A Framework for End-to-End Evaluation of Register Allocators

V. Krishna Nandivada Fernando Pereira Jens Palsberg
UCLA
{nvk,fernando,palsberg}@cs.ucla.edu

Abstract

We present RALF, a framework for end-to-end evaluation of register allocators. Built on top of gcc, RALF enables evaluation and comparison of register allocators in the setting of an industrial-strength compiler. RALF supports modular plug-and-play of register allocators without modifying the compiler implementation at all. RALF provides any plugged-in register allocator with an intermediate program representation that is independent of the data structures of the framework. In return, the register allocator provides RALF with a set of register allocation directives. The contract between RALF and a register allocator is given by requirements on the intermediate program representation and the register allocation directives. RALF checks that the produced directives satisfy the requirements, thereby helping with finding bugs in a register allocator. We demonstrate the versatility of RALF by presenting our experiments with six different register allocators.

1. Introduction

Register allocation [ASU86] plays a pivotal role in compilation of high level programs to machine code. A register allocator can be a significant part of the compiler implementation (10% for lcc [FH95] and 12% for gcc 2.95.2). A register allocator and various optimization phases have a considerable and complicated impact on each other. The mutual impact makes it difficult to give a realistic evaluation of a register allocator. The best form of evaluation of a register allocator would be in the setting of an industrial-strength compiler. Such an implementation would enable a researcher to measure the performance of realistic, executable code generated with the help of the register allocator. Performance numbers provide a more accurate evaluation of a register allocator than static counts of, for example, the number of registers used and the number of spill instructions inserted. Ideally, a researcher can plug a register allocator into an existing industrial-strength compiler without modifying the compiler implementation at all.

Existing public domain compilers such as GCC [gcc05], SMLNJ [sml00], and Tiger [AP02], as well as compiler frameworks such SUIF [HAA+96] and SOOT [VRCG+99] allow a programmer to implement a new register allocator. Their main drawback is that the programmer of a register allocator has to understand and work with the data structures of the underlying compiler. The data structures that interface a register allocator with the rest of the compiler can be quite complicated. Tabatabai et al. [ATGL96] have presented a register allocation framework in cmcc (CMU C compiler). Their framework presents different modules (e.g. graph construction, coalescing, color ordering, color assignment, spill code insertion, and others) that different register allocation techniques might need. The key idea is that by providing these commonly used methods as libraries, different register allocation schemes can be easily coded and a lot of code can be reused. However, in case a register allocator needs other mechanisms than those provided by the framework, the programmer must understand and work with the data structures of the underlying compiler. In summary, the existing frameworks tend to require a programmer of a register allocator to understand and modify the underlying compiler.

In this paper we present RALF, a framework for end-to-end evaluation of register allocators. Built on top of gcc, RALF enables evaluation and comparison of register allocators in the setting of an industrial-strength compiler. The three main design criteria for RALF are modularity, reliability, and versatility.

Modularity. RALF supports modular plug-and-play of register allocators without modifying the compiler implementation at all. In effect, RALF allows a programmer to replace the register allocator of gcc without modifying the gcc implementation itself. RALF provides any plugged-in register allocator with an intermediate program representation. In return, the register allocator provides RALF with a set of register allocation directives. Both the intermediate program representation, called MIRA (Mathematical Intermediate representation for Register Allocation), and the register allocation directives, called FORD (FOrmat for Register allocation Directives), are independent of the underlying data structures of the framework and of the plugged-in register allocator.

Reliability. Finding bugs in a register allocator can be difficult. Bugs may materialize while running the register allocator, while running the code generator, while assembling the code, or even while running the target code. RALF supports early bug finding by checking that the intermediate program representation and the register allocation directives satisfy a list of requirements. Those requirements form the contract between RALF and a register allocator. When those requirements are met, various problems cannot occur later during code generation, assembly, or at run time. We report on the time to implement various register allocators and we argue that the development time is fairly low in part because of RALF's simpler input output interface and good support for bug finding.

Versatility. We have designed MIRA and FORD to support a wide variety of register allocators. The FORD directives include support for such things as register assignment to pseudo registers, spill/reload code, coalescing, stack location allocation [NP03], pairing of loads and stores, and insertion of new instructions (such as move operation and bitwise operations). We have experimented with six different register allocators in RALF, including iterated register coalescing, linear scan register allocation, and an ILP-based register allocator. Our experiments give a fair comparison of the register allocators because *everything else remains constant*.

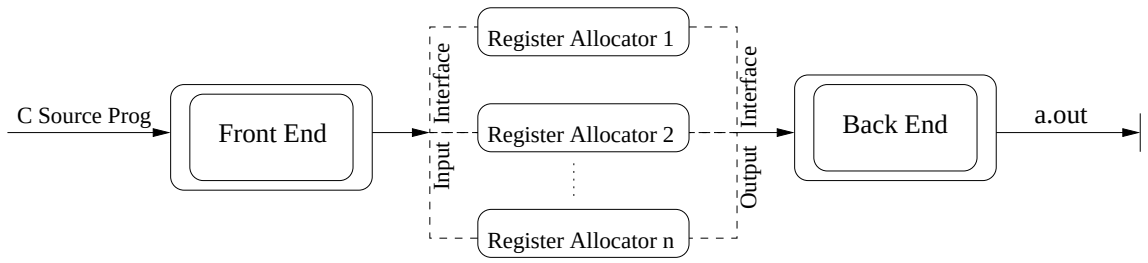

Figure 1. Block diagram of RALF.

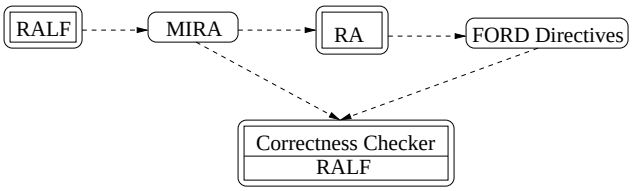

Figure 2. Interaction of a plugged in register allocator with RALF.

In the following section we describe the MIRA intermediate representation and the FORD register allocation directives. In Section 3 we present the requirements on MIRA and FORD, in Section 4 we present our experience with six register allocators, and in Section 5 we conclude.

2. Framework Description

We present a high-level block diagram for our register allocation framework RALF in Figure 1. RALF has two main parts, namely a front end and a back end. The front end consists mainly of gcc's front end, gcc's optimization phases, and code for producing a MIRA program representation. The back end consists mainly of a correctness checker, an implementation of the register allocation directives, and gcc's code generation phase. Given a register allocator, RALF first runs the front end, then the register allocator, and then the back end. Figure 2 shows the interaction of RALF and any plugged-in register allocator. Given any C program P, RALF translates P into a MIRA program M, which is then fed to the plugged-in register allocator. The register allocator outputs a set of FORD directives D, which are then fed back to RALF. RALF checks that (M, D) satisfies a list of requirements, applies the directives D to M, and generates machine code. Along with applying the directives, RALF also does some mundane tasks, like inserting loads and stores of callee save registers at the entrance and exit of each function, thus relieving the register allocator from those activities. Next we describe MIRA and FORD.

2.1 Input Interface - MIRA

The input to the register allocator is a MIRA program which contains program-specific information and architecture-specific information. MIRA hides most of the compiler specific details from the programmer, while including a plethora of information that is required by different register allocators. The required information includes use-def information, liveness information, pre-colored pseudos (pseudos that already have machine registers assigned), known loads and stores (loads and stores already present in the input program), move instructions, etc. A MIRA program consists of sets and parameters written in AMPL syntax [FGK93]. A set is a symbolic enumeration and a parameter can be a scalar value or a collection of values indexed by one or more sets.

Program-specific information. The program-specific information is given using the following definitions.

Insts	$\subseteq$	$\{1..\text{nInsts}\}$		
Pseudos	$\subseteq$	$\{1..\text{nPseudos}\}$		
Loc	$\subseteq$	$\{1..\text{nPseudos}\}$		
lsmPseudos	$\subseteq$	Pseudos	$\rightarrow$	$\{0,1\}$
memPseudos	$\subseteq$	Pseudos	$\rightarrow$	$\{0,1\}$
Req	$\subseteq$	Insts $\times$ Pseudos	$\rightarrow$	$\{0,1\}$
Def	$\subseteq$	Insts $\times$ Pseudos	$\rightarrow$	$\{0,1\}$
prevInst	$\subseteq$	Insts $\times$ Insts	$\rightarrow$	{0,1}
joinInst	$\subseteq$	Insts $\times$ Insts	$\rightarrow$	Insts
callInst	$\subseteq$	Insts	$\rightarrow$	$\{0,1\}$
jumpInst	$\subseteq$	Insts	$\rightarrow$	$\{0,1\}$
moveInst	$\subseteq$	Insts	$\rightarrow$	$\{0,1\}$
Freq	$\subseteq$	Insts	$\rightarrow$	N
Live	$\subseteq$	Insts $\times$ Pseudos	$\rightarrow$	$\{0,1\}$
liveHardReg	$\subseteq$	Insts $\times$ Regs	$\rightarrow$	$\{0,1\}$

where nInsts is the maximum number of instructions, nPseudos is the maximum number pseudos present in the program, and N is the set of natural numbers.

The sets of instructions, pseudos, and locations for the pseudos are given by Insts, Pseudos, and Loc respectively. Before register allocation is done, compiler represents pseudos in two forms: as temporaries and as memory locations. Accordingly, there can be two types of pseudos in a MIRA program: (a) Pseudos corresponding to the scalars that come from the compiler used temporaries (represented as lsmPseudos), (b) Pseudos corresponding to memory locations (represented as memPseudos). Parameter memPseudos(p) is set to 1, if pseudo p is a memory pseudo. Parameter lsmPseudos(p) is set to 1, if pseudo p is not a memory pseudo. Accesses of these memPseudos results in memory accesses. A register allocator may decide to move these instructions to reduce total execution time. (e.g. out of a loop etc). These pseudos can also be used by the stack location allocation phase.

Each instruction is considered to have a (possibly empty) set of required pseudos and can set a pseudo or a register. 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 0 otherwise. And Def(i, p) is set to 1, if instruction i sets pseudo p, and 0 otherwise.

The control flow of the program is given by two maps prevInst and joinInst. Parameter prevInst(i_p, i_1) is set to 1, if instruction i_1 has exactly one previous instruction i_p and 0 otherwise. Given two connected basic blocks ($b_p \rightarrow b_1$), if i_1 is the first instruction of b_1 and i_p is the last instruction of b_p, then parameter joinInst(i_p, i_1) is the set to 1.

A subset of instructions are declared as function calls and another subset as jump (conditional or unconditional) instructions. Parameter callInst(i) has value 1 if instruction i is a call instruction, and 0 otherwise. Parameter jumpInst(i) has value 1 if instruction i is a jump instruction, and 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. The moveInst information can be used by register allocators doing coalescing.

For each instruction i, parameter Freq(i) returns the frequency of execution of that instruction. RALF uses static estimates of the frequencies of each instruction. In future work one might use profiling-based estimates of the frequencies.

At each instruction the liveness information is given for each pseudo and machine registers. Problem parameter Live(i, p) is set to 1, if pseudo p is live at instruction i. And parameter liveHardReg(i, r) is set to 1, if machine register r is live at instruction i. The liveness information can be computed from the rest of the program but is provided by RALF to take that burden away from the register allocators. RALF can provide weaker liveness results, if so desired. (Some register allocators [NP05] want weaker liveness information to get more flexibility to do code motion.) The quality of the liveness information is controlled by an environment variable.

Architecture specific information. Architecture specific information chiefly deals with information about different registers and costs of different operations. The architecture-specific information is given using the following definitions.

Regs	$\subseteq$	$\{1..\text{nRegs}\}$
callerSaveRegs	$\subseteq$	Regs $\rightarrow \{0, 1\}$
loadCost	$\in$	N
storeCost	$\in$	N
storePairCost	$\in$	N
loadPairCost	$\in$	N
invStoreCost	$\in$	N
invLoadCost	$\in$	N

where nRegs is total number of machine registers available to a register allocator.

The parameter Regs represents the set of available registers. A subset of machine registers are designated as caller save registers and are represented by callerSaveRegs. The contents of caller save registers are not saved across calls. Each function must save and restore any register that is not a caller save register (that is, those other registers are callee save registers).

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. The cost of inversion caused by a store-pair and load-pair instruction is given by invStoreCost and invLoadCost respectively.

We omit the description of some parameters that give information about pre-colored registers, known-loads (load instructions that are already present), known-stores (store instructions that are already present),

Input interface requirements. RALF produces MIRA programs which satisfy the following requirements.

- *Type correct:* We check the type correctness of each parameter definition in the MIRA program. For example, for callInst we check that the listed instructions are indeed in the interval specified by Insts.
- *Three address codes:* We check that the MIRA program is similar to three-address code: Req allows at most two pseudos to be used in any instruction, and Def allows at most one pseudo or a machine register to be defined in any instruction.
- *Liveness:* The liveness information in the output is conservative, that is, if a pseudo is live in the MIRA program, then the liveness information given in the Live section of the MIRA program will reflect that.

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

Figure 3. Sample input program, using two pseudos.

```
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: jumpInst:= ;
param: callInst:= ;
param nRegs := 3;
param nInsts := 3 ;
param nPseudos := 3;
param loadCost := 41;
param loadPairCost := 42;
param storeCost := 51;
param storePairCost := 52;
```

Figure 4. Sample MIRA program.

We enforce these requirements to ensure a simple and convenient program model for any plugged-in register allocator.

Example MIRA program. We show in Figure 4 a sample of the generated MIRA code for the C code shown in Figure 3. In Figure 4 the program has three instructions and two pseudos, and the machine has three registers. For readability we use i1, i2, ... as names of the elements of Insts, we use p1, p2, ... as names of the elements of Pseudos, and we use r0, r1, ... as names of the elements of Regs; in RALF the elements of such sets are all represented as numbers. Parameters are specified by indexing over these sets and hold integer values. The framework outputs parameters and value pairs for nonzero values only.

Figure 4 specifies that 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. The `prevInst` parameter gives the control flow of the program: i1 precedes i2, and i2 precedes i3. 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, store cost, and cost of a store-pair.

2.2 Output Interface - FORD

For a register allocator framework to be generally applicable it must be flexible enough to understand different types of 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.

RALF provides a simple format FORD that a plugged-in register allocator can use to encode different register allocation direc-

tives. RALF uses the directives to generate executable code. FORD is organized by sections; it 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.

PsR	$\subseteq$	Insts $\times$ Pseudos $\times$ Regs
xDef	$\subseteq$	Insts $\times$ Pseudos $\times$ Regs
f	$\subseteq$	Pseudos $\times$ Pseudos $\cup \{-1\}$
spLoad	$\subseteq$	Insts $\times$ Pseudos $\times$ Regs
loadPair	$\subseteq$	Insts $\times$ Pseudos $\times$ Pseudos
inverseLoad	$\subseteq$	Insts $\times \{0, 1\}$
spStore	$\subseteq$	Insts $\times$ Pseudos $\times$ Regs
storePair	$\subseteq$	Insts $\times$ Pseudos $\times$ Pseudos
inverseStore	$\subseteq$	Insts $\times \{0, 1\}$
moveInst	$\subseteq$	Insts $\times$ Regs $\times$ Regs

To minimize the communication overhead between the framework and the register allocator, RALF requires that the register allocator outputs just the nonzero entries for each tuple in each of the sections, wherever applicable.

The ten sections in FORD can be classified into three categories depending on the type of the directives: register assignment information, spill code information, and new instructions.

Register assignment information. The `PsR` section gives the pseudo to register map at each instruction. 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 that is 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 code information. For each pseudo, section `f` gives the assigned stack location number and -1 if the pseudo does not have any assigned location. Each tuple is of the form (p, l), signifying pseudo p gets location l.

Pseudo reload information is given in 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 specified 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, an entry in the `inverseLoad` section gives information about *inversion* [NP03]. 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 in 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 specified 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, an entry in the `inverseStore` section provides information about *inversion*. 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 register allocators. Such a feature is helpful in bitwidth-aware register allocation schemes such as the one presented by Tallam and Gupta [TG03]). RALF supports such capabilities but we have omitted them from this paper.

```
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 5. Sample FORD directives.

Example output interface. Fig 5 presents a sample output from a register allocator for the sample code shown in Fig 3. One register is enough to do the register allocation in this program. The register used is the caller-save register `r0`, and hence the allocator need not save or restore callee save registers. Since both the pseudos have been placed in registers, they do not need a place in the stack (and hence the negative values in the `f` section).

3. Safety Checks

A compiler framework that allows plugging of register allocators can help with bug finding in several ways:

1. No checks are performed by the framework and instead errors may be caught by an assembler or spotted during execution of the target code.
2. The framework can throw an *exception* during the post-register allocation phase when it encounters an inconsistency.
3. The framework performs safety checks before applying the register allocation directives.

The drawback of the first two options is that it is not easy to pin point the exact nature of error. The third option is eager and aggressively checks for consistency and hence can provide better debugging support for the register allocator.

RALF has a safety checker which enforces both syntactic and semantic constraints on a MIRA program and a FORD file. The goal of the checker is to ensure that the plugged-in register allocator preserves the syntax and semantics of the input program. Non-compliance with these checks might lead to a situation in which the framework cannot generate machine code or generates incorrect machine code.

3.1 Syntactic constraints

Syntactic constraints are simple checks to ensure that every entry output by the register allocator is valid. RALF checks that every instruction, pseudo, register and location in the FORD directives are from the sets Insts, Pseudos, Regs, and Loc, respectively.

3.2 Semantic constraints

Semantic constraints are checks to enforce the underlying semantics of register allocation. RALF enforces the following requirements. Absence of such an enforcement can have varying implications: Assembler error, incorrect program behavior during execution, failure of the framework to generate any code. We categorize these constraints based on the target phase.

Assembler error

Compliance of these checks are required for the assembler to parse the machine code generated by RALF.

Defined register. Every set instruction (declared using the Def map) must have a target register.

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

For example, in the directives specified in Figure 5 for the MIRA program given in Figure 4, it would have been erroneous if $\forall$ r $\in$ Regs, (i1,p1,r) $\notin$ xDef.

Used 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.

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

For example, in the directives specified in Figure 5 for the MIRA program given in Figure 4, it would have been erroneous if $\forall$ r $\in$ Regs, (i2,p1,r) $\notin$ PsR.

Incorrect program behavior

Incorrect register allocation may lead to generation of semantically incorrect machine code. Such errors lead to undefined behavior (incorrect output, segmentation faults etc) during the execution of generated binary.

Consistent pseudos A pseudo can be mapped to at most one register at any time.

$$\forall i \in \text{Insts} : \forall p \in \text{Pseudos} : \forall r_1 \in \text{Regs} : \forall r_2 \in \text{Regs} : \\ r_1 \neq r_2 \Rightarrow ((i,p,r_1) \in \text{PsR} \Rightarrow (i,p,r_2) \notin \text{PsR})$$

For example, in the directives specified in Figure 5 for the MIRA program given in Figure 4, it would have been erroneous if {(i2,p1,r0),(i2,p1,r1)} $\subseteq$ PsR.

Conflicting registers A register can be bound to at most one pseudo at any time.

$$\forall i \in \text{Insts} : \forall p_1 \in \text{Pseudos} : \forall p_2 \in \text{Pseudos} : \\ \forall r \in \text{Regs} : p_1 \neq p_2 \Rightarrow ((i,p_1,r) \in \text{PsR} \Rightarrow (i,p_2,r) \notin \text{PsR})$$

For example, in the directives specified in Figure 5 for the MIRA program given in Figure 4, it would have been erroneous if {(i2,p1,r0),(i2,p2,r0)} $\subseteq$ PsR.

Live pseudo. A pseudo must be alive to be mapped to a register.

$$\forall i \in \text{Insts} : \forall p \in \text{Pseudos} : \forall r \in \text{Regs} : \\ (i,p,r) \in \text{PsR} \Rightarrow \text{Live}(i,p) = 1$$

For example, in the directives specified in Figure 5 for the MIRA program given in Figure 4, it would have been erroneous if $\exists$ r $\in$ Regs, (i1,p1,r) $\in$ PsR.

Reload. Every reload requires that after the reload, the pseudo be available in the destination register of the reload instruction.

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

For example, in the directives specified in Figure 5 for the MIRA program given in Figure 4, it would have been erroneous if $\exists$ r $\in$ Regs, (i1,p1,r) $\in$ spLoad.

Spill-store. Every spill-store requires that the pseudo be available in the source register of the spill store before the location of that spill-store.

$$\forall i \in \text{Insts} : \forall p \in \text{Pseudos} : \forall r \in \text{Regs} : \\ (i,p,r) \in \text{spStore} \Rightarrow (i,p,r) \in (\text{PsR} \cup \text{xDef})$$

For example, in the directives specified in Figure 5 for the MIRA program given in Figure 4, it would have been erroneous if $\exists$ r $\in$ Regs, (i1,p2,r) $\in$ spStore.

Stack location. Every pseudo that is loaded or stored must have a stack location.

$$\forall i \in \text{Insts} : \forall p \in \text{Pseudos} : \forall r \in \text{Regs} : \\ (i,p,r) \in (\text{spStore} \cup \text{spLoad}) \Rightarrow (p,-1) \notin f$$

Neighbors. 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.

$$\forall i \in \text{Insts} : \forall p_1,p_2 \in \text{Pseudos} : \forall p_i,p_j \in \text{Loc} : \\ ((i,p_1,p_2) \in \text{storePair} \wedge (p_1,p_i),(p_2,p_j) \in f) \Rightarrow \\ \text{abs}(j-i) = 1$$

$$\forall i \in \text{Insts} : \forall p_1,p_2 \in \text{Pseudos} : \forall p_i,p_j \in \text{Loc} : \\ ((i,p_1,p_2) \in \text{loadPair} \wedge (p_1,p_i),(p_2,p_j) \in f) \Rightarrow \\ \text{abs}(j-i) = 1$$

The function abs returns the absolute value of the argument.

Spill code after jump instructions. If a pseudo is stored after a (conditional) jump instruction, it's semantics can be misleading, because such spill code will be executed in only one of the branches (taken or follow through). And hence RALF issues a warning here.

$$\text{[Warning]} \\ \forall i \in \text{Insts} : \forall p \in \text{Pseudos} : \forall r \in \text{Regs} : \\ \text{jumpInst}(i) \Rightarrow (i,p,r) \notin \text{spStore}$$

Problems in code generation

These are the errors that halt the framework, because such directives cannot be processed.

Double-load. A double-load instruction before any instruction i requires that there are two load instructions before i.

$$\forall i \in \text{Insts} : \forall p_1,p_2 \in \text{Pseudos} : \\ (i,p_1,p_2) \in \text{loadPair} \Rightarrow \\ (\exists r_1,r_2 \in \text{Regs} : (i,p_1,r_1),(i,p_2,r_2) \in \text{spLoad})$$

For example, in the directives specified in Figure 5 for the MIRA program given in Figure 4, it would have been erroneous if (i1,p1,p2) $\in$ loadPair.

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

$$\forall i \in \text{Insts} : \forall p_1,p_2 \in \text{Pseudos} : \\ (i,p_1,p_2) \in \text{storePair} \Rightarrow \\ (\exists r_1,r_2 \in \text{Regs} : (i,p_1,r_1),(i,p_2,r_2) \in \text{spStore})$$

For example, in the directives specified in Figure 5 for the MIRA program given in Figure 4, it would have been erroneous if (i1,p1,p2) $\in$ storePair.

The correctness checker takes the MIRA program and the output directives in FORD format and then checks to see that the above constraints are met. RALF applies these directives only if these constraints are satisfied.

Our safety checks are independent of the register allocator. The safety checks are necessary for ensuring the correct functioning of the framework. The safety checks are also necessary for the correctness of the executable code generated by the framework. These safety checks offer an enormous amount of help when debugging a register allocator. Since errors are caught at an early stage, the safety checks help with pin pointing the exact source of an error.

4. Experimental Results

We have implemented RALF over the strongARM port of the gcc-2.95.2 compiler. The framework gets activated by different compiler switches and is implemented by around 5000+ lines of C code and around 125+ functions.

```
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 6. Pseudo code for naive register allocator.

4.1 Versatility: Test by Implementation

We show the versatility of our framework by implementing a variety of register allocators. Each of the register allocators has a different need for data about programs and hence poses different types of challenges to the framework. We have implemented a variety of register allocators to cover a spectrum of typical needs of different register allocators. They are a naive register allocator, linear scan register allocation [PS99], iterated register coalescing [GA96], integer linear program (ILP) based register allocation [NP05], stack location allocation combined with register allocation (SARA) [NP05], and chordal graph based register allocation [PP05]. We present the naive register allocator as a vehicle for explaining some detailed points about RALF. For the reader's convenience, we give a brief presentation of the register allocators in the appendix.

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 Figure 6. Because of the input interface requirements presented in section 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 register 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. The naive register allocator, though only of academic interest, can be used as a first level test case for a register allocation framework.

For the code snippet shown in Figure 3, the output generated (FORD directives) by the naive register allocator is shown in Figure 7. It can be easily checked that all the syntactic and semantic checks specified in section 3 are satisfied. For example, the semantic constraint Stack Location, would require that both `p1` and `p2` have a stack location; the section f in Figure 7 confirms that. Thus, the correctness checker verifies the FORD directives and then RALF generates assembly code as shown in Figure 8. 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 register allocation directives, shown in Figure 7.

Owing to the simplicity of the allocation scheme the code generated is obviously inefficient. However, it can also be seen that, naive register allocator does not use any caller save registers and hence avoids some extra spills/reloads that might have taken place during calls.

4.2 Experience with Register Allocation

In this section we present our experience in using RALF with the register allocation techniques listed in section 4.1. We will

```
PsR :=                  f:=
i2 p1 r0                p1 p1
i3 p1 r0 ;              p2 p2 ;
xdef :=                 spLoad:=
i1 p1 r0                i2 p1 r0
i2 p1 r0                i3 p1 r0 ;
i3 p2 r0 ;              loadPair:=;
spStore:=               storePair:=;
i1 p1 r0                inverseLoad:=;
i2 p1 r0                inverseStore:=;
i3 p2 r0 ;              moveInst:=;
```

Figure 7. Output of Naive register allocator for the code snippet in Figure 3.

```
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 8. Assembly code generated from the register allocator output in Figure 7

RA	#LOC		Hrs to Code
	RA	Interface	the interface
Naive	196 (J)	773 (J)	< 10
IRC	3538 (J)	773 (J)	< 10
CG	4134 (J)	773 (J)	< 10
LS	385 (J)+	1100 (J)+	< 5
RA_i	495 (A)	298 (A)	< 1
SARA	731 (A)	400 (A)	< 1

Table 1. Experimental evaluation of RALF.

be using the following abbreviations: (Naive - The naive register allocator, IRC - Iterated register coalescing, LS - Linear scan, CG - Register allocation via coloring chordal graphs, RA_i - ILP based register allocation, SARA - Combined ILP based stack allocation and register allocation)

For each of these register allocator techniques, Table 1 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. The number of lines of code is annotated by (J) or (A), signifying the coding language used: Java or AMPL [FGK93]. The framework also provides a MIRA grammar in javacc format. For 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 that the number of hours taken to write the interface code is 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. For the two ILP based register allocators we present here, we did not have to write much

bench	#LOC	#RTL	#fns	gcc-O2		Naive		IRC		DMC		LS		RA		SARA	
				mem	csr	mem	csr	mem	csr	mem	csr	mem	csr	mem	csr	mem	csr
sieve	39	134	3	0	9	96	14	0	14	0	18	7	17	0	9	0	9
matmul	56	254	6	9	22	194	23	7	23	7	30	37	24	9	20	7	19
queen	58	144	4	11	14	110	14	14	14	12	17	24	13	12	11	8	11
url	790	1264	12	115	62	845	42	235	48	165	61	313	16	120	56	120	58
yacr2	3979	10838	58	1060	123	2287	358	2144	295	7992	258	3056	335	1003	123	1109	142
ft	2155	3218	35	219	92	1376	116	609	91	369	156	538	151	225	87	230	106
c4	885	3388	21	189	289	3547	123	406	142	434	171	979	148	190	305	184	320

Table 2. Benchmark characteristics and compile time statistics

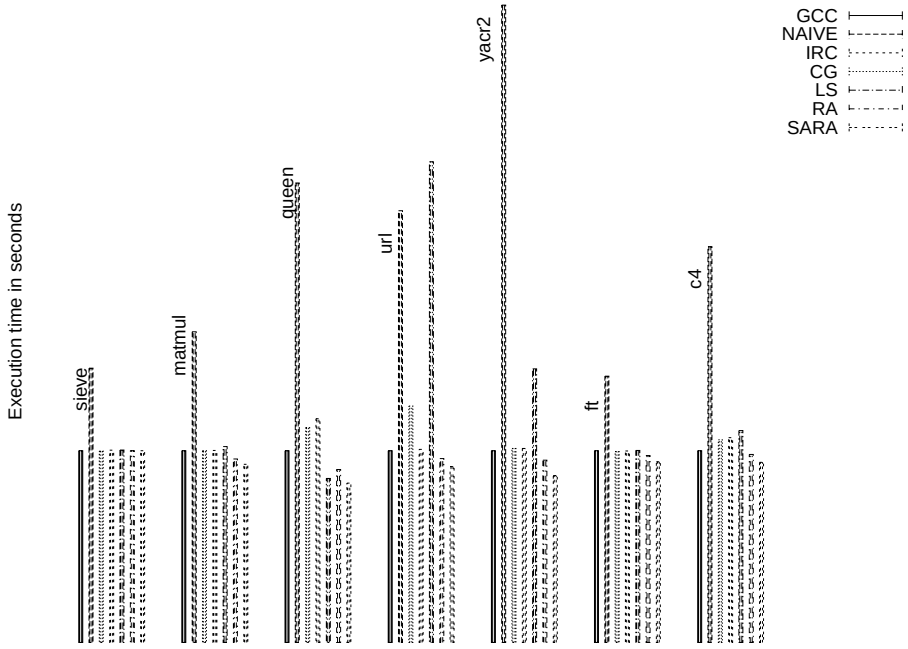

Figure 9. Comparison of different register allocators using execution time of benchmarks as the metric.

interface code as the syntax of MIRA is a subset of AMPL. We just had to write some code to ignore some parameters that are unused.

We have implemented each of the register allocators listed in section 4.1 in RALF and then tested the generated the target code on a Stargate platform. Stargate has a StrongARM/XScale processor, 64MB SDRAM, and no cache. We have drawn our benchmark programs from a variety of sources:

- Stanford Benchmark suite: The three benchmarks sieve, matmul, and queen, though small and simple, are typical of the subroutines of many larger programs.
- NetBench: Url is a network related benchmark from the NetBench [MBSW01] suite, 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 c4 benchmark is 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.

The static characteristics and compile time statistics of these benchmarks are presented in Table 2. The static characteristics we present here include the number of lines of C code, the number of instructions seen by the register allocator (which depends on the number of RTL instructions in the intermediate representation of the program), and the number of functions. All these benchmarks have the common characteristic that they are non-floating point benchmarks. We chose benchmarks that operate on (sub)integer / character types only. This is to ensure that the programs need to access only non-floating point registers. (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.) For each benchmark, we present two compile time statistics: The number of memory accesses (mem) introduced (because of spill/reload instructions), and the number of callee save registers (csr) used (leads to more memory accesses).

In Figure 9 we present a comparative study of the register allocators described in section 4.1. 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.

The naive register allocator performs most poorly 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 performance 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, followed by (b) conservative allocation for the whole function. It can be seen that even without tuning CG and ICR much, their performance is comparable to that of gcc.

One important observation that can be made from the graph is that it suggests an upper limit on the gains any register allocator can make. Even in the best of the cases, the code generated by the naive allocator is worse by a factor of 2.5 (as compared to the ILP based allocator). One would expect a much higher gain just because of the number of loads and stores the naive allocator would introduce. However, the important point is that most of these benchmarks deal with structures and arrays that require compulsory memory access. And it has been observed that those accesses overshadow the spill cost. So, if speed is not a concern at all then even naive register allocator will do a good job. A simple approach like linear scan, also gives good results in most of the cases. And by improving it a little bit (using chordal graph based approach etc) the improvements are many folds. However, if compiler wants to drain every bit of improvement possible then it can fallback upon ILP based or other near-optimal solutions.

Let us compare the static counts in Table 2 and the execution times in Figure 9. For example, for the benchmark ft, we have a significant difference in the static counts among the register allocators, but only small differences in the execution times. Those numbers confirm that benchmark execution times give a better comparison of register allocators than static counts of spill code and number of callee save registers.

Table 3 presents a list of different register allocators (including the ones described in section 4.1 and more) and the different types of data required by them. We categorize the data in a format similar to the input interface described in section 2.1: Three different sets: I(nsts), P(seudos), and L(oc), and seven parameters: L(ive), R(eq), D(ef), P(revI), J(oinI), C(allI), and M(oveI). We have grouped all the register allocators into four categories: (a) Graph coloring based, (b) Control flow based, (c) ILP based, and (d) others. It can be noticed that the data provided by RALF to the register allocators is expressive enough that most of the register allocators can be implemented. Because of the limitation of the current framework about handling pseudos of different sizes, namely that RALF assumes that every pseudo is of the same size, register allocators requiring the size of the pseudos (e.g. [TG03]) would not be able to perform to their fullest potential.

5. Conclusion

We have presented a framework for end-to-end evaluation of register allocators. We have shown that the framework is easy to use and versatile enough that a variety of register allocation schemes can be implemented relatively easily.

Most publicly available compilers and general compiler frameworks present a complicated interface for register allocators, partly due to many corner cases that can arise during compilation. In contrast, many papers on register allocation focus on the core problems and ignore the corner cases. RALF bridges the gap by (a) presenting various pieces of program-specific information independently and (b) hiding some complications from the register allocator by simplifying the program model. As a result, RALF has the following limitations:

- RALF handles integer and sub-integer data types, but not temporaries of type float or double.
- RALF does not provide any information about the *size* of the pseudos and lets the programmer assume that all the pseudos are of the same size. A register allocator requiring size information will find RALF inadequate.
- RALF does not handle pair registers (and hence not pair temporaries). We believe this limitation can be overcome by extending MIRA with information about the pairing of machine registers.
- RALF works for ARM targets only.

Along with RALF, we have a MIRA grammar in javacc format. Programmers can build libraries based on the MIRA grammar which can be used by many register allocators. Currently we have implemented library classes to display a MIRA program as three address codes, build live intervals, and build interference graphs. We also have implemented a MIRA program visualizer that can graphically display the control flow and data flow information in the program. It also uses the use-def information information to generate interference graphs. The framework, MIRA grammar, grammer for FORD directives, and our tools can be found at the RALF homepage:

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

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*, pages 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.

[AP02] Andrew W. Appel and Jens Palsberg. *Modern compiler implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2002.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[ATGL96] Ali-Reza Adl-Tabatabai, Thomas Gross, and Guei-Yuan Lueh. Code reuse in an optimizing compiler. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 51–68, New York, NY, USA, 1996. ACM Press.

[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.

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

[EAEF99] K.M. Elleithy and E.G. Abd-El-Fattah. A genetic algorithm for register allocation. In *Ninth Great Lakes Symposium on VLSI*, pages 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.

	Alg	Sets			Parameters							
		I	P	L	L	R	D	P	J	C	M	Others
Graph Based	[PP05]	√	√	√	√	√	√	√	√	√	√	
	[GA96]	√	√	√	√	√	√	√	√	√	√	
	[Cha82]	√	√	√	√	√	√			√	√	
	[LGAT96]	√	√	√	√	√	√	√	√	√		
C F Based	Naive	√	√	√		√	√					
	[PS99]	√	√	√	√	√	√			√		
	[THS98]	√	√	√	√	√	√			√		
	[Fre74]	√	√	√		√	√	√	√	√		
ILP Based	[AG01]	√	√	√	√	√	√	√	√	√	√	
	[FW02]	√	√	√	√	√	√	√	√	√		
	[GW96]	√	√	√	√	√	√	√	√	√	√	
	[NP05]	√	√	√	√	√	√	√	√	√		
Other	[TG03]	√	√	√	√	√	√	√	√	√		√
	[EAEF99]	√	√	√		√	√	√	√	√		

Table 3. Data structures used by each register allocation algorithm.

[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*, pages 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.

[gcc05] GNU C compiler. 2005. http:// gcc. gnu. org.

[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.

[HAA+96] Mary W. Hall, Jennifer-Ann M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[LGAT96] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Global register allocation based on graph fusion. In *Languages and Compilers for Parallel Computing*, pages 246–265, 1996.

[MBSW01] G. Memik, B.Mangione-Smith, and W.Hu. Netbench: A benchmarking suite for network processors. *IEEE International Conference Computer-Aided Deisgn*, November 2001.

[NP03] V. Krishna Nandivada and Jens Palsberg. Efficient spill code for SDRAM. In *Proceedings of the international conference on Compilers, architecture and synthesis for embedded systems*, pages 24–31, 2003.

[NP05] V. Krishna Nandivada and Jens Palsberg. Sara: Combining stack allocation and register allocation. In *Manuscript*, 2005.

[PP05] Fernando MQ Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of the 3rd 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.

[sml00] Standard ML of New Jersey. 2000. http:// gcc. gnu. org.

[TG03] Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. In *Proceedings of the 30th Symposium on Principles of programming languages*, pages 85–96, 2003.

[THS98] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 1998.

[VRCG+99] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of CASCON*, 1999.

A. Appendix: Register Allocation Schemes

A.1 Linear Scan Register Allocation

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 is shown in Figure 8. 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 paper 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

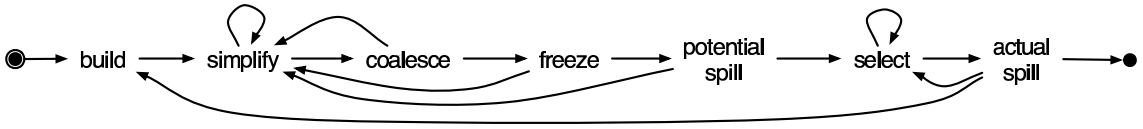

Figure 10. Iterated register coalescing.

- 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.

A.2 Iterated Register Coalescing

George and Appel proposed iterated 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 Figure 10 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.

A.3 ILP-based Register Allocation

We use the integer linear program (ILP) based register allocator (RA_i) presented in [NP05] 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.

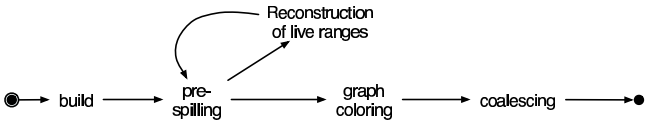

Figure 11. Chordal graph based register allocation.

A.4 SARA

We present in [NP05] a combined phase for register allocation and stack location allocation. Such a register allocator can be 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 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.

A.5 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 Figure 11 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.