University of California
Los Angeles

Combining Stack Location Allocation with Register Allocation

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Venkata Krishna Nandivada

2005

The dissertation of Venkata Krishna Nandivada is approved.

Majid Sarrafzadeh

Rupak Majumdar

Lei He

Bill Mangione-Smith

Jens Palsberg, Committee Chair

University of California, Los Angeles

2005

To the almighty

Table of Contents

List of Figures

List of Tables

Acknowledgments

When the sun rises in the horizon tomorrow, having witnessed the approval of this thesis, it would do so bearing witness to all the help, affection and cooperation I have gotten throughout my stint as a PhD student both at Purdue as well as UCLA.

The list above is a silent spectator to all the people whom I came in touch, learnt something, and forgot about them in the most ungrateful way. One day, I will remember their contribution and thank god that I met them.

Vita

1977	Born, and still alive!
1998	B.E. (Computer Science), Regional Engineering College, Rourkela, India.
2000	M.E. (Computer Science), Indian Institute of Science, Bangalore, India.
2000–2001	Senior software engineer, Hewlett Packard.
2001–2003	Research assistant, computer science department, Purdue university
2001 summer	Intern, Sun Labs, Burlington.
2003–2005	Research assistant, computer science department, UCLA.
2005–present	Teaching assistant, computer science department, UCLA.

Publications

Efficient Spill Code with SDRAM with Jens Palsberg. In Proceedings of 4th International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp:24-31, October 2003.

Dynamic State Restoration Using Versioning Exceptions with Suresh Jagannathan. To appear Higher Order Symbolic Computation(HOSC) in 2006.

Compile-Time Concurrent Marking Write Barrier Removal with David Detlefs. In the proceedings of the 3rd annual IEEE/ACM international symposium on Code Generation and Optimization, pp: 37-48, March 2005.

Timing analysis of TCP servers for surviving denial-of-service attacks with Jens Palsberg. In the proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, pp: 541-549, March 2005.

SARA: Combining Stack Allocation and Register Allocation with Jens Palsberg. *manuscript*

A framework for implementing and evaluating register allocators with Fernando M Q Pereira and Jens Palsberg. *manuscript*

Abstract of the Dissertation

Combining Stack Location Allocation with Register Allocation

by

Venkata Krishna Nandivada

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2005

Professor Jens Palsberg, Chair

Machine-specific optimizations are important to compiler technology because of the significant performance improvements they can achieve. The increasing cost of memory accesses has expanded the need for machine-specific optimizations aimed at reducing the performance penalties associated with accessing memory. Register allocation, which is the phase of compilation that assigns temporaries to machine registers and/or memory locations, has been one of the most successful machine specific optimization.

In this thesis, we present a new extension to register allocation, which we name stack location allocation (SLA). The SLA phase rearranges the variables on the stack, to facilitate merging of loads and stores, to speed up the memory access. This optimization is particularly advantageous, in the context of processors such as StrongARM with peripheral memories like SDRAM, where multiple loads and stores can be executed efficiently in one single instruction.

We show that the core problem for SLA is NP-complete and present an exact

solution for SLA by implementing it as an integer linear program (ILP). We tested the effect of an independent SLA phase, using both a heuristic based as well as an ILP based register allocator. We found that in both the cases, programs compiled with SLA give better performance. A phase ordering issue arises between SLA and RA, which we solved by replacing the two sub phases by a single integrated phase SARA. We found that SARA outperforms both the deployments of SLA described before.

We show that as an independent phase code generated with SLA phase runs on average 4.5% faster over the baseline gcc with -O2 optimization level. When SLA is integrated with register allocation (SA+RA=SARA), the resulting code is on average 4.1% faster than that of gcc-O2 with ILP based RA followed by SLA and 7.4% faster than that of gcc-O2 with heuristic based RA followed by SLA.

We have developed a new register allocator framework, RALF, that serves as a basis for both the ILP based register allocator and SARA implementations. RALF allows many different register allocation techniques to be implemented independent of the data structures of the underlying compiler and generate end-to-end numbers for different benchmarks. We show the versatility and the ease of use of this framework by implementing a variety of register allocators in the framework and presenting performance results for each.

CHAPTER 1

Introduction

Modern computing systems come with a lot of designer architectures that are quite tailor made for the applications under consideration. Under such considerations it becomes increasingly important that the compilers exploit more of the low level issues to generate better (faster / smaller / ?) code.

One of the first low level optimization that proved it's prowess was register allocation [Cha82, BCT94]. One main reason for the importance associated with the register allocation phase is the increasing gap between processor speed and memory speed. With the advent of many custom memories researchers and engineers have spent a lot of time and focus in trying to reduce memory latencies. Along with better register allocation techniques various compiler-based techniques have been proposed to reduce memory latencies, including compiler-directed prefetching [CCH96] and value prediction [LS96, TA97]. In this thesis, we propose another compiler-based technique, *stack location allocation* (SLA), as an extension to the register allocation phase to reduce memory latency. This phase involves moving and merging memory accesses next to each other and in the process reduces the cost of memory accesses. Such an optimization phase can be effective in processors such as StrongARM, which has load-multiple/store multiple instructions, with SDRAM memory, which facilitates efficient access of multiple memory locations.

We first show that as an independent phase SLA delivers good results com-

pared to gcc at -O2 level of optimization. Further, we show that there exists a phase ordering problem with SLA's efficiency being affected negatively by the previously run register allocation phase. Thus to overcome this phase ordering problem we propose a combined SLA and register allocation phase (SA+RA = SARA). We show that this combined phase generates faster code than a sequence of SLA and RA.

1.1 Background

1.1.1 Register Allocation

A compiler goes through many phases while translating high level code to the target machine code. During the initial phases of the compiling the compiler uses many symbolic registers (or pseudo registers) to represent both the declared variables as well as all the temporary computations. However all the target hardware are constrained by the limited number of available actual registers. The task of register allocator is to map these pseudo registers to real registers and memory locations.

The register allocation problem has been studied in great detail [Cha82, CK91, BCT94, GW96, AG01, PLM01] for a wide variety of architectures. Register allocation problem has been shown to be NP-complete [Set73, LFK99]and researchers have explored both heuristic based [Cha82, CK91, BCT94] as well as practically optimal solutions (e.g. solutions based on genetic algorithms [EA99], solutions based on integer linear programs [GW96, AG01, PLM01]).

Register allocation problem has historically been studied under two subproblems: Register assignment and spilling. Register assignment is the phase of assigning of machine registers to pseudos wherever possible. Spilling is the

combined act of storing a currently used pseudo to memory and reloading back for the next use. Even though, these two subproblems can be solved sequentially, but such an approach leads to inefficient solutions. This is the main reason why researchers have tried to integrate the two subproblems into one super problem and presented solutions for it. The scope of register allocation phase has also increased due to the interdependence of this phase on other sub-phases, such as coalescing, rematerialization, code scheduling etc. Researchers have proposed solutions to these issues by closely integrating the solutions to these sub-phases with the register allocation phase. In this thesis we present a new phase *stack location allocation* as an extension to register allocation phase. We present our experience of generating code in the presence of this new phase, both as a separate post pass to register allocator, and as a combined pass solving the two problems together.

1.1.2 Stack Location Allocation

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 [TCC00] and several network processors, such as the IBM PowerPC405 in the PowerNP NP4GS3 [np4] and the Intel StrongARM in the IXP–1200 [ixpa]. Until now, little has been published work on how a compiler can take advantage of the capabilities for multiple load

and store. In this thesis we present compiler techniques for maximizing the number of multiple load/store instructions, in the context of the Intel StrongARM processor.

Processors like the Stargate [sta] and Intel IXP–1200 and 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) [Sea]. 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 [SKP00]. 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

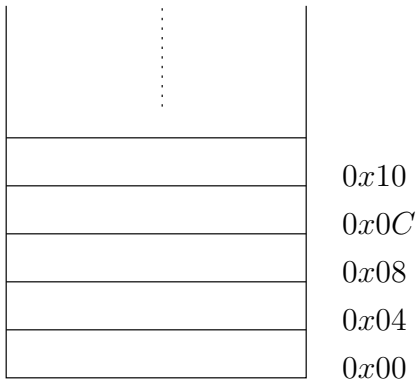

Figure 1.1: Layout of byte addressed SDRAM memory

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 thesis we will concentrate on double loads and double stores only. Investigation of triple loads and stores, and beyond, is left to future work.

To be able to replace two load (store) instructions with a load-pair (store-pair) instruction we need help from both the memory unit and the processor. We will start by studying the memory issues.

Fig. 1.1 shows the layout of the byte addressed SDRAM. If we have to load register r1 from [$0x00$] and r2 from [$0x04$] then we can issue one memory fetch instruction to load from both the locations. And this whole process will take 40 cycles. However, if we have to load $r1$ from [0x00] and $r2$ from [0x08] then we

have to make two different memory access requests and it will take 40+40=80 cycles. In essence, the addresses have to be contiguous at 4 byte boundaries to be able to merge the memory access requests.

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

$$\text{LDR } addr_1\ r_i$$
$$\text{LDR } addr_2\ r_j$$

Let us assume $i \neq j$. As we saw in Fig. 1.1 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:

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

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):

$$\text{eor } r_i,\ r_i,\ r_j$$
$$\text{eor } r_j,\ r_i,\ r_j$$
$$\text{eor } r_i,\ r_i,\ r_j.$$

```
foo(){ int a,b,c;
       ...
       a=c+a;
       return a+e; }
```

(a)

var	old loc	new loc
a	sp+0	sp+4
b	sp+4	sp+8
c	sp+8	sp+0

(b)

Figure 1.2: Issues with current way of stack allocation.

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.

We have shown that it is advantageous to replace two individual loads (stores) with a load-pair (store-pair). The gcc 2.95.2 compiler does not take full advantage of this idea. To understand the reasons let us look at a snippet of a C function in Fig. 1.1.2(a). The function foo, does some computations and then it reaches the addition statement shown. Let us assume that both the variables 'c' and 'a' have been spilled by the register allocator and both have to be loaded just before this addition statement. That is, we have two load instructions next to each other. However, for the compiler to replace these two loads with a load-pair they have

to be accessing consecutive memory locations.

The second column in Fig. 1.1.2(b) shows the locations assigned to these variables. Variable 'a' is assigned sp+0 and variable 'c' is assigned sp+8. This makes it impossible for the current compiler to merge these two loads. However, if stack locations assigned to the variables were done, as shown in the second column in Fig. 1.1.2(b), then variables 'a' and 'c' would have been neighbors and the compiler could have merged the two loads into one single load-pair instruction.

In this thesis we propose a new phase, stack location allocation (SLA) that rearranges variables on the stack to generate efficient code. We present a case for SLA, that is, making a sincere effort in the compiler to maximize the usage of double-loads and double-stores by merging as many single loads and singles stores as possible.

1.2 Our Contributions

We have three main contributions.

1. We implement stack location allocation (SLA) as a post phase to register allocation in gcc. We show that our phase SLA phase makes the code run on average 4% over code generated by gcc at -O2 level of optimization.

2. We show that SLA as an independent phase suffers from phase ordering issues generating from register allocation (RA). And thus to overcome this problem we present a combined phase of SLA and RA, named SARA, that does stack location allocation along with register assignment and spill code generation. Our implementation results in code that runs on average 8% faster than the code that is generated by a compiler doing RA and SLA in a sequence.

3. We also present a framework (RALF), which allows easy coding of the register allocators in a production compiler. The framework, can help different register allocators generate end-to-end execution time numbers without requiring the compiler writer to familiarize with the underlying data structures of a compiler. We show the versatility and ease of use of our framework by implementing and analyzing a variety of register allocators in the framework.

In chapter 2 we present the details about our SLA phase. We show the issue of phase ordering problem arising from SLA and RA along with our solution SARA in section 3. We present our register allocation framework (RALF) in section 4 and conclude in section 5.

CHAPTER 2

Stack location allocation

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 chapter 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.

2.1 Introduction

2.1.1 Background

Popular memory units like SDRAM enable a processor to load and store multiple 32 bit register registers in one instruction. In this chapter we investigate how a compiler can maximize the number of double loads and double stores, while minimizing the number of inversions as an independent phase. 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 [c9999, Section 6.9.1#9], while it prohibits rearrangement of the fields inside aggregate types [c9999, 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 place-

ment function that leads to at least q double loads and double stores, and at most r inversions.

2.1.2 Our Results

First we characterize the complexity.

Theorem The placement decision problem is NP-complete.

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 [GGJ78], 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 chapter 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 chapter. 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.

2.1.3 Example

Consider the C code in Figure 2.1.3. 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 2.1.3 shows the locations of the variables before and after the SLA pass. Notice that a permutation has been done and that after the

```
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

Figure 2.1: Example C program and locations of variables with and with SLA phase

SLA phase, the variables that are accessed together (a and c, b and d, a and e) have consecutive addresses. In Figure 2.2 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 2.1.3. 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

	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 2.2: Code without and with SLA

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.

2.1.4 Related Work

Various compiler-based techniques have been proposed to reduce memory latencies, including compiler-directed prefetching [CCH96] and value prediction [LS96, TA97]. In this chapter 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 [Bar92], and later by many authors [LDK96, AS99, SLD97, PLM01, SP01, LM96, LD98, PDN97] 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 [PLM01]. Sjödin and Platen [SP01] 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 [LDK96] 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 [LM96] 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 [LD98] present a genetic algorithm based approach to generate new offsets to handle different register file sizes and auto increment ranges. Rao and Pande [AS99] give techniques based on approximation algorithms to optimize the access sequence of variables by applying algebraic transformations on expression trees. Sudarsanam et al [SLD97], present a formulation of the storage assignment problem, parameterized by the maximum allowable increment limit and the number of address registers. Panda et al [PDN97] 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 [WL01] talk about register allocation for processors that support bit section referencing. George and Blume [GB03] 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 [AG01, GW96, LFK99, RGL96, Sto97]. 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 [LDK96] assume a fixed evaluation order for each basic block. Rao and Pande [AS99] 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.

2.2 The SLA Algorithm

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

2.2.1 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.

2.2.2 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 diff$[e] = p_2 - p_1$.

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

Objective function. Solving the placement problem contributes to saving cy-

cles 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.

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

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.

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

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

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

The following constraint computes diff.

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

2.2.3 Constraint Solving

We use AMPL [FGK93] to generate the ILP, and CPLEX [CPL00] to solve it. The gcc compiler invokes the constraint generator by providing the data in a file.

Bench characteristics			Compile time (secs)			Xformations			Execution time (secs)		
name	funcs	lines	-SLA	+SLA	% worse	lds	strs	eor	-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
IPCh	76	3453	1.69	2.67	58.0	44	14	9	0.23	0.20	15.1
Classify	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 2.3: Experimental Results

Once constraints are generated the constraint generator calls the solver, which returns the resulting solution to gcc in a file.

2.2.4 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 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.

2.3 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.

2.3.1 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 2.3, columns 2–3.

Our first two benchmarks are from the MediaBench [LPM97] 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 [MBW01] 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. *IPCh* 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: *Classify* is a network packet classifier that classifies ARP and TCP packets. *FireW* 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.

2.3.2 Measurements

We measured the compile time deterioration and the execution time improvement. Figure 2.3, 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 2.3 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.

2.3.3 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 Classify 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 2.3, 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 IPCh, 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 IPCh, 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.

2.4 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, most likely, still be possible to access 32 bit registers which would hold the smaller data units.

CHAPTER 3

SARA: Combining Stack Allocation and Register Allocation

Commonly-used memory units enable a processor to load and store multiple registers in one instruction. We showed in chapter 2 how to extend gcc with a stack-location-allocation (SLA) phase that reduces memory traffic by rearranging the stack and replacing some load/store instructions with load/store-multiple instructions. While speeding up the target code, our technique leaves room for improvement because of the phase ordering of register allocation before SLA. In this chapter we present SARA which combines SLA and register allocation into a single phase. SARA creates a synergy among register assignment, spill-code generation, and SLA that makes the combined phase generate faster code than a sequence of the individual phases. We specify SARA by an integer linear program generated from the program text. We have implemented SARA in gcc, replacing gcc's own implementation of register allocation. For our benchmarks, our results show that the target code is up to 16% faster than gcc with a separate SLA phase.

3.1 Introduction

3.1.1 Background

Processors such as Intel StrongARM together with memory such as SDRAM enable efficient execution of multiple loads and stores in a single instruction. We can find such a combination of processor and memory in Intel's IXP-2400 [ixpb], Stargate (http://www.xbow.com/Products/XScale.htm), Sun MAJC 5200 [TCC00], etc. Multiple loads and stores are particularly useful in connection with register allocation where spill code may need to save and restore multiple registers.

In the previous chapter we show how to extend gcc with a stack-location-allocation (SLA) phase that reduces memory traffic by

- moving some load and store instructions such that they occur in pairs,
- rearranging the stack such that the temporaries used in a pair of load/store instructions have neighboring stack locations, and
- replacing some loads and stores with load/store-multiple instructions.

While speeding up the target code, our technique leaves room for improvement because of the phase ordering of register allocation before SLA.

For an example of the shortcomings of gcc extended with SLA, consider the code snippet in Figure 3.1(a). The code snippet is part of a synthetic benchmark program in which `c` and `d` are needed somewhere after line 3. For the benchmark program, gcc spills the four pseudos `a`, `b`, `c`, and `d` to the memory locations shown in Figure 3.1(b) and generates the code shown in Figure 3.1(c); gcc extended with SLA generates exactly the same code. To see why SLA fails to merge the two loads and the two stores, notice first that the register allocator has done a good

(a)

```
    int a,b,c,d;
1.  c = a;
2.
3.  d = b;
4.
5.  ...
```

(b)

pseudo	line	reg		stack
		gcc	SARA	
a	1	r3	r1	fp-16
b	3	r3	r2	fp-20
c	2	r3	r1	fp-24
d	3	r3	r2	fp-28

(c)

```
ldr   r3, [fp, #-16]   ; load a
str   r3, [fp, #-24]   ; store into c

ldr   r3, [fp, #-20]   ; load b
str   r3, [fp, #-28]   ; store into d
```

(d)

```
sub   r1, fp, #20      ;
ldmia r1, {r1,r2}      ; load a and b

sub   r9, fp, #28      ;
stmia r9, {r1,r2}      ; store into c and d
```

Figure 3.1: Phase ordering problem between register allocation and SLA

job using register r3 both when loading `a` and when loading `c`. However, the use of r3 in both load instructions and both store instructions prevents SLA from moving the instruction for loading `b` to the program point just before the instruction for storing into `c`; the code motion would change the behavior of the program. Thus, the good register allocation is *counterproductive* to merging loads and stores. The compiler can generate better code for the benchmark program by first doing a worse register allocation which uses different registers when loading `a` and when loading `c`. The reason is that now the SLA phase can safely move the two load instructions together and also move the two store instructions together, then replace those instructions with a double-load (ldmia) and a double-store (stmia), and ultimately generate the code shown in Figure 3.1(d).

Another weakness of gcc extended with SLA is that first the register allocator will assign stack locations to all spilled pseudos and then SLA will try to reorganize the stack as best as it can to enable double-loads and double-stores. If SLA does not manage to find the best permutation of the stack locations, then the target code may not contain the highest possible number of double-loads and double-stores. A better approach may be is to let the register allocator know about double-loads and double-stores and do the spilling of pseudos accordingly.

Our observations about gcc extended with SLA suggest that a compiler can do better if register allocation and SLA are more tightly integrated.

> **Question:** Can a combined phase be better than a two-phase sequence of register allocation and SLA?

Our Results. In this chapter we present SARA which combines SLA and register allocation into a single phase. Our technique creates a synergy among register assignment, spill-code generation, and SLA that makes the combined phase generate faster code than a sequence of the individual phases. We specify SARA by an integer linear program (ILP) generated from the program text. Our ILP formulation uses an objective function which estimates the execution time of the memory instructions. We have implemented SARA in gcc, replacing gcc's own implementation of register allocation. For our benchmarks, our results show that the target code is up to 16% faster than gcc with a separate SLA phase.

We specify SARA by an ILP because (1) register allocation can be specified by an ILP [GW96, KW98, LFK99, AG01, FW02, NP04], (2) SLA can be specified by an ILP (chapter 2), and (3) ILPs are often easy to combine. We speculate that it would be much more difficult to build a one-phase combination of register allocation and SLA based on one of the classical non-ILP-based register allocators [Cha82, CK91, BCT94].

While solving ILPs can be slow, we note that all of the following three problems are NP-complete: (a) register assignment [Set73], (b) spill code generation [GJ79], and (c) SLA (see chapter 2). The combination of (a)+(b)+(c) is also NP-complete. We view our ILP formulation of (a)+(b)+(c) as a high-level specification which, as we demonstrate, leads to good target code. We present a technique that enables us to contain the state space explosion and allow the solver to terminate in reasonable time limits. Our proposal uses the variable liveness information that is available to the register allocator in most optimizing compilers. In future work one might investigate how to implement fast approximation algorithms for our ILP formulation.

To show that the combined phase SARA works better than the individual phases performed sequentially, we specify an ILP-based register allocation phase (RA) without SLA. Our results show that RA leads to faster code than the code generated by gcc at O2 level of optimization. Next we re-confirm our results presented in chapter 2 by showing that RA followed by SLA is better than RA alone. And finally we show that the combined phase SARA is better than the sequential composition of ILP-based register allocation and SLA. In slogan form, if P is one of our benchmark programs, and ET denotes an execution time monitor, we have

$$\mathrm{ET}(SARA(P)) \leq \mathrm{ET}(SLA(RA(P))).$$

In related work, Bradlee et al. [BEH91] and Motwani et al. [MPS95] demonstrated how to combine register allocation and code scheduling to obtain faster code. Lerner et al. [LGC02] presented a framework for composing dataflow analyses and thereby overcoming the phase ordering problem. Our approach differs from theirs in that we use and combine ILPs.

In the following section we specify an ILP-based register allocator. In Section 3 we extend the ILP-based register allocator with facilities for SLA; the result is

SARA. In Section 4 we discuss how we control state-explosion problem, and in Section 5 we present our experimental results.

3.2 ILP-based Register Allocation

Our ILP-based register allocator does register assignment and spill code generation. We defined our register allocator with inspiration from the ILP-based register allocators of Goodwin and Wilken [GW96] and of Appel and George [AG01]. The key property of our register-allocator specification is that we can easily add SLA, as shown in the following section. We will now present the three main phases of the register allocator: model extraction, constraint generation, and constraint solving.

3.2.1 Model extraction.

From the input program we extract a model consisting of sets and parameters.

Insts	$\subseteq$	{1..nInsts}	Req	:	Insts $\times$ Pseudos	$\rightarrow$	$\{0,1\}$
Pseudos	$\subseteq$	{1..nPseudos}	Def	:	Insts $\times$ Pseudos	$\rightarrow$	$\{0,1\}$
Regs	$\subseteq$	{1..nRegs}	prevInst	:	Insts	$\rightarrow$	Insts
Loc	$\subseteq$	{1..nPseudos}	joinInst	:	Insts $\times$ Insts	$\rightarrow$	Insts
			callInst	:	Insts	$\rightarrow$	$\{0,1\}$

The set of instructions, pseudos, registers, and stack locations for the pseudos is given by Insts, Pseudos, Regs, Loc, respectively. For the example shown in Figure 3.1, Insts = {1,2,3,4}, Pseudos = {a,b,c,d}, Regs = {1,2,3,4,5,6,7,8,9,10}. The parameter Req(i, p) is set to 1 if instruction i requires pseudo p and hence needs p to be present in a register. The parameter Def(i, p) is set to 1 if instruction i sets pseudo p. The control flow of the program is given by three parameter maps.

The parameter prevInst(i) is a singleton set containing the previous instruction of i if it has only one previous instruction, and null otherwise. The parameter joinInst(i) is the set of previous instructions of i if instruction i is a join point with multiple previous instructions, and null otherwise. The parameter callInst(i) has value 1 if the instruction i is a call instruction, and 0 otherwise.

For each instruction i, the parameter freq(i) returns the frequency of execution of that instruction. In this thesis, we use static estimates of freq(i); alternatively one might use a profiling-based approach. The parameters loadCost and StoreCost give the cost of one single load and one single store respectively. Also a subset of Regs is designated as caller save registers and are represented by callerSaveRegs For the target environment we have the set of caller save registers is {0,1,2,3,9,12}. Each function must save and restore any register that is a callee save register, that is, not a caller save register.

3.2.2 Constraint Generation.

From the input program we generate an ILP whose main purpose is to ensure the following properties: (1) each pseudo is assigned at most one register, (2) each register is assigned at most one pseudo, (3) at any instruction, the number of used registers is bound by the available number of registers, (4) for every definition and use of a pseudo, the pseudo has a register assigned to it, and (5) a pseudo keeps it's mapping to a register, unless the pseudo is no longer live or the pseudo is defined, loaded, or stored.

We will use the following maps. Intuitively, the map PsR maps pseudos to registers for each instruction, the map xDef gives the register map for a pseudo p at a given instruction defining p, the maps SpLoad and SpStore represent the load and store instructions that need to be inserted into the program, and the

map inUse tracks whether a register is used.

$$\begin{array}{lcll} \text{PsR} & : & \text{Insts} \times \text{Pseudos} \times \text{Regs} & \rightarrow \quad \{0,1\} \\ \text{xDef} & : & \text{Insts} \times \text{Pseudos} \times \text{Regs} & \rightarrow \quad \{0,1\} \\ \text{SpStore} & : & \text{Insts} \times \text{Pseudos} \times \text{Regs} & \rightarrow \quad \{0,1\} \\ \text{SpLoad} & : & \text{Insts} \times \text{Pseudos} \times \text{Regs} & \rightarrow \quad \{0,1\} \\ \text{inUse} & : & \text{Regs} & \rightarrow \quad \{0,1\} \end{array}$$

$\text{PsR}(i,p,r)$ returns 1 if pseudo p is present in register r at instruction i. $\text{xDef}(i,p,r)$ returns 1 if pseudo p is defined in instruction i, in register r. Pseudo p will be present in register r in the next instruction. $\text{SpStore}(i,p,r)$ returns 1 if pseudo p is spilled *after* instruction i and is currently mapped to register r. $\text{SpLoad}(i,p,r)$ returns 1 if pseudo p is (re)loaded *before* instruction i into register r. We generate the following constraints.

Each pseudo is assigned to at most one register and each register is assigned to at most one pseudo:

$$\begin{aligned} \forall i \in \text{Insts},\ \forall p \in \text{Pseudos} \quad &: \sum_{r \in \text{Regs}} \text{PsR}(i,p,r) \leq 1 \\ \forall i \in \text{Insts},\ \forall r \in \text{Regs} \quad &: \sum_{p \in \text{Pseudos}} \text{PsR}(i,p,r) \leq 1 \end{aligned}$$

At any program point the number of pseudos that are available in registers is bound by the number of registers available:

$$\forall i \in \text{Insts} : \sum_{p \in \text{Pseudos}, r \in \text{Regs}} \text{PsR}(i,p,r) \leq \text{nRegs}$$

A pseudo that is used in an instruction has to be present in a register at that point:

$$\forall i \in \text{Insts},\ \forall p \in \text{Pseudos} : \sum_{r \in \text{Regs}} \text{PsR}(i,p,r) \geq \text{Req}(i,p)$$

A pseudo being defined needs a register:

$$\forall i \in \text{Insts},\ p \in \text{Pseudos} : \sum_{r \in \text{Regs}} \text{xDef}(i, p, r) = \text{Def}(i, p)$$

A pseudo p retains it's mapping to a register unless it is spilled or another pseudo is mapped to that register. If the instruction has only one previous instruction:

$$\forall i \in \text{Insts},\ p \in \text{Pseudos},\ r \in \text{Regs},\ pr \in \text{prevInst}(i) :$$
$$\text{PsR}(i, p, r) = (\text{SpLoad}(i, p, r) \vee \text{PsR}(pr, p, r) \vee \text{xDef}(pr, p, r)) \wedge \neg\text{SpStore}(pr, p, r)$$

If the instruction is next to a join point and hence have multiple predecessors:

$$\forall i \in \text{Insts},\ p \in \text{Pseudos},\ r \in \text{Regs} :$$
$$\text{PsR}(i, p, r) = (\bigwedge_{pr \in \text{joinInst}(i)} \text{PsR}(pr, p, r) \wedge \neg\text{SpStore}(pr, p, r)) \vee \text{SpLoad}(i, p, r)$$

Pseudos mapped to caller save registers loose their mapping after a call instruction:

$$\forall i \in \text{Insts},\ \forall p \in \text{Pseudos}\ \forall r \in \text{callerSaveRegs} : \text{callInst}(i) \Rightarrow \text{PsR}(i, p, r) = 0$$

A register is used if it is mapped to a pseudo:

$$\forall i \in \text{Insts},\ \forall p \in \text{Pseudos}\ \forall r \in \text{Regs} : \text{inUse}(r) \geq \text{PsR}(i, p, r)$$

3.2.3 Objective function.

Our objective function estimates the execution time of the inserted loads and stores for spilling and for storing and restoring the callee save registers at the beginning and end of a function. The objective of our ILP solver is to minimize SpillCost + CalleeSaveCost where

SpillCost =

$$\sum_{i \in \text{Insts}} \text{freq}(i) \times \left(\sum_{p \in \text{Pseudos},\ r \in \text{Regs}} \left(\begin{array}{c} (\text{SpLoad}(i,p,r) \times \text{loadCost}) \\ + \\ (\text{SpStore}(i,p,r) \times \text{StoreCost}) \end{array} \right) \right)$$

and

CalleeSaveCost =

$$\sum_{r \in \text{Regs}-\text{callerSaveRegs}} \text{inUse}(r) \times (\text{loadCost} + \text{StoreCost})$$

3.2.4 Constraint Solving.

We use AMPL [FGK93] to generate the ILP, and CPLEX (http://www.cplex.com) 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.

The result of solving the constraints for the running example in Figure 3.1 is shown in the following table. (Only tuples with non-zero values are shown.)

PsR	=	{(1,a,r3),(2,c,r3),(3,b,r3),(4,d,r3)}
SpLoad	=	{(1,a,r3),(3,b,r3)}
SpStore	=	{(2,c,r3),(4,d,r3)}
xDef	=	{(1,c,r3),(3,d,r3)}
inUse	=	{(r3,1)}
SpillCost	=	$2 \times \text{loadCost} + 2 \times \text{StoreCost} = 184$
CalleeSaveCost	=	0

3.3 SARA

The advantage of using an ILP-based framework for combining multiple phases is that each phase can be added as a module on top of an already existing ILP. SARA, the combined phase of SLA and RA, is built upon the set of parameters and constraints given for the ILP-based RA in section 3.2. We now present the additional parameters, variables and constraints required for SARA over RA. The new phase SARA requires five additional variables:

$$\begin{array}{lcll} \text{LoadPair} & : & \text{Insts} \times \text{Pseudos} \times \text{Pseudos} & \rightarrow \quad \{0,1\} \\ \text{StorePair} & : & \text{Insts} \times \text{Pseudos} \times \text{Pseudos} & \rightarrow \quad \{0,1\} \\ \text{InverseLoad} & : & \text{Insts} \times \text{Pseudos} \times \text{Pseudos} & \rightarrow \quad \{0,1\} \\ \text{InverseStore} & : & \text{Insts} \times \text{Pseudos} \times \text{Pseudos} & \rightarrow \quad \{0,1\} \\ f & : & \text{Pseudos} \times \text{Loc} & \rightarrow \quad \{0,1\} \end{array}$$

For a given instruction i, and two pseudos p_1 and p_2 ($p_1 \neq p_2$), the map LoadPair(i, p_1, p_2) returns 1 if we can replace the two spill loads by a pair, and 0 otherwise. Map InverseLoad(i, p_1, p_2) returns 1, if pseudos p_1 and p_2 can be paired as one load-pair but would need inversion, 0 otherwise. Maps StorePair and InverseStore behave similarly for stores. The map f maps a pseudo to it's location: $f(p, l)$ returns 1 if pseudo p is placed in location l. Note that, in practice, not all pseudos would need a location.

A pseudo can have at most one location and a location can have at most one pseudo mapped to it.

$$\forall p \in \text{Pseudos} : \sum_{l \in \text{Loc}} f(p,l) \leq 1 \qquad \forall p \in \text{Loc} : \sum_{l \in \text{Pseudos}} f(p,l) \leq 1$$

A pseudo needs a location if it is spilled and/or reloaded.

$\forall i \in \text{Insts},\ p \in \text{Pseudos}$:

$$2 \times \sum_{l \in \text{Loc}} f(p,l) \geq \sum_{r \in \text{Regs}} (\text{SpLoad}(i,p,r) + \text{SpStore}(i,p,r));$$

Two consecutive loads or stores can be replaced by an LDM or STM instruction.

$\forall i \in \text{Insts},\ \forall p_1, p_2 \in \text{Pseudos}$:

$$2 \times \text{LoadPair}(i,p_1,p_2) \leq \sum_{r \in \text{Regs}} (\text{SpLoad}(i,p_1,r) + \text{SpLoad}(i,p_2,r))$$

$$2 \times \text{StorePair}(i,p_0,p_2) \leq \sum_{r \in \text{Regs}} (\text{SpStore}(i,p_1,r) + \text{SpStore}(i,p_2,r))$$

LDM and STM require that the memory locations are consecutive.

$\forall i \in \text{Insts},\ \forall p_1, p_2 \in \text{localPseudos}$:

$$\text{LoadPair}(i,p_1,p_2) \leq 1 \text{ only if } \text{diff}(p_1,p_2) == 1 \text{ else } 0.$$

$$\text{StorePair}(i,p_1,p_2) \leq 1 \text{ only if } \text{diff}(p_1,p_2) == 1 \text{ else } 0.$$

$$\text{diff}(p_1,p_2) = ((\textstyle\sum_{l \in \text{Loc}} l \times f(p_1,l)) - (\textstyle\sum_{l \in \text{Loc}} l \times f(p_2,l)))$$

It may be noted that we do not need to check for the absolute value of diff. This is because the optimizing solver will consider both the options (p_1,p_2) and (p_2,p_1) and can pick the best one.

Inversion takes place if we have $\text{LoadPair}(i,p_1,p_2) = 1$ and the register assigned to p_1 is smaller than that of p_2. In the case of an inversion, we will have to swap the contents of the registers at run-time.

$\forall i \in \text{Insts},\ \forall p_1, p_2 \in \text{Pseudos}$:

$$\text{InverseLoad}(i,p_1,p_2) = \frac{\text{isSmaller}(i,p_1,p_2) + \text{LoadPair}(i,p_1,p_2)}{2}$$

$$\text{InverseStore}(i,p_1,p_2) = \frac{\text{isSmaller}(i,p_1,p_2) + \text{StorePair}(i,p_1,p_2)}{2}$$

$\forall i \in \text{Insts},\ \forall p_1, p_2 \in \text{Pseudos}$:

$$\text{isSmaller}(i, p_1, p_2) = \sum_{r_1 \in \text{Regs}} r_1 \times \text{PsR}(i, p_1, r_1) < \sum_{r_2 \in \text{Regs}} r_2 \times \text{PsR}(i, p_2, r_2))$$

Objective function. The objective function used in SARA is similar to the one used by our ILP-based RA given in section 3.2. The new twist is that SpillCost takes pairs into account. and inversions.

SpillCost =

$$\sum_{i \in insts} \text{freq}(i) \times \left(\begin{array}{ll} \sum_{p \in \text{Pseudos}} \text{SpLoad}(i, p, r) \times \text{loadCost} & - \\ \sum_{p_1, p_2 \in \text{Pseudos}} (\text{LoadPair}(i, p_1, p_2) \times \text{loadPairSave}) & \\ \sum_{p_1, p_2 \in \text{Pseudos}} (\text{InverseLoad}(i, p_1, p_2) \times \text{invLoadCost}) & + \\ & \\ \sum_{p \in \text{Pseudos}} \text{SpStore}(i, p, r) \times \text{StoreCost} & - \\ \sum_{p_1, p_2 \in \text{Pseudos}} (\text{StorePair}(i, p_1, p_2) \times \text{StorePairSave}) & \\ \sum_{p_1, p_2 \in \text{Pseudos}} (\text{InverseStore}(i, p_1, p_2) \times \text{invStoreCost}) & \end{array} \right)$$

Here loadPairSave is the savings that one gets because of replacing two loads by a load-pair and StorePairSave is the savings that one gets by replacing two stores by a store-pair. If loadPairCost is the cost of executing one load-pair instruction (this will include the cost of setting the base register) then loadPairSave is given by (loadCost $-$ 2 $\times$ loadPairCost). Similarly StorePairSave is calculated as (StoreCost $-$ 2 $\times$ StorePairCost). Cost of introducing inversions for loads and stores are given by invLoadCost and invStoreCost respectively. In the model generated by the compiler loadPairCost and StorePairCost , invLoadCost, and invStoreCost are given as parameters.

The result of solving the above constraints for the running example shown in Figure 3.1 is shown below. As can be seen the cost has gone down by nearly

50% as compared to the ILP-based RA in section 3.2. This is because of the introduction of the load-pair and store-pair instructions in the code.

PsR	=	{(1,a,r1),(2,c,r1),(3,b,r2),(4,d,r2)}
SpLoad	=	{(1,a,r1),(3,b,r2)}
LoadPair	=	{(1,a,b)}
StorePair	=	{(4,c,d)}
xDef	=	{(1,c,r1),(3,d,r2)}
inUse	=	{(r1,1),(r2,1)}
SpillCost	=	loadPairCost + StorePairCost = 94
CalleeSaveCost	=	0

Our implementation of SARA uses a superset of the constraints presented in this chapter. The additional constraints take care of (1) pre-colored pseudos (pseudos that require a certain register, as required, for example, in connection with parameter passing), and (2) non-spill memory instructions (generated in the presence of pointer based accesses in the code). A practical register allocator has to take care of these issues to be able to generate executable code. The reader can obtain the full set of constraints from our webpage, http://compilers.cs.ucla.edu/nvk/sara.mod.

3.4 SARA Improvements

In this section we will explain three techniques that are used in SARA, namely two techniques for reducing the size of the ILP state space and one technique for improving the quality of the generated code.

3.4.1 Reducing the size of the ILP state space.

Our first technique uses liveness information. Notice first that the domain of the pseudo-to-register map PsR is Insts × Pseudos × Regs. However, for a pseudo to be assigned a register, the pseudo has to be live, that is, the map PsR is valid only at those instructions where the pseudo is live. For our benchmarks, most of the pseudos are live in only small parts of the program. So we define PsR only for live pseudos. Similarly, we define SpLoad, SpStore, LoadPair, and StorePair only for live pseudos. By the same token, we define constraints only for defined ILP variables. Our focus on live pseudos let us reduce the number of variables and constraints by a big factor. We have tried a version of SARA without this optimization on our benchmark programs, and in many case the preprocessor that translates the constraints specified in high level language (AMPL) to a format that is understood by the solver (CPLEX) runs out of memory and fails. With the liveness-based optimization in place, SARA does not run out of space when handling our benchmark programs.

Our second technique manages the number of ILP variables needed to represent the generated load and store instructions. Our technique inserts a dummy instruction after each instruction, generates load instructions only before real instructions, and generates store instructions only after dummy instructions. A dummy instruction does not use any pseudos nor define any; we use dummy instructions as place holders for spill instructions. Let us now explain the details and merits of dummy instructions in more detail. We are trying to track the mapping of pseudos to registers at each instruction. However, sometimes it is not sufficient to know the mapping of a pseudo just at each instruction! For example, in the code fragment without dummy instructions:

$$
\begin{aligned}
i_1: &\quad x = y + p \quad ; \textit{// } p \textit{ dies after } i_1 \\
i_2: &\quad y = y + z \quad ;
\end{aligned}
$$

let us assume pseudo x has to be spilled (because of register pressure) to memory after the instruction labeled i_1 but before i_2, and let us assume pseudo z has to be loaded before i_2. In the case where we do not have any more free registers, we could use the same register (say $r1$) for p, x and z. Notice that because x is being set, x needs a register. But since x will be spilled that register will be free immediately afterwards and can be used for loading z. So we have a mapping of x to $r1$ between i_1 and i_2. But at i_1, p is mapped to $r1$, and at i_2, z is mapped to $r1$. This leads to the situation that x does not have a mapping to $r1$ in PsR. To avoid such situations, we inserted a dummy instruction after each instruction before generating the ILP:

$$
\begin{aligned}
i_1: &\quad x = y + p \quad ; \textit{// } p \textit{ dies after } i_1 \\
d_1: & \\
i_2: &\quad y = y + z \quad ; \\
d_2: &
\end{aligned}
$$

The register allocator can assign register $r1$ to pseudo x at the dummy instruction d_1. Additionally, the register allocator can emit an instruction to spill x after d_1, and an instruction to load z before i_2, thereby establishing the desired pseudo-to-register mapping. Our notion of dummy instructions is related to the notion of points between instructions that was used by Appel and George [AG01]. Instead of using dummy instructions or points between instructions, one might find a way to allow the generation of loads and stores before and after every instruction, although we believe such an approach is more awkward.

3.4.2 Improving the quality of the generated code.

SARA can benefit from having freedom to move the spill and reload instructions around. Perhaps surprisingly, the use of strict (exact) liveness information can lead to the generation of inefficient code. For example, in code for copying structures, we come across patterns like:

$$
\begin{array}{lll}
 & & // \ x_1,\ x_2,\ y_1,\ y_2 \ \textit{are dead} \\
i_1\text{:} & y_1 = x_1; & // \ \textit{live}\ x_1 \\
i_2\text{:} & & // \ x_1 \ \textit{and}\ y_1 \ \textit{are dead} \\
i_3\text{:} & y_2 = x_2; & // \ \textit{live}\ x_2 \\
i_4\text{:} & & // \ x_2 \ \textit{and}\ y_2 \ \textit{are dead} \\
\ldots & &
\end{array}
$$

Here x_1, x_2, y_1, y_2 could be globals or be accessed by globals. We must load x_1 before instruction i_1 and x_2 before i_3. Recall that a load/store requires that the pseudo is live. Forcing the such liveness constraints would constrain SARA so much that it cannot move these two loads together. The same logic holds for the spill of pseudo y_1 and y_2 after instruction i_1 and i_3. Assuming that we have an additional register for the duration of these instructions, and the liveness constraints were a bit relaxed, we would give SARA a bit more breathing room to pair up more loads and stores. For example, if we deliberately make the liveness information a bit more conservative and convey to SARA that x_2 is live at i_1 as well, then SARA could generate a load-pair for x_1 and x_2. A similar argument can be given for y_1 and y_2 as well. This leads to an interesting trade off issue: strict liveness reduces the search space and state space but might result in inefficient code.

We have experimented with relaxing the liveness information by different amounts: (a) strict liveness, (b) liveness extended to basic blocks—each pseudo

is live from the beginning of the basic block until the end; unless it dies in between, (c) liveness relaxed by three instructions. Let us consider (c) in more detail. If a pseudo is live starting at instruction i_1, then the pseudo is assumed to be live starting at $i_1 - 2 \times 3$ (multiplied by 2, to take care of the dummy instructions) unless i_1 is one of the first three instructions in the basic block. And if it is, then the pseudo is assumed to be live starting from the beginning of the basic block until it's death or end of basic block. We arrived at the magic number three from our experience with the benchmarks code. Our experience confirmed our belief that most of the need for code motion arises in code that does copying of structures, etc. In such cases, relaxing the liveness by three instructions is effective.

From our experience, we found that case (b) above, even though it gives more flexibility to the solver to move the spill code, it often resulted in large data sets that causes the ILP solver to return no feasible solution even after a lengthy execution. We present in this chapter our experience with cases (a) and (c). We refer to the case (c) as SARA and case (a) as SARA$_s$ (the subscript denoting *strict* liveness).

3.5 Experimental Results

We have implemented SARA in gcc-2.95.2, replacing gcc's own implementation of register allocation, and we have tested the target code from the new compiler on a Stargate platform. Stargate has a StrongARM/XScale processor and 64MB SDRAM and no cache. We have drawn our benchmark programs from a variety of sources:

- Stanford Benchmark suite: The first four benchmarks small and simple,

but typical of the subroutines of many other benchmarks.

- NetBench: Route and url are network related benchmarks from the NetBench [MBW01] suite. Route is an implementation of IPv4 routing according to RFC 1812, and url is a switching protocol that implements url based switching.

- Pointer-intensive benchmark: This benchmark suite is a collection of pointer-intensive benchmarks [ABS94]. Yacr2 is an implementation of a channel router and Ft is an implementation of a minimum spanning tree algorithm [FT87].

- The last two benchmarks are taken from the comp.benchmarks FAQ at http://www.cs.wisc.edu/~thomas/comp.benchmarks.FAQ.html. The c4 benchmark is an implementation of the connect-4 [All88] game and mm is an implementation of nine different matrix multiplication algorithms.

The static characteristics and compile time statistics of these benchmarks are presented in Figure 3.2. The static characteristics we present here include the number of lines of C code, the number of instructions seen by the ILP solver (which depends on the number of RTL instructions in the intermediate representation of the program), and the number of functions. Due to space constraints, we limit ourselves to presenting compile time statistics for three different register allocators: gcc's default register allocator followed by SLA, our ILP-based RA followed by SLA, and SARA (with the liveness information extended to three instructions, see section 3.4). For each of these combinations we present an estimate of the number of memory accesses; the number of loads and stores (mem), the number of load-pair/store-pair instructions (pr) inserted, and the number of callee save registers (csr) used.

bench	loc	#rtl	nfn	gcc+SLA			RA+SLA			SARA		
				mem	pr	csr	mem	pr	csr	mem	pr	csr
sieve	39	134	3	0	0	9	0	0	9	0	0	9
matmul	56	254	6	9	2	22	9	0	20	7	6	19
perm	34	112	3	5	0	14	5	0	12	4	2	12
queen	58	144	4	11	0	14	12	1	11	8	5	11
route	2246	4672	23	519	4	110	506	6	116	546	19	107
url	790	1264	12	115	8	62	120	5	56	120	8	58
yacr2	3979	10838	58	1060	8	123	1003	6	123	1109	24	142
ft	2155	3218	35	219	5	92	225	9	87	230	14	106
c4	885	3388	21	189	3	289	190	7	305	184	18	320
mm	647	2884	14	386	9	130	375	4	116	380	23	92

Figure 3.2: Benchmark characteristics and compile time statistics

All these benchmarks have the common characteristic that they are non-floating point benchmarks. (We had to edit few of them to remove some code that uses floating point operations; we did so only after ensuring that the code with floating point operations is not critical to the behavior of the program.)

Studying the compile time characteristics gives a good insight into the way SARA works. We can see that in the compile time statistics, SARA outperforms both gcc+SLA and RA+SLA by a big margin in terms of the number of pairs generated. Notice, though, that SARA sometimes uses more callee save registers. because of the added register pressure that comes from pairing up loads and stores. Another point that can be easily noticed is that in some cases, such as c4, SARA and RA+SLA are generating more memory instructions than gcc. This is because the constraints use the frequency of the instruction as a parameter to compute the cost of the objective function. And in such cases, generating

Benchmark	Exec Time (seconds)					
	gcc-O2	RA	gcc+SLA	RA+SLA	$SARA_s$	SARA
sieve	9.26	9.26	9.26	9.26	9.26	9.26
matmul	71.59	68.19	67.49	67.02	66.45	66.28
perm	154.45	151.26	146.90	143.24	140.10	140.10
queen	27.33	24.39	26.80	23.39	22.90	22.24
route	20.9	18.91	18.82	18.10	17.8	17.18
url	10.85	10.36	10.55	10.36	9.86	9.86
yacr2	4.40	4.21	4.30	4.11	3.99	3.95
ft	46.25	45.26	46.15	45.26	45.26	43.21
c4	42.3	41.1	42.19	40.53	40.23	39.65
mm	330.02	326.2	326.5	324.21	322.60	311.32

Figure 3.3: Execution time numbers

loads/stores outside the loop is a better option. One final point to note here is that, for benchmarks route, yacr2, ft and mm, SARA generates more loads/stores than our ILP-based register allocator. The reason is that by generating more loads and stores in non-loop code and generating load-pairs in the loops SARA is able to reduce the overall cost.

We do not give detailed compilation times; our solver sometimes took more than 30 minutes and we had to terminate CPLEX and work with a perhaps suboptimal solution. The total compilation time for all the benchmarks is in the order of hours.

We now present the execution time numbers for the benchmarks. In Figure 3.3 we present the time each benchmark took to run when compiled with different compilers. Each of these is compiled at the -O2 level of optimization.

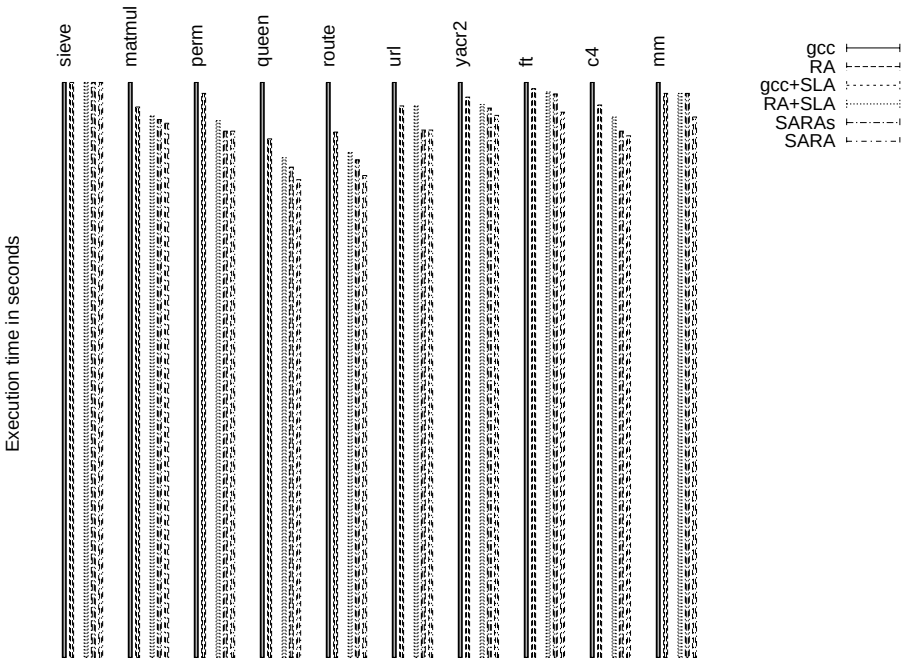

Figure 3.4: Normalized execution times

.

To get an overall comparison of the different register allocators, we present the normalized execution time numbers in Figure 3.4.

Our experience can be represented in a lattice as shown in Figure 3.5. We use the notation $A \leq B$ to denote that time taken to execute code when compiled with A less than or equal to the time taken to execute the same when compiled with B.

Let us now analyze the results in more detail. Sieve is one benchmark where no spill code was needed and gcc's register allocator and our register allocator both perform in the same way. For benchmarks matmul and route, gcc+SLA performs better than RA, indicating that SLA in itself is fairly powerful. For other benchmarks RA is doing better than gcc+SLA, showing that our ILP-

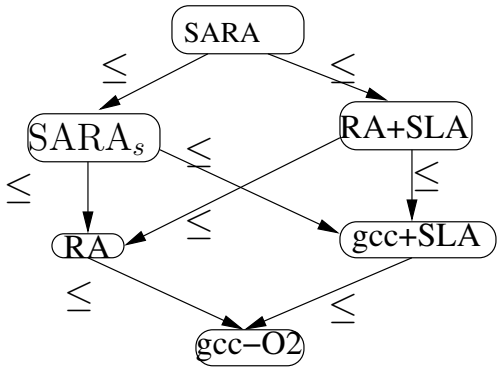

Figure 3.5: A comparison of different register allocator schemes

based register allocation is giving better results than gcc's default module run followed by SLA. For ft, RA + SLA does not give any improvement over RA. That is because SLA could not introduce many pairs in the frequently executed code. Also SARA$_s$ is not giving much improvement either. That's because the ILP solver could not generate many pairs with the strict liveness constraints. However SARA does show an improvement which is due to the relaxed bounds. Theoretically one can imagine cases where RA+SLA could be doing better than SARA$_s$ or even SARA, but we did not find any such cases in our benchmarks. Further experimentation may reveal such cases.

A general point to note about the numbers is that there is a appreciable amount of tension between the number of callee save registers used, the number of normal loads and stores, and the pairs inserted. As a result, SARA shows a significant but not earthshaking improvement over the other register allocators. Overall, we see that SARA yields improvements up to 16% compared to the gcc compiler's own register allocator extended with SLA, and up to 8% compared to our own ILP-based register allocator followed by SLA. On average (excluding the numbers for sieve), the improvements are 7.4% and 4.1% respectively.

3.6 Conclusion and Future work

We have presented an ILP-based approach to combining register allocation and stack location allocation. We have shown that doing these optimizations together gives better results than doing them separately in sequence.

In future work, one might implement SARA using fast heuristics and compare the results to the results of solving the ILPs using CPLEX. One might also add register coalescing, register rematerialization, etc. to SARA and study the effect on code quality and compilation time.

CHAPTER 4

RALF: A register allocation framework

Implementing a given register allocation technique to obtain executable code is a non-trivial task. This task becomes harder when the implementation testbed is an industrial strength compiler. A quick look at some of the recent publications on register allocators show that few actually generate executable code. The majority of researchers address this issue by presenting static compile time numbers such as number of registers used and number of spill instructions inserted. But such statistics have the disadvantage that they do not reflect the real state of affairs vis-a-vis the real impact of the given algorithm in the compiler. They do not take into account (a) how much impact the new phase will have in the presence of other optimizations, and (b) dynamic behavior of the program in the presence of user inputs, loops and function calls.

We present here a gcc based framework (RALF) over which different register allocators can be easily specified and end-to-end results obtained to evaluate the efficiency. We show the versatility of our framework by implementing seven different register allocators and comparing their effect. This framework can be used to implement many register allocators without actually dealing with the details of the underlying compiler.

4.1 Introduction

The register allocation problem has gotten a lot of attention due to it's pivotal role in compilation of high level programs to machine code. It is probably the most important step required while translating code to machine code as this is the only step in which variables (temporaries or pseudos) are actually mapped to real machine registers. Also compared to any other phase, register allocation is known to have the largest impact on the generate code in terms execution time [HP02]. And as one can easily guess, this phase forms a big part of the compiler code (for example in lcc [FH95], the register allocator forms 10% of the code and in gcc-2.95.2 it forms around 12% of the code). Correspondingly, it's interaction with the rest of the phases is also significant and for any non-trivial compiler it is complicated. These complications have proved as a big obstacle for the researchers interested in pursuing research in this area.

A quick look at the recent publications on register allocators show that few actually implement the register allocator in a production compiler and test the executable code to check for performance improvement. Majority of researchers present static compile time numbers such as number of registers used and number of spill instructions inserted. Even though, these numbers do give an estimate of the direct impact of the register allocator, it still does not tell the real impact of the register allocator on the compiler. They do not take into account (a) how much impact the new phases will have in the presence of other optimizations, (b) dynamic behavior of the program in the presence of user inputs, loops and function calls. The missing link in this chain of action is a framework that can provide just the relevant input to the register allocator and take the output of the register allocator and generate executable code. We present in this thesis a framework that can be used for many different register allocators. The framework specifies

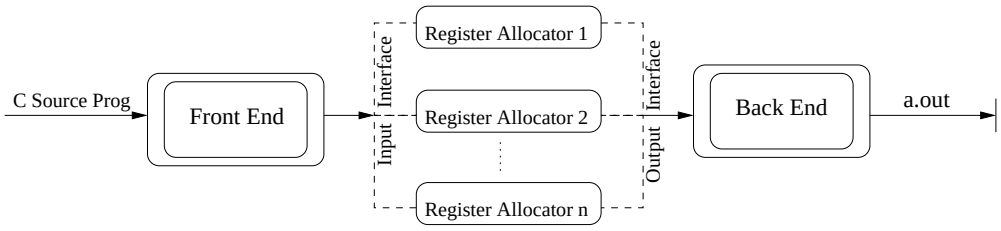

Figure 4.1: Framework block diagram.

an input and output interface for the register allocator which is quite simple. The register allocator can be plugged into the framework as an independent module. The framework is general enough to handle coalescing, stack location allocation, and pairing of loads and stores along with standard register allocation directions specified by the register allocator. The only restriction on the allocator is that it has to adhere to the input/output specification.

In this thesis we show that this framework is quite general and easy to use, by implementing seven different register allocators in this framework and comparing their behavior. This provides a good way of comparing different register allocators as the framework treats all of them with similar input and output requirements.

In the following sections we describe the framework in detail, and then present our experience with a set of seven register allocators. Finally we present some of our observations and limitation of our framework.

4.2 Framework description

We present in Fig 4.1 block diagram for our register allocation framework. It has a front end that parses a given source program and after doing different optimizations it calls the specified register allocator. Once the register allocator has finished with allocating, the register allocator passes the control back to the

framework with the decisions it made about the pseudos. The framework uses this output of register allocator to do the actual register assignment, spill code assignment and stack location allocation and then generates executable code.

One important requirement of a register allocation framework is that it should be able to hide most of the compiler specific details from the programmer. However it must also include all the information that is required by a register allocator, for example, the use-def information, liveness information, pre-colored pseudos (pseudos that already have hard registers assigned), known loads and stores, move instructions etc. Similarly, the framework should be flexible enough to understand different outputs of different register allocators, e.g. pseudo to register mapping, spill loads and stores, coalescing, stack location allocation, pairing of loads and stores etc. Also along with these the framework should be able to do mundane tasks like loading and storing of callee save registers at the entrance and exit of each function, and filling up the holes in the allocated stacks etc. In the following subsections we will present the interfaces our framework presents to the register allocators and argue about the ease of it's use in the following sections by implementing a variety of register allocators.

4.2.1 Input Interface

The input to the register allocator is a model of the given input program, presented in terms of sets and parameters. A set is a symbolic enumeration and a parameter can be a scalar value or a collection of values indexed by set(s). The language we choose is similar to the language used for the data input in AMPL [FGK93].

The inputer interface has two basic components: Program specific information, and target architecture specific information.

Program specific information

The sets of instructions, pseudos, and locations for the pseudos are given by Insts, Pseudos and Regs respectively.

$$\begin{aligned} \text{Insts} &\subseteq \{1..\text{nInsts}\} \\ \text{Pseudos} &\subseteq \{1..\text{nPseudos}\} \\ \text{Loc} &\subseteq \{1..\text{nPseudos}\} \end{aligned}$$

where nInsts is the maximum number of instructions, nPseudos is the maximum number pseudos present in the program. Each instruction is considered to have a (possibly empty) set of required pseudos and can set a pseudo or a register. At each instruction the liveness information is given for each pseudo. Problem parameter Live(i, p) is set to 1 if pseudo p is live at instruction i.

$$\text{Live} \subseteq \text{Insts} \times \text{Pseudos} \rightarrow \{0, 1\}$$

Problem parameter Req(i, p) is set to 1 if instruction i requires pseudo p and hence need to be present in a register. And Def(i, p) is set to 1 if instruction i sets pseudo p.

$$\begin{aligned} \text{Req} &\subseteq \text{Insts} \times \text{Pseudos} \rightarrow \{0, 1\} \\ \text{Def} &\subseteq \text{Insts} \times \text{Pseudos} \rightarrow \{0, 1\} \end{aligned}$$

The control flow of the program is given by three maps:

$$\begin{aligned} \text{prevInst} &\subseteq \text{Insts} \rightarrow \text{Insts} \\ \text{joinInst} &\subseteq \text{Insts} \times \text{Insts} \rightarrow \text{Insts} \\ \text{callInst} &\subseteq \text{Insts} \rightarrow \{0, 1\} \end{aligned}$$

Parameter prevInst(i) is a singleton set containing the previous instruction of i if it has only one previous instruction, null otherwise. Parameter joinInst(i) is the

set of previous instructions of i if instruction i is a join point with multiple previous instructions, null otherwise. Parameter callInst(i) has value 1 if instruction i is a call instruction, 0 otherwise.

A subset of instructions are declared as move instructions. These instructions can be pseudo-pseudo or pseudo-register move instructions. The source and destination of the move instruction can be found from the Req and Def parameters. This information can be used by register allocators doing coalescing.

$$\text{moveInst} \quad \subseteq \quad \text{Insts} \quad \rightarrow \quad \{0, 1\}$$

For each instruction i, parameter freq(i) returns the frequency of execution of that instruction. We use static estimations for obtaining the frequencies of each instruction. Studying the possible improvements by using profile based approaches is beyond the scope of this thesis and we leave it as a future work.

Architecture specific information

Architecture specific information chiefly deals with information about different registers and costs of different operations.

$$\text{Regs} \quad \subseteq \{1..\text{nRegs}\}$$

where nRegs is total number of registers available to the register allocators. Parameters loadCost and StoreCost give the cost of one single load and one single store respectively. Similarly loadPairCost and StorePairCost give the cost of a load-pair and store-pair instruction respectively. Also a subset of hard registers are designated as caller save registers and are represented by callerSaveRegs These are the registers whose contents are not saved across calls.

$$\text{callerSaveRegs} \quad \subseteq \quad \text{Regs} \quad \rightarrow \quad \{0, 1\}$$

```
p1 = 1;
p1 = p1 + 2;
p2 = p1 + 3;
```

Figure 4.2: Sample input program, using two pseudos.

Each function must save and restore any register that is not a caller save register (i.e. callee save register).

Input interface requirements

We impose a set of requirements on the framework generated input to the plugged in register allocator. These requirements serve as a set of guarrantees regarding the input to the register allocator. These requirements have been put in place to ensure a simpler and convenient program model for the plugged in register allocator.

- `Three address codes`: The model of the program presented to the register allocator is similar to three address codes; each instruction has at most two uses and an optional definition.
- `Liveness`: The liveness informtion output is conservative. That is, if a pseudo is live in the program then the liveness information given in the `Live` section will reflect that.

4.2.1.1 Example input interface

We show in Fig. 4.3 a sample of the generated interface for the program shown in Fig. 4.2.

It shows that it has three instructions, two pseudos and the machine has three registers. Parameters are specified by indexing over these sets and hold integer values. We only output here parameters and value pairs for non zero values.

As shown the register r0 is a caller save register, each instruction has a (static) execution frequency of 1, and pseudos p1 is live in all the instructions and pseudos p2 is live only in the last instruction. Parameters def and req give the def and use information: pseudo p1 is defined in instruction i1 and i2 and pseudo p2 is defined in i3. Similarly, instruction i2 uses pseudo p1, and instruction i3 uses pseudos p1 and p2. In the end, the framework outputs a set of scalar parameters; number of registers, total number of instructions, number of pseudos, cost of single load, cost of a load-pair, cost of inversion for a load-pair, store cost, cost of a store-pair, and the cost inversion for a store-pair.

In this thesis, we omit some more parameters that give information about jump instructions, pre-colored registers, jump targets, known-loads (load instructions that are already present), and known-stores (store instructions that are already present). A complete guide to our framework can be found at:

http://compilers.cs.ucla.edu/ralf

4.2.2 Output Interface

The register allocator that is plugged into the framework must assign registers to pseudos, output any spill and reload instruction, and assign stack locations to spilled pseudos. All this information is fed back to the framework which then uses this information to generate executable code. The output has ten different sections corresponding to different types of information that a register allocator might want to convey. Each section consists of a set of tuples as described below.

```
set insts := i1 i2 i3 ;
set pseudos := p1 p2 ;
set regs := r0 r1 r2 ;
set loc  := p1 p2;
param: callerSave:=
r0 1 ;
param: freq:=
i1 1
i2 1
i3 1 ;
param: Live:=
i1 p1 1
i2 p1 1
i3 p1 1
i3 p2 1 ;
param: prevInst:=
i1 i2 1
i2 i3 1 ;
param: joinInst:=;
```

```
param: def:=
i1 p1 1
i2 p1 1
i3 p2 1 ;
param: req:=
i2 p1 1
i3 p1 1
i3 p2 1 ;
param: moveInst:= ;
param: callInst:= ;
param nRegs := 3;
param nInsts := 3 ;
param nPseudos := 3;
param loadCost := 41;
param loadPairCost := 42;
param invLoadCost := 3;
param storeCost := 51;
param storePairCost := 52;
param invStoreCost := 6;
```

Figure 4.3: Sample input interface data.

To minimize the file I/O the framework requires that the register allocator output just the non zero entries for each tuple in each of the sections, wherever applicable. This information can be classified into three categories, register assignment information, generated spill code, and any possible new instructions:

Register assignment information:

- At each instruction mapping of each pseudo to register is given by the section `PsR`. This section consists of tuples of the form (i, p, r), signifying pseudo p is present in register r at instruction i.
- For each pseudo set in an instruction the target register is given in the `xDef` section. This section consists of tuples of the form (i, p, r), signifying pseudo p is set in register r at instruction i.

Spill information:

- For each pseudo section `f` gives the assigned stack location number or -1 otherwise. Each tuple is of the form (p, l), signifying pseudo p gets location l.
- Pseudo reload information is given by the `SpLoad` section. This section consists of tuples of the form (i, p, r), signifying pseudo p is loaded in register r before instruction i. If two loads can be replaced by a load-pair instruction then that is given in the `LoadPair` section. Each tuple in this section is of the form (i, p_1, p_2), signifying pseudo p_1 and p_2 are loaded before instruction i and can be combined to make a load-pair instruction. For each tuple in the `LoadPair` section, the information about the requirement of *inversion* is given in `InverseLoad` section. Each entry in this section is 1, if the

corresponding entry in the `LoadPair` section requires an inversion, and 0 otherwise.

- Pseudo spill information is given by the `SpStore` section. This section consists of tuples of the form (i, p, r), signifying pseudo p is stored from register r after instruction i. If two stores can be replaced by a stores-pair instruction then that is given in the `StorePair` section. Each tuple in this section is of the form (i, p_1, p_2), signifying pseudo p_1 and p_2 are stored after instruction i and can be combined to make a store-pair instruction. For each tuple in the `StorePair` section, the information about the requirement of *inversion* is given in `InverseStore` section. Each entry in this section is 1 if the corresponding entry in the `StorePair` section requires an inversion, and 0 otherwise.

New instructions:

- If the register allocator wants to insert any move instruction (because of coalescing or any other pass), it can instruct the framework to do so by the `moveInst` section. Each tuple in this section is of the form (i, r_1, r_2), signifying that r_2 is moved to r_1 before instruction i.

- Capabilities to insert bitwise operations is one more popular requirement for some of the register allocators. Such a feature is helpful in bitwidth aware register allocation schemes (for example [TG03]). (We do not show these operations in the examples that follow for brevity.)

```
PsR :=
i2 p1 r0
i3 p1 r0 ;
xdef :=
i1 p1 r0
i2 p1 r0
i3 p2 r0 ;
spLoad:= ;
spStore:= ;
```

```
f:=
p1 -1
p2 -1 ;
loadPair:=;
storePair:=;
inverseLoad:=;
inverseStore:=;
moveInst:=;
```

Figure 4.4: Sample output interface data.

Sample output interface

Fig 4.4 presents a sample output from a register allocator for the sample code shown in Fig 4.2. As it is obvious, one register is enough to do the register allocation in this program. It can be seen that the register used is the caller save register `r0`, and hence the allocator need not save / restore callee save registers. And since both the pseudos have been placed in registers, they do not need a place in the stack (and hence negative values in the `f` section).

4.2.3 Correctness issues

The framework we present here, has a number of checks built into it to ensure that the plugged in register allocator does preserve the syntax of compilation and semantics of register allocation.

Syntactic constraints

Syntactic constraints are simple checks to ensure that every entry output by the register allocator is valid.

- Every instruction, pseudo, register and location specified by the register allocator in its output must be from the set of Insts, Pseudos, Regs, and Loc respectively.

Semantic constraints

Semantic constraints are checks to enforce the underlying semantics of register allocation.

- Every *set* instruction (declared using the Def map) must have a target register.
- Each used temporary must have a register assigned to it. If an instruction uses a pseudo then it must be available in a register. Also, if an instruction sets a pseudo, then the pseudo must be assigned the target register.
- Every spill-store / reload requires the pseudo to be live at that point.
- Every spill-store requires that the pseudo be available in the source register of the spill store before the location of that spill-store.
- Every reload requires that the pseudo be available in the in the destination register of the reload instruction after the location of that reload.
- Every pseudo that is loaded or stored must have a stack location.
- A double-load instruction before any instruction i requires that there are two load instructions before i.

- A double-store instruction after any instruction i requires that there are two after instructions before i.

- If two pseudos p_1 and p_2 are loaded (or stored) using a load-pair (or store-pair) instruction then they must be assigned neighboring locations.

4.3 Versatility: Test by implementation

We show the versatility of our framework by implementing a variety of register allocators. Each of the register allocator has a different input requirements and hence poses different type of challenge to the framework. We chose different register allocators to cover a spectrum of typical requirements of different register allocators. The details of the register allocators we use are presented in the following subsections.

4.3.1 Naive Register Allocator

The most naive register allocator would load each pseudo before each use and store it back after each definition. The pseudo code for such an allocator is presented in Fig. 4.3.1. Becaue of the input interface requirements presented in section 4.2.1 the algorithm assumes that there will be at most two pseudos used and at most one pseudo defined in any instruction. Thus, there will be at most two loads before any instruction and after each instruction there will be at most one store instruction. The naive allocator requires that there will be at least two free registers, which is the minimum number of registers required to do register allocation for code in three address form. This register allocator though is only of academic interest, can still be used as a first level test case for a register allocation framework.

```
function NaiveRegAlloc()
    for each instruction i do
        for p1 and p2 used in i
            load p1 before i into register r4
            load p2 before i into register r5
        for p3 defined in i
            set r4 as the target register for i
            store p3 after i from register r4
```

Figure 4.5: Pseudo code for naive register allocator.

For the code snippet shown in Fig. 4.2, the output generated by the naive register allocator is shown in in Fig. 4.6 and the assembly code generated is shown in Fig. 4.7. Owing to the simplicity of the allocation scheme the code generated is obviously inefficient. The framework places pseudos `p1` and `p2` at the memory locations pointed by `sp-4` and `sp-8` respectively, where `sp` is the stack pointer. The loads and stores before and after every `mov` and `add` instructions are in accordance with the directives, given by the register allocator, shown in 4.6.

4.3.2 Linear Scan Register allocator

Linear scan register allocation was proposed by Poletto and Sarkar [PS99] and is popular for its speed. The allocator assumes a linear representation for the input program. That is, the set of instructions are countably finite. (Note, any program can be presented in an linear form by many ways: for example doing a depth first search over the control flow graph is one such option, generated

```
PsR :=
i2 p1 r0
i3 p1 r0 ;
xdef :=
i1 p1 r0
i2 p1 r0
i3 p2 r0 ;
spStore:=
i1 p1 r0
i2 p1 r0
i3 p2 r0 ;
```

```
f:=
p1 p1
p2 p2 ;
spLoad:=
i2 p1 r0
i3 p1 r0 ;
loadPair:=;
storePair:=;
inverseLoad:=;
inverseStore:=;
moveInst:=;
```

Figure 4.6: Output of Naive register allocator for the code snippet in Fig. 4.2.

```
mov r0, 1
str [sp-4], r0

ldr r0, [sp-4]
add r0, r0, 2
str [sp-4], r0

ldr r0, [sp-4]
add r0, r0, 3
str [sp-8], r0
```

Figure 4.7: Assembly code generated from the register alloctor output in Fig. 4.6

is shown in Fig. 4.7. we use here.). This allocator depends on the live intervals information which is computed easily from the variable liveness information. Two intervals are considered to be interfering if they overlap. The goal of linear scan algorithm is to allocate registers to as many intervals as possible from a given set of registers such that no two overlapping intervals get the same register.

The basic idea of the algorithm is as follows: At the beginning of each new interval, the allocator tries to see if the number of live intervals is less than the available number of registers. If so, then it allocates one of the available registers to the new live range. Else it spills one of the live ranges to make a register available and then assigns this registers to the new live range. The spilled intervals set is given by a set of pairs of pseudos and instructions (p, i), which denotes that pseudo p is live at instruction i but has its interval spilled. The candidate live range for spilling can be chosen by different heuristics and accordingly the quality of the code will vary. For this thesis we chose a simple heuristic; the end point of the interval, that is, the interval that whose end point is farthest from the current point is spilled. For each pseudo and instruction pair (p, i), corresponding to any spilled interval

- If p used in i (given in Req map), we reload the pseudos from the memory using two available registers before i.
- If p is set in i (given by the Def map), we write to an available register and generate spill code to store that register back to the location of the pseudo after i.

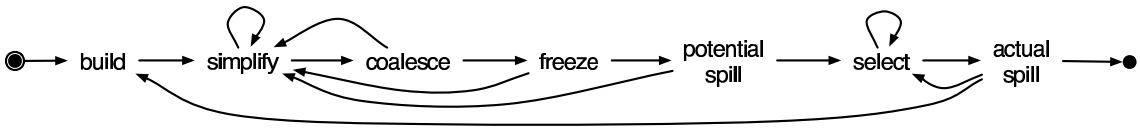

Figure 4.8: Iterated register coalescing.

4.3.3 Iterative Register Coalescing

George and Appel proposed iterative register coalescing [GA96] to do aggressive coalescing along with graph coloring based register allocation. The techniques proposed have been found to be improvements over Chaitin [Cha82] and Briggs [BCT94] methods in terms of elimination of move instructions and overall execution time. The goal of the algorithm is to identify as many opportunities as possible to coalesce, to attach the coalesced pseudos together with same register, remove the move instruction and as a result reduce the register pressure.

The algorithm shown in Fig. 4.8 has five main phases over which it iterates selectively.

1. `Build`: Builds interference graph and recognize operands participating in move instructions. Mark every node corresponding to a pseudo participating in a move instruction *move-related*.

2. `Simplify`: Modify the interference graph, by removing a node (corresponding to one or more pseudos) of low degree that is not part of any move instruction.

3. `Coalesce`: Do conservative coalescing [BCT94]. Repeat steps 2 and 3 until we get graph where each node has degree higher than the number of available registers or each node is part of a move instruction.

4. `Freeze + potential spill`: If neither step 2 and step 3 can be applied

select a move-related node of low degree and reset the *move-related* mark. Go back to step 2.

5. `Select + actual spill`: Assign colors to nodes in the graph. If some pseudos are spilled then go back to step 1 and see if these spills have changed the colorability of the rest of the graph.

Even though the this algorithm could iterate for a number of times (linear in the number of pseudos), in practice this algorithm iterates very few times and has been found to be fast for an aggressive algorithm.

4.3.4 Usage count based register allocator

Usage count based register allocation proposed by Freiburghouse [Fre74] again assumes a linear representation for the input program like the linear scan algorithm. The main idea in this work is to use the usage count information to decide on which pseudo to spill. The idea is that a pseudo can be spilled if it's usage count is zero.

To do register allocation via usage counts, the model of the program that the allocator must maintain is quite simple: the pseudo to register map at each program point. For each pseudo, at the point of definition, the allocator assigns the max usage count. As the allocator scans the instructions and updates the mapping, it decrements the usage of a pseudo, each time it encounters a reference to the pseudo. Once a pseudo has its usage count reduced to zero, the assigned register is free and can be used by some other pseudo. And if for any particular use of a pseudo, there are not enough free registers then we spill the pseudo with least usage count. For all the instructions following this spill point, that spilled pseudo is considered *unavailable*.

For each spilled pseudo p the spill code generated by the following rule:

- At each instruction i, if there is a use of pseudo p, and p is unavailable at instruction i we reload it before i, into an available register.
- At each instruction i, if pseudo p is defined in i, and p is unavailable at instruction i we use an available register as the target register and then store it back to the location of p.

4.3.5 ILP based register allocator

We use the integer linear program (ILP) based register allocator (RA_i) presented in section 3.2 as an example of ILP based register allocator. The register allocator has its similarities with other ILP based register allocators of Goodwin and Wilken [GW96] and of Appel and George [AG01].

This register allocator takes the benefits of liveness information to reduce the state space and search space together. It also takes into consideration known loads and known stores and tries to see if they can be moved around to get better performance.

4.3.6 SARA

We present in chapter 3 a combine phase for register allocation and stack location allocation. Such a register allocator can be very effective for processors like StrongARM (which have load-multiple/store-multiple instructions to load and store multiple words at a time) and memories like SDRAM (a 64 bit memory and allows efficient access of 64 bits) when present together. We use SARA as one more of our points of reference.

In SARA both register allocator as well as stack location allocation both

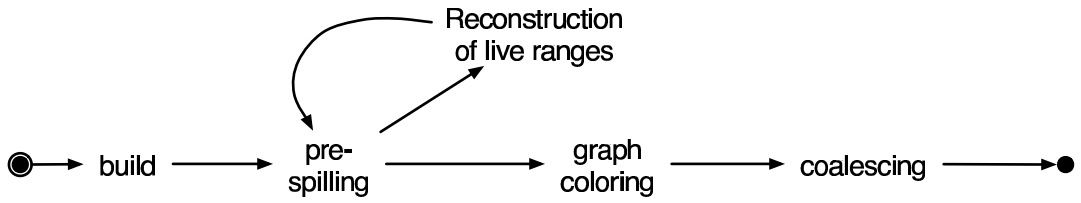

Figure 4.9: Chordal graph based register allocation.

are specified as a single integer-linear-program (ILP), with a single objective function. This combined phase creates a synergy between register assignment, spill code generation and stack location allocation. For a such a phase to be effective, the framework must be able to inform the register allocator about the known-loads/known-stores as well as it the register allocator should be able to communicate back to the framework any load-pairs and store-pairs generated, along with the inversions. Our framework RALF provides all of these, and more.

4.3.7 Register Allocation via Coloring of Chordal Graphs

The chordal graph based allocator [PP05] is an iterative algorithm that has four phases: (1) spilling, (2) coloring, (3) reconstruction of live ranges, (4) coalescing. The algorithm is represented in Fig. 4.9 is an extension of [PP05]. In contrast to the original algorithm which had a linear transition among these phases, here the register allocator makes multiple passes over the phases to generate better spill code. The algorithm works for both chordal and non-chordal interference graphs; however, when the interference graph is chordal, it can find an optimal allocation of registers if spilling does not occurs. They show that the majority of programs under their consideration have chordal interference graphs and hence result in good optimal coloring.

The chordal based approach searches for potential spills before the coloring

phase. If the chromatic number of the (chordal) graph is greater than the quantity of available registers, spilling must be performed. In order to minimize the number of spills, the algorithm attempts to remove nodes that are part of many cliques. (A clique of a graph G is a complete subgraph of G.) If the spilling phase is executed, it is necessary to reconstruct the control flow graph of the target program, and re-execute the spill analysis. The next phase is the coloring of the interference graph. A chordal graph $G = (V, E)$ can be optimally colored in $O(|V| + |E|)$ time. It is possible to prove that after the spilling stage, no further spills will happen in the coloring phase. The last stage of the algorithm is the coalescing of move instructions. Coalescing is performed in a greedy fashion: for each pair of move related registers, the algorithm attempts to assign them the same color.

4.4 Experimental results

In this section we present our experience in using RALF with the register allocation techniques described in section 4.3. We will be using the following abbreviations: (Naive - The naive register allocator, UC - Usage count based register allocation, IRC - Iterated register coalescing, LS - Linear scan, CG - Register allocation by coloring chordal graphs, RA_i - ILP based register allocation, SARA - Combined ILP based stack allocation and register allocation)

For each of the register allocator techniques in Fig. 4.4 presents some statistics to demonstrate the ease of use of the framework. For each of the register allocation scheme we present the number of lines for the register allocation code, number of lines of code required to interface with the framework and a rough estimate on the number of hours to code the interface. We annotate the numbers for the lines of code column by (J) or (A), signifying Java code or AMPL [FGK93] code.

RA	#LOC		Hrs to Code
	RA	Interface	
Naive	196 (J)	773 (J)	< 10
IRC	3538 (J)	773 (J)	< 10
CG	4134 (J)	773 (J)	< 10
UC	402 (J)+	1100 (J)+	< 5
LS	385 (J)+	1100 (J)+	< 5
RA_i	495 (A)	298 (A)	0
SARA	731 (A)	400 (A)	0

Figure 4.10: Experimental evaluation of RALF.

Figure 4.11: Comparison of different register allocators

As we discuss in section 4.5 the framework also provides a grammar for the input interface in javacc format. For LCC and LS we use this grammar and the provided library classes to read the input, generate intermediate data structures and write the output. We annotate with a '+' symbol to designate the use of those library classes. It can be seen the number of hours taken to write the interface code is very minimal. We found in our experience that most of the time the interface code once written could be reused. For example we could use the same interface code for Naive, CG, and IRC. Also we could reuse the interface code written for UC, for LS. For the two ILP based register allocators we present here, we did not have to write any interface code as the language used for the input interface specification is a subset of AMPL.

In Fig. **??** we present a comparative study of the register allocators described in section 4.3. The graph is based on the execution time numbers normalized to

the execution time numbers of the same benchmark programs compiled with the gcc compiler at -O2 optimization level.

As it can be easily guessed, the naive register allocator performs most poorly. However, this is obvious because of the number of loads and stores inserted. Because of the optimal nature of the solution provided by the ILP based register allocator, it tends to outperform the heuristic based solutions. It can be seen that the perrformance of CG (register allocation by coloring chordal graphs) and ICR (Iterated register coalescing) are quite comparable to each other as well as gcc-O2. The register allocator present in the gcc compiler, uses a two phase algorithm for register allocation: (a) aggressive register allocation for local variables within basic blocks, follwed by (b) conservative allocation for the whole function. It can be seen that even without tuning CG and ICR too much, their perofmance can be compared to that of gcc. *What about linear scan and usage count based? tbd*

4.5 Tools for the framework

Along with the framework, we also present the LL(1) grammar for the input interface in javacc format which can be used along with jtb to help in coding. This can be found at: http://compilers.cs.ucla.edu/ralf/input-format.html One can build tools on top of this grammar which can work as library classes for different register allocators. Currently we have implemented library classes to read the input into three address codes, build live intervals, and build interference graphs.

We also present a visualizer for the input data for the register allocator that is output by the framework.

These tools can be found at the RALF homepage:

4.6 Observations and limitations

Our experience with RALF has shown that it is fairly general and easy to use. It has options to handle different extensions to register allocation (coalescing, stack location allocation with and without known-loads and known-stores etc). However, if a certain register allocator chooses to handle some (or all) of these extensions then it can do so. However, if a certain register allocator does not want to handle some (or all) of these extensions then it can ignore these without affecting the correctness of the output generated.

However, the framework does have it's limitations.

- The framework currently handles only integer and sub-integer data types. It does not handle temporaries that are of type float or double.
- The framework does not handle pair registers (and hence pair temporaries). We feel this can be easily extended as it mostly requires changing of the input interface for the register allocator with information regarding the pairing of hardware registers.
- Due to inherent relations between register allocation and the target hardware, our current framework is only set up for ARM targets only. We plan to extend it multiple architectures in future.
- Currently, our framework does not do register coalescing, or SLA phase as a post pass that can be done after register allocation is done to get better code. We plan to add these phases as an optional post pass in future.

- One important future work that remains is to write a checker for the framework that checks the correctness of the register assignment.

4.7 Conclusion

We present here a framework for testing register allocation techniques. We show that the framework is easy to use and at the same time versatile enough that different register allocation schemes can be implemented relatively easily to study the end-to-end numbers.

CHAPTER 5

Conclusion and Future Work

5.1 Conclusion

In this thesis we present an argument showing the importance of good stack location allocation and merging of single loads and single stores into double-loads and double-stores wherever possible. We show that stack location allocation, when done along with register allocation can have a stronger impact. We show our improvement over the publicly available gcc compiler at -O2 level of optimization.

We also present a framework over which new register allocators can be easily implemented and end-to-end numbers obtained to compare against other register allocators. By implementing a variety of register allocators in a very short period of time we show that the framework is versatile and easy to use. Such a framework has many advantages. For example, such a framework gives a good understanding of the overall impact of the register allocator in the compilers in the presence of other optimizations. Such a framework also gives an easy way to compare two different register allocators, in terms of end-to-end numbers, by fixing the rest of the parameters of the compiler.

5.2 Future Work

The work presented in this thesis has a lot of scope for further research. For example the SLA phase as it stands does the space (stack) allocation only for local variables. We believe that it can be extended to global variables. But that would need global analysis. One approach we take in SARA is that, we allow the merger of loads and stores into double-loads and double-stores provided they are accessing neighboring locations.

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.

One direction that needs attention is the possible merging other optimizations that are related to register allocation. For example, researchers have successfully shown that register coalescing and register rematerialization etc give good results when done along with register allocation. It would be interesting to study the behavior of SARA extended with these phases.

Another idea for future work is to use heuristics based solution for both SLA (similar to [Bar92, LDK96]), as well as SARA (extending ideas for heuristic based solutions for register allocation and SLA) is to to find out whether similar performance gains can be obtained with approximate methods that possibly are faster.

Currently, RALF does not give any feedback regarding the correctness of register allocation and stack location allocation done, except for some simple checks. However, enforcing some semantic based checks would be a useful tool for researchers in this area.

References

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. "Efficient Detection of All Pointer and Array Access Errors." In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, December 1994. http:// www. cs. wisc. edu/ austin/ ptr-dist. html.

[AG01] Andrew W. Appel and Lal George. "Optimal Spilling for CISC Machines with Few Registers." In *SIGPLAN'01 Conference on Programming Language Design and Implementation*, pp. 243–253, 2001.

[All88] Victor Allis. "A Knowledge-Based Approach of Connect-Four–The Game Is Solved: White Wins." Technical Report IR–163, Vrije Universiteit Amsterdam, 1988.

[AS99] Rao A and Pande S. "Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs." *In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pp. 128–138, June 1999.

[Bar92] D. Bartley. "Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes." *Software - Practice and Experience*, **22**(2):101–110, February 1992.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. "Improvements to Graph Coloring Register Allocation." *ACM Transactions on Programming Languages and Systems*, **16**(3):428–455, May 1994.

[BEH91] D. Bradlee, S. Eggers, and R. Henry. "Integrating register allocation and instruction scheduling for RISCs." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 122–131, April 1991.

[c9999] "C9X standard." 1999. http:// www. dkuug. dk/ JTC1/ SC22/ open/ n2620.

[CCH96] Ben-Chung Cheng, Daniel A. Connors, and Wen Mei W. Hwu. "Compiler-directed early load-address generation." In *MICRO*, pp. 138–147, 1996.

[Cha82] G. J. Chaitin. "Register allocation and spilling via graph coloring." *SIGPLAN Notices*, **17**(6):98–105, June 1982.

[CK91] D. Callahan and B. Koblenz. "Register allocation via hierarchical graph coloring." In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pp. 192–203, June 1991.

[CPL00] *CPLEX mixed integer solver*. 2000. http:// www. cplex. com.

[EA99] K.M. Elleithy and E.G. Abd-El-Fattah. "A Genetic Algorithm for Register Allocation." In *Ninth Great Lakes Symposium on VLSI*, pp. 226–, 1999.

[FGK93] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL A modeling language for mathematical programming*. Scientific Press, 1993. http:// www. ampl. com.

[FH95] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[Fre74] R. A. Freiburghouse. "Register allocation via usage counts." *Commun. ACM*, **17**(11):638–642, 1974.

[FT87] Michael L. Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms." *J. ACM*, **34**(3):596–615, 1987.

[FW02] Changqing Fu and Kent Wilken. "A faster optimal register allocator." In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 245–256. IEEE Computer Society Press, 2002.

[GA96] Lal George and Andrew W. Appel. "Iterated Register Coalescing." *ACM Transactions on Programming Languages and Systems*, **18**(3):300–324, May 1996.

[GB03] Lal George and Matthias Blume. "Taming the IXP network processor." In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pp. 26–37. ACM Press, 2003.

[GGJ78] 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.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NPCompleteness*. Freeman, 1979.

[GW96] David W. Goodwin and Kent D. Wilken. "Optimal and near-optimal global register allocations using 0-1 integer programming." *Software–Practice & Experience*, **26**(8):929–968, August 1996.

[HP02] John L. Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, third edition, 2002.

[ixpa] "Intel(R) IXP1200 Network Processor." http:// www. intel. com/ design/ network/ products/ npfamily/ ixp1200.htm.

[ixpb] "Intel(R) IXP2400 Network Processor." http:// www. intel. com/ design/ network/ products/ npfamily/ ixp2400.htm.

[KW98] Timothy Kong and Kent D. Wilken. "Precise Register Allocation For Irregular Architectures." In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 297–307. IEEE Computer Society Press, 1998.

[LD98] Rainer Leupers and Fabian David. "A uniform optimization technique for offset assignment problem." In *ISSS*, 1998.

[LDK96] 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.

[LFK99] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. "Evaluation of Algorithms for Local Register Allocation." In *Compiler Construction, 8th International Conference, CC'99*, volume 1575 of *Lecture Notes in Computer Science*. Springer, 1999.

[LGC02] Sorin Lerner, David Grove, and Craig Chambers. "Composing dataflow analyses and transformations." In *Symposium on Principles of Programming Languages*, pp. 270–282, 2002.

[LM96] Rainer Leupers and Peter Marwedel. "Algorithms for address assignment in DSP code generation." In *Proceedings of IEEE International Conference on Computer Aided Design*, 1996.

[LPM97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems." In *IEEE/ACM International Symposium on Microarchitecture(MICRO)*, December 1997.

[LS96] Mikko H. Lipasti and John Paul Shen. "Exceeding the dataflow limit via value prediction." In *International Symposium on Microarchitecture*, pp. 226–237, 1996.

[MBW01] G. Memik, B.Mangione-Smith, and W.Hu. "NetBench: A Benchmarking suite for Network Processors." *IEEE International Conference Computer-Aided Deisgn*, November 2001.

[MPS95] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. "Combining Register Allocation and Instruction Scheduling." Technical Report CS-TN-95-22, 1995.

[NP04] Mayur Naik and Jens Palsberg. "Compiling with code-size constraints." *Trans. on Embedded Computing Sys.*, **3**(1):163–181, 2004.

[np4] IBM PowerNP, http:// www-3. ibm. com/ chips/ techlib/ techlib. nsf/ products/ PowerNP_NP4GS3.

[PDN97] 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.

[PLM01] Jinpyo Park, Je-Hyung Lee, and Soo-Mook Moon. "Register allocation for banked register file." In *Proceedings of Workshop on Languages, Compilers and Tools for Embedded Systems*, pp. 39–47, 2001.

[PP05] Fernando M Q Pereira and Jens Palsberg. "Register Allocation via Coloring of Chordal Graphs." In *The Third Asian Symposium on Programming Languages and Systems*, 2005.

[PS99] Massimiliano Poletto and Vivek Sarkar. "Linear scan register allocation." *ACM Transactions on Programming Languages and Systems*, **21**(5):895–913, 1999.

[RGL96] 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*, pp. 1–11, 1996.

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

[Set73] Ravi Sethi. "Complete Register Allocation Problems." In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pp. 182–195, New York, NY, USA, 1973. ACM Press.

[SKP00] Tammo Spalink, Scott Karlin, and Larry Peterson. "Evaluating Network Processors in IP Forwarding." Technical Report TR–626–00, Princeton University, November 2000.

[SLD97] Ashok Sudarsanam, Stan Liao, and Srinivas Devadas. "Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures." In *Design Automation Conference*, pp. 287–292, 1997.

[SP01] J. Sjödin and C. von Platen. "Storage allocation for embedded processors." In *Proceedings of CASES*, pp. 15–23, 2001.

[sta] "Low-Power, Small-Size, 400MHz, Linux Single Board Computer." http:// www. xbow. com/ Products/ XScale. htm.

[Sto97] Artour Stoutchinin. "An Integer Linear Programming Model of Software Pipelining for the MIPS R8000 Processor." In *Parallel Computing Technologies, 4th International Conference, PaCT-97, Yaroslavl, Russia, September 8-12, 1997, Proceedings*, volume 1277 of *Lecture Notes in Computer Science*. Springer, 1997.

[TA97] Gary S. Tyson and Todd M. Austin. "Improving the accuracy and performance of memory communication through renaming." In *International Symposium on Microarchitecture*, pp. 218–227, 1997.

[TCC00] Marc Tremblay, Jeffrey Chan, Shailender Chaudhry, Andrew W. Conigliaro, and Shing Sheung Tse. "The MAJC Architecture: A Synthesis of Parallelism and Scalability." *IEEE Micro*, **20**(6):12–25, 2000.

[TG03] Sriraman Tallam and Rajiv Gupta. "Bitwidth aware global register allocation." In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 85–96, 2003.

[WL01] Jens Wagner and Rainer Leupers. "C Compiler Design for an Industrial Network Processor." In *LCTES/OM*, pp. 155–164, 2001.