SARA: Combining Stack Allocation and Register Allocation

V. Krishna Nandivada and Jens Palsberg

UCLA
University of California, Los Angeles

Abstract. Commonly-used memory units enable a processor to load and store multiple registers in one instruction. We showed in 2003 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 paper 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.

1 Introduction

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 [1], Stargate (http://www.xbow.com/Products/XScale.htm), Sun MAJC 5200 [24], etc. Multiple loads and stores are particularly useful in connection with register allocation where spill code may need to save and restore multiple registers.

For example, on the StrongARM, the register size is 32 bits and each basic load/store operation (called LDR/STR) operates on one register at a time. However, the SDRAM has a 64 bit bus so if we are using an LDR instruction to load a 32 bit register, we are wasting half of the bandwidth of the bus. Fortunately, we can use a load/store-multiple operation (we refer to them as LDM/STM) to operate on two registers at a time, thereby taking full advantage of the bus and saving one full LDR/STR instruction (40/50 cycles) [23].

To replace two LDR instructions with one LDM instruction we need the addresses to be contiguous and the destination registers to be different. To replace

LDR $addr_1$ r_i	by	MOV r $addr_1$
LDR $addr_2$ r_j		LDM [r] {r_i,r_j}

A. Mycroft and A. Zeller (Eds.): CC 2006, LNCS 3923, pp. 232–246, 2006.

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

(a)

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

(b)

```
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
```

(c)

```
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
```

(d)

Fig. 1. (a) Fragment of C code, (b) Mappings of pseudos to registers and stack locations, (c) code generated by gcc, (d) code generated by SARA

we would need $i \neq j$ and the two base addresses $addr_1$ and $addr_2$ must be contiguous at 4 byte boundaries: $addr_2 - addr_1 = 4$.

We showed in 2003 [20] 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 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 1(b) and generates the code shown in Figure 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 job using register r3 both when loading `a` and when loading `b`. 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 `b`. 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 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 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 paper 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 [13, 14, 16, 3, 11, 19], (2) SLA can be specified by an ILP [20], 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 [8, 7, 6].

While solving ILPs can be slow, we note that all of the following three problems are NP-complete: (a) register assignment [22], (b) spill code generation [12], and (c) SLA [20]. 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 reconfirm our results in [20] 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. [5] and Motwani et al. [18] demonstrated how to combine register allocation and code scheduling to obtain faster code. Lerner et al. [15] 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 the state-explosion problem, and in Section 5 we present our experimental results.

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 [13] and of Appel and George [3]. 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.

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

$$\begin{array}{llll}
\text{Insts} & \subseteq \{1..\text{nInsts}\} & \text{Req} & : \text{Insts} \times \text{Pseudos} \rightarrow \{0,1\} \\
\text{Pseudos} & \subseteq \{1..\text{nPseudos}\} & \text{Def} & : \text{Insts} \times \text{Pseudos} \rightarrow \{0,1\} \\
\text{Regs} & \subseteq \{1..\text{nRegs}\} & \text{prevInst} & : \text{Insts} \rightarrow \text{Insts} \cup \{null\} \\
\text{Loc} & \subseteq \{1..\text{nPseudos}\} & \text{joinInst} & : \text{Insts} \times \text{Insts} \rightarrow \text{Insts} \cup \{null\} \\
 & & \text{callInst} & : \text{Insts} \rightarrow \{0,1\}
\end{array}$$

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 1, Insts = {1,2,3,4}, Pseudos = {a,b,c,d}, Regs = {1,2,3,4,5,6,7,8,9,10}. The parameter $\mathrm{Req}(i,p)$ is set to 1 if instruction i requires pseudo p and hence needs p to be present in a register. The parameter $\mathrm{Def}(i,p)$ is set to 1 if instruction i sets pseudo p. The control flow of the program is given by three parameter maps. The parameter $\mathrm{prevInst}(i)$ is a singleton set containing the previous instruction of i if it has only one previous instruction, and null otherwise. The parameter $\mathrm{joinInst}(i)$ is the set of previous instructions of i if instruction i is a join point with multiple previous instructions, and null otherwise. The parameter $\mathrm{callInst}(i)$ has value 1 if the instruction i is a call instruction, and 0 otherwise.

For each instruction i, the parameter $\mathrm{freq}(i)$ returns the frequency of execution of that instruction. In this paper, we use static estimates of $\mathrm{freq}(i)$; 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.

Constraint Generation. From the input program we generate an ILP whose main purpose is to ensure the following properties: (1) at any instruction, each pseudo is assigned at most one register, (2) at any instruction, each register is assigned at most one pseudo, (3) at any instruction, the number of used registers is bounded 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 its 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.

$$\begin{array}{ll} \text{PsR} & : \text{Insts} \times \text{Pseudos} \times \text{Regs} \rightarrow \{0,1\} \\ \text{xDef} & : \text{Insts} \times \text{Pseudos} \times \text{Regs} \rightarrow \{0,1\} \\ \text{spStore} & : \text{Insts} \times \text{Pseudos} \times \text{Regs} \rightarrow \{0,1\} \\ \text{spLoad} & : \text{Insts} \times \text{Pseudos} \times \text{Regs} \rightarrow \{0,1\} \\ \text{inUse} & : \text{Regs} \rightarrow \{0,1\} \end{array}$$

$\text{PsR}(i,p,r)$ 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:

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

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

The second of the two constraints above implies that at any program point the number of pseudos that are available in registers is bounded by the number of registers available.

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 its 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:

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

A pseudo mapped to a caller save register loses its mapping after a call:

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

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

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

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

$$\text{SpillCost} = \sum_{i \in \text{Insts}} \text{freq}(i) \times \sum_{p \in \text{Pseudos},\ r \in \text{Regs}} \begin{pmatrix} (\text{spLoad}(i,p,r) \times \text{loadCost}) \\ + \\ (\text{spStore}(i,p,r) \times \text{storeCost}) \end{pmatrix}$$
$$\text{CalleeSaveCost} = \frac{1}{2} \times \sum_{r \in \text{Regs} - \text{callerSaveRegs}} \text{inUse}(r) \times (\text{loadCost} + \text{storeCost})$$

The callee save registers are loaded and stored using load/store-multiple instructions, hence the cost is reduced by a factor of two.

Constraint Solving. We use AMPL [9] to generate the ILP, and CPLEX (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 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}
SpillCost	= 2 × loadCost + 2 × storeCost = 184
CalleeSaveCost	= 0

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 2. We now present the additional parameters, variables and constraints required for SARA over RA. The new phase SARA requires three additional variables:

$$\begin{array}{ll} \text{loadPair} & : \text{Insts} \times \text{Pseudos} \times \text{Pseudos} \rightarrow \{0,1\} \\ \text{storePair} & : \text{Insts} \times \text{Pseudos} \times \text{Pseudos} \rightarrow \{0,1\} \\ f & : \text{Pseudos} \times \text{Loc} \rightarrow \{0,1\} \end{array}$$

For a given instruction i, and two pseudos p_1 and p_2 ($p_1 \neq p_2$), the map $\text{loadPair}(i, p_1, p_2)$ returns 1 if we can replace the two spill loads by a pair, and 0 otherwise. The map f maps a pseudo to its location: $f(p, l)$ returns 1 if pseudo p is placed in location l. Note that 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.
$\forall i \in \text{Insts},\ \forall p_1, p_2 \in \text{Pseudos}$:

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

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

LDM and STM require that the memory locations are consecutive.
$\forall i \in \text{Insts},\ \forall p_1, p_2 \in \text{localPseudos}$:

$$\begin{array}{c} \text{diff}(p_1,p_2) \neq 1 \Rightarrow \text{loadPair}(i,p_1,p_2) = 0 \\ \text{diff}(p_1,p_2) \neq 1 \Rightarrow \text{storePair}(i,p_1,p_2) = 0 \\ \text{diff}(p_1,p_2) = ((\sum_{l \in \text{Loc}} l \times f(p_1,l)) - (\sum_{l \in \text{Loc}} l \times f(p_2,l))) \end{array}$$

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.

Objective function. The objective function used in SARA is similar to the one used by our ILP-based RA given in section 2. The new twist is that SpillCost takes pairs into account.

SpillCost =

$$\sum_{i \in insts} \text{freq}(i) \times \left(\begin{array}{l} \sum_{p \in \text{Pseudos}, r \in \text{Regs}} \text{spLoad}(i,p,r) \times \text{loadCost} \quad - \\ \sum_{p_1,p_2 \in \text{Pseudos}} (\text{loadPair}(i,p_1,p_2) \times \text{loadPairSave}) \\ \\ \sum_{p \in \text{Pseudos}, r \in \text{Regs}} \text{spStore}(i,p,r) \times \text{storeCost} \quad - \\ \sum_{p_1,p_2 \in \text{Pseudos}} (\text{storePair}(i,p_1,p_2) \times \text{storePairSave}) \end{array} \right)$$

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 ($2 \times$ loadCost $-$ loadPairCost). Similarly storePairSave is calculated as ($2 \times$ storeCost$-$storePairCost). In the model generated by the compiler loadPairCost and storePairCost are given as parameters.

The result of solving the above constraints for the running example shown in Figure 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 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,r2}
SpillCost	= loadPairCost + storePairCost = 94
CalleeSaveCost	= 0

Our implementation of SARA uses a superset of the constraints presented in this paper. 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), (2) non-spill memory instructions (generated in the presence of pointer based accesses in the code), and (3) inversions [20]. 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.

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.

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:

$$i_1 : x = y + p \ ; \ // \ p \ \textit{dies after} \ i_1$$
$$i_2 : y = y + z \ ;$$

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:

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

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. The introduction of dummy instructions also overcomes the need to introduce dummy basic blocks as additional place holders for spill code. Our notion of dummy instructions is related to the notion of points between instructions that was used by Appel and George [3]. 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.

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:

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

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 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 instructions 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: 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 its 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, 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 paper 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).

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. The impact of SARA may be different for systems with cache. We have drawn our benchmark programs from a variety of sources:

- Stanford Benchmark suite: The first four benchmarks are small and simple, but typical of the subroutines of many other benchmarks.
- NetBench: Route and url are network related benchmarks from the NetBench [17] 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 [4]. Yacr2 is an implementation of a channel router and Ft is an implementation of a minimum spanning tree algorithm [10].
- 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 [2] 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 2. The static characteristics we present here include the number of lines of C code, the number of instructions seen by the ILP solver

Benchmark	LoC	#RTLs	#Funcs	gcc+SLA			RA+SLA			SARA		
				Mem	Pair	CSR	Mem	Pair	CSR	Mem	Pair	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

Fig. 2. Benchmark characteristics and compile time statistics

(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 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 (Pair) inserted, and the number of callee save registers (CSR) used.

All these benchmarks have the common characteristic that they are non-floating point benchmarks. (We had to edit a 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 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 nonoptimal 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 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.

To get an overall comparison of the different register allocators, we present the normalized execution time numbers in Figure 4. Our experience can be represented in a lattice as shown in Figure 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-based register allocation is giving better results than gcc's default module run

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

Fig. 3. Execution time numbers

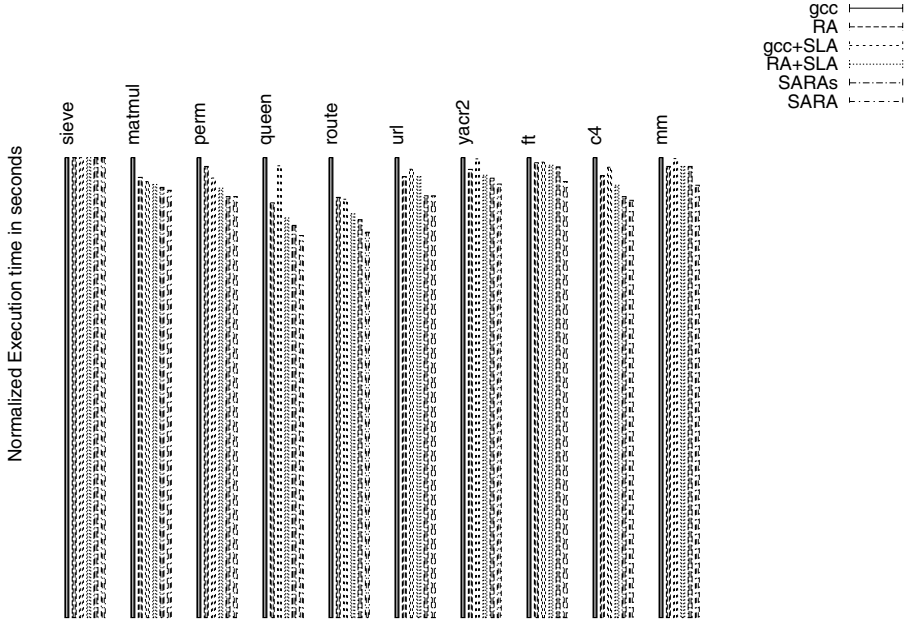

Fig. 4. Normalized execution times

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

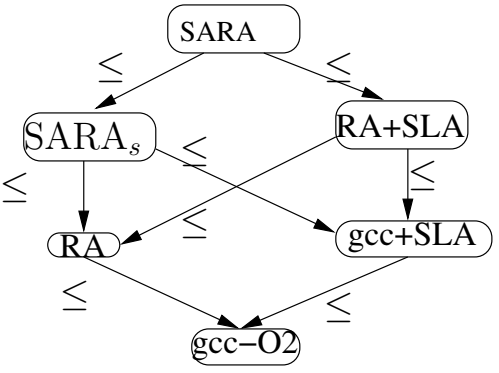

Fig. 5. A comparison of different register allocator schemes

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.

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.

Acknowledgments. We thank the anonymous reviewers for helpful comments on draft of the paper. We were supported by Intel and by a National Science Foundation ITR Award number 0401691.

References

1. Intel(r) IXP2400 network processor. http://www.intel.com/design/network/products/npfamily/ixp2400.htm.
2. Victor Allis. A knowledge-based approach of connect-four–the game is solved: White wins. Technical Report IR–163, Vrije Universiteit Amsterdam, 1988.
3. Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In PLDI 2001, pages 243–253.
4. Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In PLDI 1994, pages 290–301.

5. D. Bradlee, S. Eggers, and R. Henry. Integrating register allocation and instruction scheduling for riscs. In ASPLOS 1991, pages 122–131.
6. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. ACM TOPLAS 16(3):428–455, May 1994.
7. D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In PLDI 1991, pages 192–203.
8. G. J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982.
9. Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL A modeling language for mathematical programming.* Scientific Press, 1993.
10. Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
11. Changqing Fu and Kent Wilken. A faster optimal register allocator. In Proceedings of ACM/IEEE MICRO 2002, pages 245–256.
12. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NPCompleteness.* Freeman, 1979.
13. 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.
14. Timothy Kong and Kent D. Wilken. Precise register allocation for irregular architectures. In Proceedings of ACM/IEEE MICRO 1998, pages 297–307.
15. Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In POPL 2002, pages 270–282.
16. Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. Evaluation of algorithms for local register allocation. In CC 1999, pages 137–152, LNCS 1575.
17. G. Memik, B.Mangione-Smith, and W.Hu. Netbench: A benchmarking suite for network processors. In IEEE ICCAD 2001.
18. Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining register allocation and instruction scheduling. Tech. Report CS-TN-95-22, 1995.
19. Mayur Naik and Jens Palsberg. Compiling with code-size constraints. *ACM Transactions on Embedded Computing Systems*, 3(1):163–181, 2004.
20. V. Krishna Nandivada and Jens Palsberg. Efficient spill code for SDRAM. In CASES 2003, pages 24–31.
21. R. Rivest. The md5 message-digest algorithm. *Request for Comment: 1321*, 1992.
22. Ravi Sethi. Complete register allocation problems. In ACM STOC 1973, pages 182–195.
23. Tammo Spalink, Scott Karlin, and Larry Peterson. Evaluating network processors in ip forwarding. Technical Report TR–626–00, Princeton University, 2000.
24. 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.