any permutation of the locations in the scalar memory area. Intuitively, a placement function produces a new stack layout by rearranging the variables in the scalar memory area. We now reformulate our problem into our core challenge, which we state both as an optimization problem and as a decision problem for blocks (i.e., sets) of memory accesses.

> **The Placement Problem:** Given a set of blocks of memory accesses, find a placement function that leads to maximizing the number of double loads and double stores, while minimizing the number of inversions.

> **The Placement Decision Problem:** Given a set of blocks of memory accesses and natural numbers q, r, does there exists a placement function that leads to at least q double loads and double stores, and at most r inversions.

1.2 Our Results

First we characterize the complexity.

Theorem The placement decision problem is NP-complete.

PROOF. It is straightforward to show that the placement decision problem is in NP: a placement function is polynomial in the size of the scalar memory area, and we can check in polynomial time whether a given placement function leads to at least q double loads and double stores, and at most r inversions.

To show NP-hardness, we do a reduction from the hamiltonian path problem. Suppose we are given a graph (V, E) of vertices V and undirected edges E. We can assume, without loss of generality, that for every edge (v_1, v_2), we have $v_1 \neq v_2$. From the graph, we construct a program where each vertex becomes a stack location and each edge (v_1, v_2) becomes a basic block consisting of two consecutive instructions which access exactly two different stack locations, corresponding to v_1, v_2, and operate on two different registers. We now claim that the graph has a hamiltonian path if and only if the program has a placement function which enables (at least) $|V| - 1$ double loads/stores and any number of inversions. To see that, notice that a hamiltonian path and a placement function essentially are the same thing: they both order the stack locations. Since the hamiltonian path problem is NP-complete [8], the placement decision problem is NP-hard. □

Given that the placement decision problem is NP-complete, we are discouraged from trying to find a polynomial-time algorithm for the placement problem. Instead, we can either try to find approximate solutions or we can use exact methods that run in exponential time.

In this paper we present an exact method for solving the placement problem. Our approach is based on integer linear programming (ILP) and is therefore an NP algorithm. We have implemented our method in the gcc 2.95.2 compiler for the StrongARM and we have experimented with it in the context of the IXP–1200. Our approach is implemented in gcc as an additional phase, named SLA (stack location allocator), that follows immediately after register allocation and spilling.

Our experimental results show that our ILP-based method is effective and that the code generated for our benchmarks runs 0.8–15.1% faster. Considering the fact that we get this improvement over an already optimized code (compiled with –O2 option of gcc), this is significant. Most importantly, our SLA phase of optimization is guaranteed not to deteriorate the execution time. The StrongARM supports the efficient implementation of double loads and stores that we consider in this paper. For machines that break down the double loads and stores into individual bus transactions for each register, there would not be any gains in speed.

1.3 Example

Consider the C code in Figure 1. The example illustrates that if we are accessing scalars that are more than 4 bytes apart, then the code generated by the standard gcc compiler is poor. Notice the calls to the functions bar1, bar2, and bar3. Each one takes one or more addresses of variables as arguments; the calls were inserted to ensure that the compiler reloads the variables after the function call, because the callee might modify the variables.

The table in Figure 1 shows the locations of the variables

```
foo(){ int a,b,c,d,e;
       bar1(&a,&b,&c);
       a=c+a;
       bar2(&b,&d);
       e=b+d;
       bar3(&e);
       return a+e; }
```

var	old loc	new loc
a	fp-20	fp-24
b	fp-24	fp-32
c	fp-28	fp-20
d	fp-32	fp-36
e	fp-36	fp-28

	Without SLA	With SLA	
1	sub r0, fp, #20	sub r0, fp, #24	;&*a*
2	sub r4, fp, #24	sub r4, fp, #32	;&*b*
3	mov r1, r4	mov r1, r4	
4	sub r2, fp, #28	sub r2, fp, #20	;&*c*
5	bl bar1	bl bar1	;*call*
6			
7	ldr r3, [fp, #-28]	sub r2, fp, #24	
8	ldr r2, [fp, #-20]	ldmia r2, {r2-r3}	;**load a,c**
9	add r3, r3, r2	add r3, r3, r2	;*a=a+c*
10	str r3, [fp, #-20]	str r3, [fp, #-24]	;*store a.*
11	sub r0, r4	mov r0, r4	
12	sub r1, fp, #32	sub r1, fp, #36	;&*d*
13	bl bar2	bl bar2	;*call*
14			
15	ldr r3, [fp, #-24]		
16	ldr r2, [fp, #-32]	ldmia sp, {r2-r3}	;**load b,d**
17	add r3, r3, r2	add r3, r3, r2	;*e=b+d*
18	str r3, [fp, #-36]	str r3, [fp, #-28]	;*store e*
19	sub r0, fp, #36	sub r0, fp, #28	;&*e*
20	bl bar3	bl bar3	;*call*
21			
22	ldr r3, [fp, #-20]	sub r0, fp, #28	
23	ldr r0, [fp, #-36]	ldmia r0, {r0,r3}	;**load a,e**
24	add r0, r3, r0	add r0, r3, r0	;*a+e*

Figure 1: Code without and with SLA

before and after the SLA pass. Notice that a permutation has been done and that after the SLA phase, the variables that are accessed together (a and c, b and d, a and e) have consecutive addresses. In Figure 1 we show the code generated by the standard gcc compiler and the code generated by the gcc compiler with SLA for the example C program. We have omitted the code that stores and restores the callee-save registers and sets the stack frame. For the code shown we have sp=fp–36.

The function bar1 expects the addresses of the variables a, b and c to be passed in the registers r0, r1 and r2, respectively. Because the SLA phase modifies the locations of the variables, we can see different locations for these variables in the SLA and non-SLA version of the code (lines 1, 2, 4), in accordance with the new mapping shown in the table in Figure 1. We can see similar changes in lines 10, 12, 18, 19.

In lines 7–8 the two loads in the non-SLA version of the code cannot be merged as the memory accesses are not consecutive. In the SLA version of the code, we first set r2 with the address of the *a* and the two loads are replaced by one LDM instruction (*ldmia*). We can see similar changes in lines 22-23. Finally let us look at lines 15-16. In the SLA version of the code, we need not insert any 'add/sub' instruction as the location first accessed is already there in register sp.

1.4 Related Work

Various compiler-based techniques have been proposed to reduce memory latencies, including compiler-directed prefetching [6] and value prediction [16, 29]. In this paper we propose a compiler-based technique that uses the StrongARM

processor's load-multiple/store multiple instructions, in the presence of SDRAM memory, by rearranging local variables to reduce the overhead of memory accesses.

The storage assignment problem was first studied by Bartley [4], and later by many authors [14, 22, 28, 21, 25, 13, 12, 20] in the last decade, for many different types of special purpose architectures. Traditionally the problem has been studied from two angles. (i) One range of benefits from good storage assignment include allocating variables in registers, reducing the cost of inter-bank copy, and reducing the code size [21]. Sjödin and Platen [25] present a model for storage allocation to describe architectures with memories of varying speed and with several native pointer types. The goal here is to ensure heavily accessed variables are placed in faster memory and are accessed with cheap pointers (registers). They frame the problem as an ILP formulation and use the solution to allocate non-local-scoped variables. (ii) Another range of benefits from good storage assignment come from good allocation of variables in memory after register allocation has been done. Most of the work here is done by keeping in mind the presence of auto increment/auto decrement indirect addressing modes in DSP processors. The goal here is to reduce the number of explicit instructions to load the address of variables into the address registers. For example, Liao et al [14] present an approach for optimal storage assignment such that explicit instructions for address arithmetic are minimized, and, hence resulting in compact code. Leupers and Marwedel [13] present approximate algorithms to optimize the utilization of a proposed parallel Address Generation Units(AGUs) by computing appropriate memory layouts for scalar variables. Leupers and David [12] present a genetic algorithm based approach to generate new offsets to handle different register file sizes and auto increment ranges. Rao and Pande [22] give techniques based on approximation algorithms to optimize the access sequence of variables by applying algebraic transformations on expression trees. Sudarsanam et al [28], present a formulation of the storage assignment problem, parameterized by the maximum allowable increment limit and the number of address registers. Panda et al [20] present a heuristic based approach for storage assignment to data improve cache locality. Our work goes for yet another advantage from good storage allocation: better utilization of a 64-bit bus.

Recently, network processors have received considerable attention. Most software for network processors is written directly in machine code because most current compilers cannot achieve wire-speed performance of the generated code. For complicated applications such as routers with firewalls and sniffing capabilities, even carefully crafted machine code can have trouble keeping up with wire speed. These observations motivate better optimizing compilers for network processors, and make almost any performance gain important. Our results are a step in this direction. There are a few other recent projects on compiling for network processors and exploiting many of their special features. Wagner and Leupers [30] talk about register allocation for processors that support bit section referencing. George and Blume [9] propose a new language and address issues with register banks and aggregate registers.

There has been widespread interest in using ILP for compiler optimizations such as instruction scheduling, software pipelining, and particularly register allocation [3, 10, 15, 23, 27]. Our work shows that solving ILPs is sufficiently fast for our per-procedure placement problem.

We have three main contributions. (a) We observe that the amount of time taken to execute three exclusive-or instructions is far less than the time to do two memory accesses. Hence we introduce the concept of inversions to help improve performance. However note that the number of double loads/double stores and the number of inversions are not linearly correlated. We handle this by prioritizing two requirements in our objective function. (b) Liao et al [14] assume a fixed evaluation order for each basic block. Rao and Pande [22] relaxed this assumption by allowing code motion inside algebraic expression trees. We further relax this by allowing code motion to take place anywhere inside the basic block, as long as the dataflow is not affected. This gives us more opportunities to get consecutive memory accesses next to each other. (c) We present an exact method using ILP to solve the storage assignment problem along with inversions.

2. THE SLA ALGORITHM

The SLA algorithm has four phases: model extraction, constraint generation, constraint solving, and code transformation.

2.1 Model Extraction

We first extract a model from the program. We use the following notation for arrays, for example, array $\{1..n\}$ of $\{0,1\}$, which denotes an array of size n with elements from $\{0,1\}$. The model has eight components:

• vars = $\{1..n\}$. This is the set of the n scalar variables present in the function currently being compiled.

• blocks = $\{1..k\}$. A block is an unordered collection of loads/stores that can be moved together. Each element of blocks identifies one of the k blocks.

• triples = blocks × vars × vars.

• edge: array {triples} of {0,1}. If edge$[b, v_1, v_2]$, then the memory accesses of v_1, v_2 are candidates to be replaced by one load/store multiple instruction in block b. For simplicity, we insist that if edge$[b, v_1, v_2]$=1, then $v_1 < v_2$. We can guarantee that $v_1 \neq v_2$, because if we have two consecutive accesses to the same location, then one of them can be removed (in case of store), or replaced by a mov instruction (in case of load).

• inv: array {triples} of {-1,1}. For an edge $e = (b, v_1, v_2)$, suppose v_1, v_2 are loaded/stored to/from the registers r_i, r_j. If $i < j$, then inv$[e] = 1$, else inv$[e] = -1$. The idea is that if inv$[e] = 1$, then we would prefer that the new address of v_1 be smaller than the new address of v_2. Similarly, if inv$[e] = -1$ then we would prefer that the new address of v_1 be greater than the new address of v_2. We use inv to help minimize the number of inversions. We can guarantee that $i \neq j$, because if the destinations of two loads are the same register, then we eliminate the first instruction. Similarly, if the sources of two stores are the same register, then if we have a free register available then we use that register here else we do not add an edge for those two instructions.

• cost: array {triples} of {40,50}. Gives the cycle count for each element of the edge, where cost[e] = 40 if the edge consists of two load instructions, and cost[e] = 50 if the edge consists of two store instructions.

• eorCost: array {triples} of {3,6}. Gives the number of eor instructions that have to be inserted if an inversion occurs for the edge. In case of a load it is 3, and in case of a store it is either 6 or 3, depending on whether the source

registers are live after the store or not.

• w : array {triples} of {1,50}. Gives an estimate of the execution frequency of each edge. The weight of an edge is statically assigned as 50 if the instruction corresponding to the edge is in a loop and 1 otherwise.

The quality of the model generated, and consequently the resulting code, depends on precise nature and power of code motion algorithm we use. We use a simple code motion algorithm which tries to move memory accessing instructions, present within one basic block, next to each other, based on memory dependencies and anti/output dependencies. We get the register liveness details, dependence constraints, and basic block information from the underlying gcc compiler's optimization framework.

2.2 Constraint Generation

Once we have done the model extraction, we generate an ILP from the program model. A solution to the ILP expresses a placement for the local variables.

Variables. The ILP uses the following variables:

• f : array {vars × vars} of {0,1}. In the solution to the ILP, f represents the desired placement function as a permutation matrix: if $f[v,p] = 1$, then the variable v has the position p.

• diff : array {triples} of integer. In the solution to the ILP, for an edge $e = (b, v_1, v_2)$, if $f[v_1, p_1] = 1$ and $f[v_2, p_2] = 1$, then $\text{diff}[e] = p_2 - p_1$.

• isPair : array {triples} of {0,1}. If $\text{isPair}[e] = 1$, then we can introduce a load/store multiple instruction for e.

Objective function. Solving the placement problem contributes to saving cycles in the overall execution. So, our ILP maximizes an objective function which approximates the number of saved cycles:

$$\sum_{e \in \text{triples}} \text{edge}[e] \times w[e] \times s[e]$$

We use $s[e]$ to denote the number of cycles that are saved in one execution of edge e. Note that savings only happen when $\text{isPair}[e] = 1$. The precise formula for $s[e]$ is:

$$\begin{aligned} s[e] \;=\; & \text{isPair}[e] \times \text{cost}[e] \;+ \\ & \text{isPair}[e] \times \tfrac{1}{2} \times (\text{diff}[e] \times \text{inv}[e] - 1) \times \text{eorCost}[e]. \end{aligned}$$

The first part, $\text{isPair}[e] \times \text{cost}[e]$, expresses that we save the cost of one load or one store. The second part says that if $\text{diff}[e] \times \text{inv}[e] = -1$, that is, if we have an inversion, then the second part is negative, and we pay $\text{eorCost}[e]$.

Notice that the second part of $s[e]$ contains a product of $\text{isPair}[e]$ and $\text{diff}[e]$, which makes the objective function nonlinear. The nonlinearity takes us outside the realm of ILP, so instead we use the term $\text{diff}[e] \times \text{inv}[e] \times \text{eorCost}[e]$. The term $\text{diff}[e] \times \text{inv}[e]$ is in the interval $[-(n-1), n-1]$ and so it can overwhelm the first part of $s[e]$. To compensate, we multiply the first term of $s[e]$ by n, arriving at this definition:

$$s[e] \;=\; n \times \text{isPair}[e] \times \text{cost}[e] \;+\; \text{diff}[e] \times \text{inv}[e] \times \text{eorCost}[e].$$

We have not seen any deterioration in the quality of the final solution due to this approximation for many hand-coded examples.

Constraints. We generate the following integer linear constraints. The following constraints ensure that f is a permutation matrix.

$$\forall v \in \text{vars} : \sum_{p \in \text{vars}} f[v,p] = 1 \;; \qquad \forall p \in \text{vars} : \sum_{v \in \text{vars}} f[v,p] = 1$$

The following constraint expresses that we can introduce a load/store multiple instruction only for edges that are present in the program model.

$$\forall e \in \text{triples} : \qquad \text{isPair}[e] \le \text{edge}[e].$$

A vertex can appear only once in a Pair per block. For each block, the following constraints say that if an edge is counted as a Pair, then we cannot have any other edge with common vertices in the same block. Note that these constraints are per block and do not restrict pairs in different blocks from having common variables.

$$\begin{aligned} & \forall b \in \text{blocks} : \forall v_1, v_2, v \in \text{vars } (v_1, v_2, v \text{ are different}) : \\ & \text{isPair}[(b, v_1, v_2)] + \text{isPair}[(b, v_1, v)] \le 1 \\ & \text{isPair}[(b, v_1, v_2)] + \text{isPair}[(b, v_2, v)] \le 1 \\ & \text{isPair}[(b, v_1, v_2)] + \text{isPair}[(b, v, v_1)] \le 1 \\ & \text{isPair}[(b, v_1, v_2)] + \text{isPair}[(b, v, v_2)] \le 1 \end{aligned}$$

The idea of the following constraint is that for all triples e, if $\text{isPair}[e] = 1$, then $|\text{diff}[e]| = 1$.

$$\begin{aligned} \forall e \in \text{triples} : \qquad & n \times \text{isPair}[e] \le (n+1) - \text{diff}[e] \\ & n \times \text{isPair}[e] \le (n+1) + \text{diff}[e] \end{aligned}$$

The following constraint computes diff.

$\forall e = (b, v_1, v_2) \in \text{triples}$:

$$\text{diff}[e] \;=\; \left(\sum_{p_2 \in \text{vars}} f[v_2, p_2] \times p_2 \right) - \left(\sum_{p_1 \in \text{vars}} f[v_1, p_1] \times p_1 \right)$$

2.3 Constraint Solving

We use AMPL [7] to generate the ILP, and CPLEX [1] to solve it. The gcc compiler invokes the constraint generator by providing the data in a file. Once constraints are generated the constraint generator calls the solver, which returns the resulting solution to gcc in a file.

2.4 Code Transformation

Finally we use the ILP solution to replace all stack offsets and introduce double loads and stores in the code.

Offsets. For each instruction: (1) if it is an add or sub instruction with the stack pointer as the source and an integer constant as the second operand (offset) then if the offset is in the scalar memory area, we replace it by the new offset computed; and (2) if it is a load/store instruction with the stack pointer as the base register then if the offset is in the scalar memory area, we replace it by the new offset computed.

Double loads/stores. For each edge e where $\text{isPair}[e] = 1$, we replace the corresponding two instructions with a load/store multiple instruction. If the lowest address of the loads/stores being merged is *not* already present in a register, then we insert an *add* instruction to set up the base register. In the case of inserting an *add* instruction before a double store, we need a free register; if we don't find one, then we do not do the replacement. In the case of inserting an *add* instruction before a double load, we can use one of the target registers for the two load instructions.

Benchmark characteristics			Compile time (sec)			Transformations			Execution time (sec)		
name	funcs	lines	w/o SLA	SLA	% worse	loads	stores	eor	w/o SLA	SLA	% imp
GSM	98	8643	5.22	5.90	13.5	18	8	6	0.57	0.55	3.6
EPIC	49	3540	1.34	2.67	99.2	228	30	24	0.65	0.61	6.2
Url	12	790	0.25	0.52	108.0	12	4	0	6.32	6.27	0.8
Md5	17	753	0.27	0.30	14.8	4	0	0	0.75	0.73	2.7
IPChains	76	3453	1.69	2.67	58.0	44	14	9	0.23	0.20	15.1
Classifier	25	2850	2.27	4.73	107.0	26	2	6	2.71	2.70	0.8
Firewall	30	2281	1.84	2.71	47.3	24	0	6	3.49	3.41	2.4

Figure 2: Experimental Results

3. EXPERIMENTAL RESULTS

We have implemented SLA as a phase of optimization in the gcc 2.95.2 compiler for the StrongARM processor. In addition, our implementation has a peephole optimization phase that tries to combine loads/stores of *non-scalar* variables without rearranging them, in a way that goes beyond the peephole phase of gcc.

3.1 Benchmark Characteristics

We have evaluated our implementation using a total of seven benchmark programs, drawn from three different suites. Some statistical details about them can be found in Figure 2, columns 2–3.

Our first two benchmarks are from the MediaBench [11] suite: *GSM* is an implementation of the European GSM 06.10 provisional standard for full-rate speech transcoding. *EPIC* is an experimental image compression utility.

The second set is a collection of three benchmarks drawn from the NetBench [18] suite: *Url* implements a context switching mechanism called URL-based switching. *Md5* is a message digest algorithm that creates a cryptographically secure signature for each outgoing packet. *IPChains* is a firewall application. NetBench has three classes benchmarks: small, medium and large. The subset we present here includes a benchmark from each class. Note that we have presented all the benchmarks from the NetBench and MediaBench suites that we could run on our setup. The rest of the benchmarks programs could not be run even in the absence of our optimization.

The third set of two benchmarks were written specifically for the Intel IXP–1200 processor: *Classifier* is a network packet classifier that classifies ARP and TCP packets. *Firewall* implements a firewall. For each packet received, it either drops it or stores it depending on the rules set and the contents of the packet. Developed by graduate students for the Network Processors course offered by Douglas Comer at Purdue University, these programs have two parts. The first part runs on the StrongARM processors, and the second one runs on microengines. To be able to time the StrongARM code alone, we added code to simulate microengine code. The microengine simulation code supplies network packets that we collected offline. The source code of these benchmarks can be obtained from the authors.

3.2 Measurements

We measured the compile time deterioration and the execution time improvement. Figure 2, columns 4–9, show the compile time statistics measured on a Pentium i686 machine running Linux. For each benchmark it gives the time taken to compile at –O2 level of optimization without and with SLA, percentage deterioration in compile time, number of loads/stores replaced with double loads/stores, and number of eor instructions added. Note that the current peephole-optimization phase of gcc is switched on by default, at the O2 level of optimization.

Columns 10–11 in Figure 2 show the execution time statistics in terms of the time taken to execute the benchmark program compiled without and with SLA. The execution time reported is the execution time as an average over four runs on the Intel IXP Network processors. The last column shows the percentage improvement in execution time. The figures we show here are from the runs of our programs in the presence of data cache. We believe that in the absence of data cache the comparative gains would be bigger.

3.3 Assessment

For our benchmarks, the execution time improvements are in the range 0.8–15.1%. The geometric average improvement is 2.8% and the arithmetic average improvement is 4.5%. The improvement is over a powerful baseline, namely code generated at –O2 level of optimization. Hence, we believe the improvement is significant.

The compile time overhead is up to a factor of about two. For our benchmarks, this amounts to at most one or two extra seconds. We believe that the compile time overhead is affordable for a significant improvement in execution time. Such optimizations can be run for the production builds and omitted for regular debug builds. Notice that Url and Classifier have the highest overhead in compile time and the lowest performance improvement. The long compilation times are due to a large number of edges in the generated ILPs. However, for both benchmarks, most of the replacements are done in infrequently executed code, giving a small performance improvement of 0.8%. So, longer compilation times do not necessarily entail larger performance gains.

For some benchmarks, there were few replacements and, yet, there were significant improvements in execution time. For example, for Md5, even though only four instructions got replaced, the replacements took place in a frequently executed function giving a significant performance improvement of 2.7%. So, large performance gains do not always require many replacements.

For all benchmarks, the number of replaced load instructions is significantly higher than the number of replaced store instructions. This was expected because there are fewer store edges than load edges for each benchmark (we omit the detailed counts).

The number of inserted exclusive-or (eor) instructions is moderate and in two cases even zero. However, the two

benchmarks for which the most eor instructions were inserted also saw the largest performance gains. This suggests that the ability to handle inversions is important. However, if code size is a constraint, then we can maximize the number of double loads and double stores only, and ignore inversions. This would mean that no exclusive-or instructions would be inserted, and hence the code generated would always be at most the size of the code generated without SLA.

The first two applications, the largest of our benchmarks, show significant improvements. They were run against the standard data files that come with the benchmarks.

The last two sets of benchmarks are network applications. They typically have an initialization code, followed by a main loop where each incoming packet is processed. The initialization part is run only once, while the main loop is executed many times. In our measurements, we ensure that gains made in the initialization part of the code are mostly amortized away and that gains in the main loop are reflected well. In Table 2, we report the time taken to process a fairly high number of packets, namely 5000.

In addition to the numbers reported, we also ran the last two sets of benchmarks against varying number of input packets. Leaving aside IPChains, in all these benchmarks, we found that the SLA phase modifies the initialization part of the code significantly. Due to this we observed high gains when the number of packets are in the order of few hundreds. However, when we observed it for longer duration, these high peaks got amortized away and we noticed more stabilized gains, due to the replacements done in the main loops. In case of IPChains, we did not find many replacements in the initialization part of the code, and accordingly we have observed low gains for very low packet counts. Most of the replacements in this application were in the main loop. Due to this, for higher packet counts (around 1000), we got around 15% improvement and the gain maintained itself around that level for higher packet counts.

4. CONCLUSION

We have implemented the SLA phase in the gcc compiler for the StrongARM. The code generated with SLA will always run faster; for our benchmarks the improvements are in the range 0.8–15.1%. As the gap between processor speed and memory latency continues to widen, optimizations such as SLA will be increasingly important.

We have given an ILP formulation independent of the compiler's target processor. We believe our ILP formulation can be used with little modification as a placement function for many other targets, such as the Microengines in the IXP's, the IBM NP4GS3, or the Sun MAJC 5200.

Our optimization will be beneficial even for processors with 64-bit registers and a 64-bit memory bus. This is due to the fact that even in the presence of 64 bit registers, it would, mosly likely, still be possible to access 32 bit registers which would hold the smaller data units.

Future Work. The current register allocator present in gcc works independently of our stack location allocator. We plan to integrate register allocation and stack location allocation. The idea is that such a combined phase can do better than the current set up. Manual inspection of the code seems to support our conjecture.

The current SLA does the work only for local variables. We believe that it can be extended to global variables.

Another idea is to extract a more precise program model using an interprocedural analysis, rather than the intraprocedural analysis that we currently use. The weight of each edge is currently calculated based on, rather rough, static execution counts. Our approach might be more efficient if we instead profile the program and use the dynamic execution counts.

Another idea for future work is to use heuristics similar to [4, 14] to find out whether similar performance gains can be obtained with approximate methods that possibly are faster.

Acknowledgments

We thank Dennis Brylow, Christian Grothoff, Vidyut Samanta, Ben Titzer, Thomas VanDrunen, and the anonymous referees for helpful comments on a draft of the paper. We thank Christopher Telfer and Shireen Javali for their help throughout the project. We thank Douglas Comer for his suggestions and for providing us with access to IXP–1200 network processors. We were supported by Intel and by a National Science Foundation ITR Award number 0112628.

5. REFERENCES

[1] CPLEX mixed integer optimizer. http://www.ilog.com/products/cplex/product/mip.cfm.

[2] Intel (r) IXP1200 network processor. http://www.intel.com/design/network/products/npfamily/ixp1200.htm.

[3] Andrew Appel and Lal George. Optimal spilling for CISC machines with few registers. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of PLDI'01, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–253, 2001.

[4] D. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software – Practice & Experience*, 22(2):101–110, Feb 1992.

[5] C9x standard. 1999. http://www.dkuug.dk/JTC1/SC22/open/n2620.

[6] Ben-Chung Cheng, Daniel A. Connors, and Wen Mei W. Hwu. Compiler-directed early load-address generation. In *MICRO*, pages 138–147, 1996.

[7] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL A modeling language for mathematical programming*. Scientific Press, South San Francisco, CA, 1993.

[8] M. R. Garey, R. L. Graham, D. S. Johnson, and D. E. Knuth. Complexity results for bandwidth minimization. *SIAM Journal on Applied Mathematics*, 34(3):477–495, May 1978.

[9] Lal George and Matthias Blume. Taming the IXP network processor. In *Proceedings of PLDI'03, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[10] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software – Practice & Experience*, 26(8):929–965, August 1996.

[11] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[12] Rainer Leupers and Fabian David. A uniform optimization technique for offset assignment problem. In *ISSS*, 1998.

[13] Rainer Leupers and Peter Marwedel. Algorithms for address assignment in DSP code generation. In *Proceedings of IEEE International Conference on Computer Aided Design*, 1996.

[14] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven Tjiang, and Albert Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, May 1996.

[15] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. Evaluation of algorithms for local register allocation. In *Proceedings of Compiler Construction*. Springer-Verlag (*LNCS* 1575).

[16] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture*, pages 226–237, 1996.

[17] http://www.sun.com/processors/MAJC/.

[18] G. Memik, B.Mangione-Smith, and W.Hu. Netbench: A benchmarking suite for network processors. *IEEE International Conference Computer-Aided Design*, Nov 2001.

[19] IBM PowerNP, http://www-3.ibm.com/chips/techlib/techlib.nsf/products/PowerNP_NP4GS3.

[20] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM Transactions on Design Automation of Electronic Systems.*, 2(4):384–409, 1997.

[21] Jinpyo Park, Je-Hyung Lee, and Soo-Mook Moon. Register allocation for banked register file. In Cindy Norris and James B. Fenwick Jr., editors, *Proceedings of LCTES'01, Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 39–47. ACM Press, 2001.

[22] Amit Rao and Santosh Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 128–138, 1999.

[23] John C. Ruttenberg, Guang R. Gao, Woody Lichtenstein, and Artour Stoutchinin. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 1–11, 1996.

[24] David Seal. *Arm Architecture Reference Manual*. ISBN 0 201 73791.

[25] J. Sjödin and C. von Platen. Storage allocation for embedded processors. In *Proceedings of CASES*, pages 15–23, 2001.

[26] Tammo Spalink, Scott Karlin, and Larry Peterson. Evaluating network processors in IP forwarding. Technical Report TR–626–00, Princeton University, November 2000.

[27] A. Stoutchinin. An integer linear programming model of software pipelining for the MIPS R8000 processor. In *PaCT'97, Parallel Computing Technologies, 4th International Conference*, pages 121–135. Springer-Verlag (*LNCS* 1277), 1997.

[28] Ashok Sudarsanam, Stan Liao, and Srinivas Devadas. Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. In *Design Automation Conference*, pages 287–292, 1997.

[29] Gary S. Tyson and Todd M. Austin. Improving the accuracy and performance of memory communication through renaming. In *International Symposium on Microarchitecture*, pages 218–227, 1997.

[30] Jens Wagner and Rainer Leupers. C compiler design for an industrial network processor. In *Proceedings of LCTES'01, Languages, Compilers, and Tools for Embedded Systems, joint with OM'01*, pages 155–164, 2001.