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

V. Krishna Nandivada[1]
Fernando Magno Quintão Pereira[2]
Jens Palsberg[2]

[1] IBM India Research Laboratory, Delhi
[2] UCLA Computer Science Department, University of California, Los Angeles

Abstract. This paper presents a framework for designing, verifying, and evaluating register allocation algorithms. The proposed framework has three main components. The first component is MIRA, a language for describing programs prior to register allocation. The second component is FORD, a language that describes the results produced by the register allocator. The third component is a type checker for the output of a register allocator which helps to find bugs. To illustrate the effectiveness of the framework, we present RALF, a tool that allows a register allocator to be integrated into the gcc compiler for the StrongARM architecture. RALF simplifies the development of register allocators by sheltering the programmer from the internal complexity of gcc. MIRA and FORD's features are sufficient to implement most of the register allocators currently in use and are independent of any particular register allocation algorithm or compiler. To demonstrate the generality of our framework, we have used RALF to evaluate eight different register allocators, including iterated register coalescing, linear scan, a chordal based allocator, and two integer linear programming approaches.

1 Introduction

1.1 Background

The register allocator is one of the most important parts of a compiler. Our experiments show that an optimal algorithm can improve the execution time of the compiled code by up to 250%. Although researchers have studied register allocation for a long time, many interesting problems remain. For example, in recent years PLDI (ACM SIGPLAN Conference on Programming Language Design and Implementation) has published several papers on register allocation [2004 (2 papers), 2005 (3 papers), 2006 (2 papers)]. While the essence of register allocation is well understood, developing a high-quality register allocator is nontrivial. In addition to understanding the register allocation algorithm, which can be complex, the developer must also know the internals of the compiler where the allocator will be implemented. For example, public domain compilers such as GCC [2] or SMLNJ [1] and compiler frameworks such as SUIF [15] or

SOOT [31] allow a programmer to implement a new register allocator. However, the programmer has to understand and work with their data structures, which are complicated because register allocation affects both the machine specific and machine independent parts of the compilation process. One attempt to address this problem was Tabatabai et al's. [3] register allocation framework, implemented in the CMU C compiler. Their framework presents modules (for example, graph construction, coalescing, color assignment, spill code insertion, and others) that different register allocators might need. However, if the allocator needs mechanisms other than those provided by the framework, the programmer must still deal with the internals of the CMU C compiler. A goal of our work is to completely shield the developer of register allocators from the internal complexities of a compiler.

Debugging register allocators is also a complicated task. Errors may surface in non-trivial ways; sometimes many instructions after the incorrect code. Moreover, the low-level nature of the machine code and its large size makes visual inspection of the register-allocator's output tedious and error-prone. As a testimony of these difficulties, most recent publications in this field report only static data and not run-time measurements, let alone implementations in industrial compilers. Although static data (such as number of spills, and number of registers used) is important, it does not reflect the behavior of the register allocator in the presence of other optimizations and run time factors.

A few researchers have developed techniques for proving register allocators correct. Naik and Palsberg [21] proved the correctness of the ILP-based register allocator of Appel and George [6]. Ohori [26] designed a register allocation algorithm as a series of proof transformations which is correct by construction. Other researchers have shown how to validate the output of register allocators. Necula [24] presented a translation validation infrastructure for the gcc compiler that includes register allocation. Necula's scheme treats the memory address of a spilled register as a variable, which allows reasoning about its live ranges, although relying on specific characteristics of the gcc compiler, such as addressing modes. Leroy [17] formally describes a technique to validate the output of graph coloring based register allocation algorithms. Basically, if the interference graph contains a pair of adjacent temporaries allocated to the same register, the verifier emits an error, otherwise it assumes that the code generated is correct. Andersson [5] and Pereira et al. [27] adopted similar approaches. Huang et al. [16] presented a more general approach which matches the live ranges of values in the original program against the live ranges of machine locations in the register-allocated program. A goal of our work is to use a type system to validate the output of register allocators, and to prove the soundness of the type system itself.

Annotations in the register-allocated code can help validation algorithms. Morrisett et al. [20] used type annotations to help guarantee memory safety, Necula and Lee [25] used more general annotations such as memory bounds, and Agat [4] used type annotations to validate the output of a register allocator.

A goal of our work is to validate the output of register allocators without extra annotations in the target code.

1.2 Our Results

We present a framework for designing, verifying, and evaluating register allocators. Our framework shields the developer from the internal complexities of a compiler, uses a type system to validate the output of register allocators, and does not rely on code annotations.

MIRA and FORD. Our framework centers around MIRA (Mathematical Intermediate representation for Register Allocation), a language for describing programs prior to register allocation, and FORD (FOrmat for Register allocation Directives), a language that describes the results produced by the register allocator. MIRA sources are abstract intermediate representations of programs immediately before the register allocation phase. MIRA descriptions contain architecture and program specific information. The former includes information such as number and classes of machine registers, number of caller-save registers, and costs of loads and stores. The latter consists of information such as the program's control flow graph, use and definition sites of each variable, and estimated usage frequency of each instruction. In the context of our framework, the register allocator emits FORD directives that control spill code generation, register and variable mapping at different program points, and any additional code that needs to be inserted (For example, move instructions). MIRA and FORD can accommodate many of the traditional register allocation algorithms and are simple enough to be easily used by the developer of register allocators.

Type system. We use a type system to verify that the output of a register allocator is correct. Our type system was inspired by Morrisett et al.'s type system for assembly language [20] and can be used with intermediate representations other than MIRA and FORD. A type correct program is guaranteed to have properties such as: (1) pseudos whose live ranges overlap are assigned to different registers, (2) live ranges of the same pseudo always reach a join point assigned to the same register, and (3) a live register is not overwritten before it is used. We have found that typical errors in the implementation of a register allocator violate these properties. The type checker points out the locations of the register-allocated code where these properties fail. We have proved type soundness for our type system using the Twelf Meta-theorem prover [32].

RALF. Our tool RALF (Register ALlocation Framework) allows a programmer to plug a new register allocator into gcc, without requiring the programmer to know any details of gcc's implementation. RALF is an extra layer added on top of gcc, acting as a glue between gcc and the plugged-in register allocator. The new register allocator takes a MIRA program as input and gives a collection of FORD directives as output. The main objective of RALF is to be *simple*: when writing a register allocator compatible with our framework, the developer only has to write a program that translates MIRA to FORD. RALF treats the register allocator, which can be implemented in any language, as a black box whose only purpose is to translate MIRA sources (provided by RALF) to FORD directives

(which are fed back to RALF). Given an input program and the plugged-in register allocator, RALF generates a StrongARM binary executable using the new register allocator and the rest of gcc. RALF uses our type checker to verify the output of the register allocator. In addition to our own experiments with RALF, we have used RALF in an advanced compiler course at UCLA. As part of a class assignment, each student implemented a different register allocation algorithm to be used with the framework. More about these experiences can be found at RALF's homepage `http://compilers.cs.ucla.edu/ralf`. Our current implementation of RALF compiles only C code to the StrongARM architecture; however, the MIRA and FORD description languages are designed to be general enough to fit other source languages and architectures. The RISC nature of the StrongARM architecture helps to keep RALF simple. Our implementation, which gets activated by different compiler switches, contains around 5000 lines of C code divided among 125 functions. The proposed framework is not intended for industrial implementations of register allocators, but for fast implementation and testing of research prototypes. The techniques used in our type system can be used also in a production compiler.

Comparison of eight register allocators. We have used RALF to implement and compare eight different registers allocators. The allocators tested range from classical algorithms, such as the usage-count based implementation [11], to novel approaches, such as register allocation via coloring of chordal graphs [27]. In addition of using static data, such as number of variables spilled, we compare the different algorithms by running the produced code on a StrongARM processor.

The remainder of this paper is organized as follows: Section 2 describes a simplified version of MIRA and FORD and characterizes the register allocation problem. Section 3 presents our type system, and Section 4 discusses our experimental results.

2 A Simplified view of MIRA and FORD

Register allocation is the process of mapping a program $\mathcal{M}$ that can use an unbounded number of variables, or *pseudo-registers*, to a program $\mathcal{F}$ that must use a fixed (and generally small) number of *machine registers* to store data. In the remainder of this paper we use *register* in place of *machine register*, and *pseudo* for *pseudo-register*. If the number of registers is not sufficient to accommodate all the pseudos, some of them must be stored in memory; these are called *spilled* pseudos. Following the nomenclature normally used in the gcc community, we call the intermediate representation of programs $\mathcal{M}$ the *Register Transfer Language (RTL)* and we use *Location Transfer Language (LTL)* to describe programs $\mathcal{F}$. As shown in Figure 1, MIRA and FORD have been designed to constitute an interface between the register allocator and the RTL/LTL intermediate representations. They facilitate the development of register allocation algorithms by hiding from the algorithm's designer the details of the RTL/LTL representation that are not relevant to the register allocation process. In order to precisely

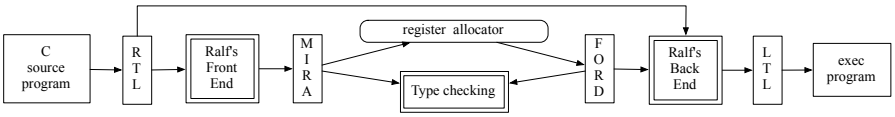

Fig. 1. Block diagram of our register allocation framework.

characterize the register allocation problem, we will use a simplified version of MIRA and a simplified version of FORD in this section. We will call them sMIRA and sFORD respectively. For our purposes, a register allocator RA is a black box which takes an sMIRA program P_{sm} and a description of the target architecture, and produces an sFORD program P_{sf}. In this section, we will assume that the architecture specific information is a list of K machine registers Regs, and a set of caller save registers $\mathsf{CallerSave} \subseteq \mathsf{Regs}$. Thus, $P_{sf} = RA(P_{sm}, \mathsf{Regs}, \mathsf{CallerSave})$. An actual register allocator would require more information, such as the class of each machine register, the cost of different operations, etc. Such information is present in the concrete specification of MIRA and FORD, which is given in the full version of this paper, available at `http://compiler.cs.ucla.edu/ralf`.

2.1 Simplified MIRA

sMIRA programs are described by the grammar in Figure 2(a). We adopt an abstract representation of programs. All the operands not relevant to register allocation, such as constants, heap memory addresses, and others, are represented with the symbol $\bullet$. The only explicit operands are pseudos (p) and pre-colored registers (r, p). The latter are pseudos that have been assigned a fixed machine register due to architectural constraints. In this simplified presentation, we only use pre-colored register to pass parameters to function calls, and to retrieve their return value. Other operands, such as constants and memory references on the heap are abstracted out.

An sMIRA program is a sequence of *instruction blocks*. Each instruction block consists of an address label, represented by L, heading a sequence of instructions (I) followed by a jump. sMIRA programs can use an unbounded number of pseudo registers p. We do not distinguish the opcode of instructions, except *branches* and function calls. Branches affect the control flow, and *function calls* may cause caller save registers to be overwritten, once register allocation has been performed. A function call such as $(r_0, p_0) = \mathsf{call}\ (r_1, p_1)..(r_s, p_s)$ uses pre-colored pseudos $(r_1, p_1)..(r_s, p_s)$ as parameters, and produces a return value in the pre-colored pseudo (r_0, p_0). Notice that in sMIRA a call instruction does not contain a label; that is because we support only intra-procedural register allocation. An example of sMIRA program is given in Figure 2(b).

(Programs)	$P_{sm} ::= L_1 I_1; \ldots; L_k I_k$
(Code labels)	$L ::= L_1 \mid L_2 \mid \ldots$
(Instr. Sequence)	$I ::=$
- (Jump)	$\mid$ jump L
- (Sequence)	$\mid i; I$
(Pseudos)	$p ::= p_1 \mid p_2 \mid \ldots$
(Registers)	$r ::= r_1 \mid \ldots \mid r_k$
(Operands)	$o ::=$
- (Constants)	$\mid \bullet$
- (Pseudos)	$\mid p$
- (Pre-coloreds)	$\mid (r, p)$
(Instructions)	$i ::=$
- (Assignment)	$\mid p = o$
- (Ass. pre-col.)	$\mid (r, p) = o$
- (Cond. jump)	$\mid$ if p jump L
- (Function call)	$\mid (r_0, p_0) =$ call $(r_1, p_1)..(r_s, p_s)$

(a)

L_1	L_2
$p_0 = \bullet$	$p_0 = p_0$
$p_1 = \bullet$	$(r_0, p_6) = \bullet$
$p_2 = p_0$	$(r_0, p_6) =$ call (r_0, p_6)
if $p1$ jump L_3	$p_2 = (r_0, p_6)$
jump L_2	$p_3 = p_0$
	$p_4 = \bullet$
L_3	$p_7 = p_4$
$p_4 = p_0$	if p_3 jump L_2
$p_5 = p_2$	jump L_3
jump $exit$	

(b)

Fig. 2. (a) Syntax of sMIRA programs. (b) Example of sMIRA program.

(Programs)	$P_{sf} ::= L_1\ I_1; \ldots; L_k\ I_k$
(Code Labels)	$L ::= L_1 \mid L_2 \mid \ldots$
(Inst. seq.)	$I ::=$
- (Jump)	$\mid$ jump l
- (Sequence)	$\mid i; I$
(Pseudos)	$p ::= p_1 \mid p_2 \mid \ldots$
(Registers)	$r ::= r_1 \mid r_2 \mid \ldots$
(Mem. locs.)	$l ::= l_1 \mid l_2 \mid \ldots$
(Operands)	$o ::=$
- (Constant)	$\mid \bullet$
- (Reg. bind)	$\mid (r, p)$
- (Mem. bind)	$\mid (l, p)$
(Instructions)	$i ::=$
- (Assig.)	$\mid (r, p) = o$
- (Store)	$\mid (l, p) = o$
- (Cond. jump)	$\mid$ if (r, p) jump L
- (Func. call)	$\mid (r_0, p_0) =$ call $(r_1, p_1)..(r_s, p_s)$

(a)

L_1	L_2
$(r_1, p_0) = \bullet$	$(r_1, p_0) = (r_1, p_0)$
$(r_0, p_1) = \bullet$	$(r_0, p_6) = \bullet$
$(r_1, p_2) = (r_1, p_0)$	$(l_1, p_0) = (r_1, p_0)$
$(l_0, p_2) = (r_1, p_2)$	$(r_0, p_6) =$ call (r_0, p_6)
if $(r_1, p1)$ jump L_3	$(r_1, p_0) = (l_1, p_0)$
jump L_2	$(r_2, p_2) = (r_0, p_6)$
	$(l_0, p_2) = (r_2, p_2)$
L_3	$(r_2, p_3) = (r_1, p_0)$
$(r_1, p_4) = (r_1, p_0)$	$(r_0, p_4) = \bullet$
$(r_0, p_2) = (l_0, p_2)$	$(r_0, p_7) = (r_0, p_4)$
$(r_0, p_5) = (r_0, p_2)$	if (r_1, p_3) jump L_2
jump $exit$	jump L_3

(b)

Fig. 3. (a) Syntax of sFORD programs (b) Example of sFORD program.

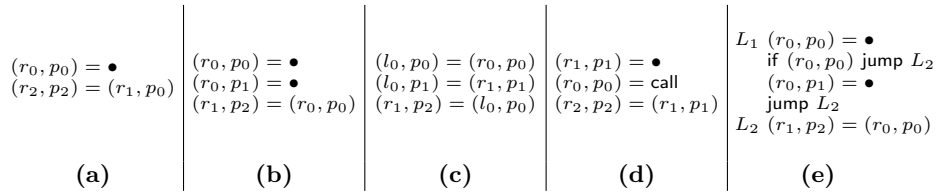

Fig. 4. (a-e) Examples of errors due to wrong register allocation.

2.2 Simplified FORD

A register allocator produces sFORD programs, which are represented by the grammar in Figure 3(a). Operands in sFORD are bindings of pseudos (p) to *machine locations*. In our representation, a machine location can be either a physical register (r), or a memory address (l). In addition to calls and branches, we distinguish loads "$= (l, p)$", and stores "$(l, p) =$", because these instructions are used to save and restore spilled values. Notice that we make an explicit distinction between *code labels*, represented by L, and data labels (stack locations), represented by l. In the sFORD representation, caller-save registers can be overwritten by function calls; thus, the register allocator must guarantee that pseudos that are alive across function calls are not mapped to caller-save registers.

Let P_{sm} be the sMIRA program in Figure 2(b), and consider an architecture where $\mathsf{Regs} = \{r_0, r_1, r_2\}$ and $\mathsf{CallerSave} = \{r_0, r_1\}$. Let RA be a hypothetical register allocator, such that $P_{sf} = RA(P_{sm}, \mathsf{Regs}, \mathsf{CallerSave})$ is the program in Figure 3(b). In our example, RA has allocated register r_1 to pseudo p_0 in the first instruction of L_1. The pseudo p_2 has been spilled due to the high register pressure in block L_2; its memory location is given by the label l_0. Furthermore, pseudo p_0 has been spilled to memory location l_1 because it is stored in the caller save register r_0 and is alive across a function call.

3 Type Checking

Inaccuracies in the implementation of a register allocation algorithm may result in different types of errors in the sFORD program. In Figure 4 we illustrate five different errors that can be produced by a flawed register allocator. In Fig. 4(a), p_0 was defined in register r_0 at instruction 1, but it is expected to be found in register r_1 when used in instruction 2. In Fig. 4(b) register r_0 is overwritten in instruction 2 while it contains the live pseudo p_0. Fig. 4(c) describes a similar situation, but in this case a memory location is overwritten while the value it holds is still alive. In Fig. 4(d), we assume that r_1 is a caller save register. In this case, pseudo p_1 may have its location overwritten during the execution of the function call in instruction 2. Finally, in Fig. 4(e) the value of p_0, stored in register r_0 may be overwritten, depending on the path taken during the execution of the program. This last error is particularly elusive, because its consequences might not surface during the testing of the target program.

In order to guarantee that the values used in the sMIRA program are preserved in the sFORD representation, we use a type checker inspired by [20]. The basic data used in our type system are machine locations, and the type of a machine location is the pseudo-register that it stores. In our case, the register allocator annotates each definition or use of data with its type. A definition of a machine location, e.g $(r_i, p_j) = \bullet$ corresponds to declaring r_i with the type p_j. Let (r_i, p_j) be an annotated machine location. Intuitively, every time this machine location is used, e.g $(r, p) = (r_i, p_j)$, its annotated type corresponds to the type that can be discovered by a type inference engine if the sFORD program is correct. For instance, the program in Figure 4(a) is incorrect because p_0, the type of r_1 in the second instruction cannot be inferred. Notice that these annotations can be inferred from the sMIRA/sFORD programs; they are not present in the final LTL code.

3.1 Operational semantics of sFORD programs

We define an abstract machine to evaluate sFORD programs. The state M of this machine is defined in terms of a tuple with four elements: (C, D, R, I). If M is a program state and we have M' such that $M \rightarrow M'$, then we say that M can *take a step*. A program state M is *stuck* if M cannot take a step. A program state M *goes wrong* if $\exists M' : M \rightarrow^* M'$ and M' is stuck. I is defined in Figure 3(a); the code heap C, data heap D, and register bank R are defined below.

$$
\begin{array}{ll}
\text{(Code Heap)} & C ::= \{L_1 = I_1, \ldots, L_k = I_k\} \\
\text{(Data Heap)} & D ::= \{l_1 = p_1, \ldots, l_m = p_m\} \\
\text{(Register Bank)} & R ::= \{r_1 = p_1, \ldots, r_n = p_n\} \\
\text{(Machine State)} & M ::= (C, D, R, I)
\end{array}
$$

The evaluation rules for our abstract machine are given in Figure 5. Rules 1, 2 and 3 evaluate the operands of sFORD. The assignment statement (Rule 4) modifies the mapping in the register bank, and the store statement (Rule 5) modifies the mapping in the data heap. The result of a conditional branch has no importance in our representation. Therefore, an instruction such as $\mathsf{if}\ (r, p)\ \mathsf{jump}\ v$ is evaluated non-deterministically by either Rule 6 or Rule 7. We conservatively assume that a call instruction changes the contents of all the caller save registers. It also defines a register with the return value (Rule 8). In order to simulate the effects of a function call on the caller-save registers we define the erasing function "$\diamond$" below. We augment the set of pseudo-registers with $\perp$. This pseudo will be used as the type of non-initialized registers and we assume that it is not defined in any instruction of the original sMIRA program.

$$
\begin{array}{l}
\diamond : R \times X \mapsto R' \\
(R \diamond X)(r) = \perp\ \mathsf{if}\ r \in X,\ \mathsf{else}\ R(r)
\end{array}
$$

In Figure 5 the set X is replaced by the set of caller-save registers. We draw the attention of the reader to the premises of rules 4 to 8, which ensure that a location used by an instruction indeed contains the pseudo that is expected by

that instruction. For example, for the sFORD instruction $(r_1, p_1) = (r_0, p_0)$, the premise of Rule 4 ensures that r_0 is holding the value p_0 when this instruction is executed.

$$D, R \vdash \bullet \tag{1}$$

$$D, R \vdash (r, p), \mathsf{if}\ R(r) = p \wedge p \neq \bot \tag{2}$$

$$D, R \vdash (l, p), \mathsf{if}\ D(l) = p \wedge p \neq \bot \tag{3}$$

$$\frac{D, R \vdash o}{(C, D, R, (r, p) = o; I) \rightarrow (C, D, R[r \mapsto p], I)} \tag{4}$$

$$\frac{D, R \vdash o}{(C, D, R, (l, p) = o; I) \rightarrow (C, D[l \mapsto p], R, I)} \tag{5}$$

$$\frac{D, R \vdash (r, p)}{(C, D, R, \mathsf{if}\ (r, p)\ \mathsf{jump}\ L; I) \rightarrow (C, D, R, I')}\ C_{cond} \tag{6}$$

$$C_{cond} = L \in \mathsf{domain}(C) \wedge C(L) = I'$$

$$\frac{D, R \vdash (r, p)}{(C, D, R, \mathsf{if}\ (r, p)\ \mathsf{jump}\ L; I) \rightarrow (C, D, R, I)} \tag{7}$$

$$\frac{\begin{array}{c}\forall (r_i, p_i), 1 \leq i \leq s, D, R \vdash (r_i, p_i) \\ (C, D, R, (r_0, p_0) = \mathsf{call}\ (r_1, p_1), \ldots, (r_s, p_s); I)\end{array}}{\begin{array}{c}\rightarrow (C, D, (R \diamond \mathsf{callerSave})[r_0 \mapsto p_0], I)\end{array}} \tag{8}$$

$$(C, D, R, \mathsf{jump}\ L) \rightarrow (C, D, R, I)\ \ \mathsf{if}\ C_{jump} \tag{9}$$

$$C_{jump} = L \in \mathsf{domain}(C) \wedge C(L) = I$$

Fig. 5. Operational Semantics of sFORD programs

Operands

$$\vdash \bullet : \mathsf{Const} \tag{10}$$

$$\frac{\Gamma(r) = p \qquad p \neq \bot}{\Gamma \vdash (r, p) : p} \tag{11}$$

$$\frac{\Delta(l) = p \qquad p \neq \bot}{\Delta \vdash (l, p) : p} \tag{12}$$

Instructions

$$\frac{\Delta; \Gamma \vdash o : t \qquad p \neq \bot}{\Psi \vdash (r, p) = o : (\Gamma \times \Delta) \mapsto (\Gamma[r : p] \times \Delta)} \tag{13}$$

$$\frac{\Delta; \Gamma \vdash o : t \qquad p \neq \bot}{\Psi \vdash (l, p) = o : (\Gamma \times \Delta) \mapsto (\Gamma \times \Delta[l : p])} \tag{14}$$

$$\frac{\Gamma \vdash (r, p) : p \qquad \Psi \vdash L : (\Gamma' \times \Delta') \qquad (\Gamma \times \Delta) \leq (\Gamma' \times \Delta')}{\Psi \vdash \mathsf{if}\ (r, p)\ \mathsf{jump}\ L : (\Gamma \times \Delta) \mapsto (\Gamma \times \Delta)} \tag{15}$$

$$\frac{\forall (r_i, p_i), 1 \leq i \leq s, \Gamma \vdash (r_i, p_i) : p_i \qquad p_0 \neq \bot}{\Psi \vdash (r_0, p_0) = \mathsf{call}\ (r_1, p_1), \ldots, (r_s, p_s)\ : (\Gamma \times \Delta) \mapsto ((\Gamma \diamond \mathsf{callerSave})[r_0 : p_0] \times \Delta)} \tag{16}$$

Instruction sequences

$$\frac{\Psi \vdash L : (\Gamma' \times \Delta') \qquad (\Gamma \times \Delta) \leq (\Gamma' \times \Delta')}{\Psi \vdash \mathsf{jump}\ L : (\Gamma \times \Delta)} \tag{17}$$

$$\frac{\Psi \vdash i : (\Gamma \times \Delta) \mapsto (\Gamma' \times \Delta') \qquad \Psi \vdash I : (\Gamma' \times \Delta')}{\Psi \vdash i; I : (\Gamma \times \Delta)} \tag{18}$$

Bank of Registers

$$\frac{\forall r \in \mathsf{domain}(\Gamma).\ \vdash R(r) : \Gamma(r)}{\Psi \vdash R : \Gamma} \tag{19}$$

Data Heap

$$\frac{\forall l \in \mathsf{domain}(\Delta).\ \vdash D(l) : \Delta(l)}{\Psi \vdash D : \Delta} \tag{20}$$

Code Heap

$$\frac{\forall L \in \mathsf{domain}(\Psi). \Psi \vdash C(L) : \Psi(L)}{\vdash C : \Psi} \tag{21}$$

Machine states:

$$\frac{\vdash C : \Psi \qquad \vdash D : \Delta \qquad \vdash R : \Gamma \qquad \Psi \vdash I : (\Gamma' \times \Delta') \qquad (\Gamma \times \Delta) \leq (\Gamma' \times \Delta')}{\vdash (C, D, R, I)} \tag{22}$$

Fig. 6. Type System of sFORD

3.2 Typing Rules

We define the following types for values:

$$\text{value types } t ::= p \mid \mathsf{Const}$$

We define three typing environments:

$$\begin{array}{ll} \text{(Code heap type)} & \Psi ::= \{L_1 : (\Gamma_1 \times \Delta_1), \\ & \qquad \ldots, L_k : (\Gamma_k \times \Delta_k)\} \\ \text{(Register bank type)} & \Gamma ::= \{r_1 : p_1, \ldots, r_m : p_m\} \\ \text{(Data heap type)} & \Delta ::= \{l_1 : p_1, \ldots, l_n : p_n\} \end{array}$$

Operands that have no effect on the register allocation process are given the type Const. The type of a machine location (register or memory address) is determined by the pseudo that is stored in that location. The environment Γ contains the types of the machine registers, and the typing environment Δ contains the types of locations in the data heap. We will refer to Γ and Δ as *location environments*. The environment Ψ determines the type of each instruction block. The type of an instruction sequence is given by the minimum configuration of the bank of registers and data heap that the sequence must receive in order to be able to execute properly. We consider instructions as functions that modify the location environments, that is, an instruction i expects an environment $(\Gamma \times \Delta)$ and returns a possibly modified environment $(\Gamma' \times \Delta')$. We define an ordering on location environments as follows:

$$\Gamma \leq \Gamma' \;\; \mathsf{if}\; \forall r,\; r : p \in \Gamma' \;\mathsf{then}\; r : p \in \Gamma \tag{23}$$

$$\Delta \leq \Delta' \;\; \mathsf{if}\; \forall l,\; l : p \in \Delta' \;\mathsf{then}\; l : p \in \Delta \tag{24}$$

$$(\Gamma \times \Delta) \leq (\Gamma' \times \Delta') \;\; \mathsf{if}\; \Gamma \leq \Gamma' \wedge \Delta \leq \Delta' \tag{25}$$

The type rules for sFORD programs are given in Fig. 6. According to rule 11, the type of a register binding such as (r, p) is the temporary p, but, only if p is the type of r in the Γ environment, otherwise it does not type-check. Similarly, Rule 12 determines the type of memory bindings. Rules 13, 14 and 16 change the location environment. The ordering comparison in the premises of Rules 15 and 17 is necessary to guarantee that all the registers alive at the beginning of an instruction block have well defined types.

None of the programs in Fig. 4 type-check. In Fig. 4(a), r_1 is not declared with type p_0, thus the premise of Rule 11 is not satisfied. In Fig. 4(b), the type of r_0, before the execution of instruction 3, is p_1, not p_0, as expected by the type annotation. Again, Rule 11 is not satisfied. Fig. 4(c) presents a similar case, but using a memory location instead of a register: the type of l_0 in instruction 3 is not p_0, as expected, but p_1. The type of the used operand would not satisfy the premise in Rule 12. In Fig. 4(d), r_1 has type $\perp$ before the execution of instruction 3, which is different from the expected type p_1. Finally, in Fig. 4(e) $\Psi(L_2) = ([r_0 : p_0] \times \Delta)$, but the type of instruction 4 is $([r_0 : p_1] \times \Delta) \mapsto ([r_0 : p_1] \times \Delta)$. This would not satisfy the inequality in Rule 17.

3.3 Type Soundness

We state the lemmas and theorems that constitute our soundness proof. Our soundness proof assumes that the sMIRA program defines each pseudo p before p is used. If the sFORD program type-checks, then it preserves the values alive in the original sMIRA code.

Theorem 1. (Preservation) *If* $\vdash M$, *and* $M \rightarrow M'$, *then* $\vdash M'$.

Lemma 1. (Canonical Values) *If* $\vdash C : \Psi$, $\vdash D : \Delta$ *and* $\vdash R : \Gamma$ *then:*

1. *If* $\Psi \vdash L : (\Gamma \times \Delta)$, *then* $L \in \mathsf{domain}(C)$, $C(L) = I$ *and* $\Psi \vdash I : (\Gamma \times \Delta)$.
2. *If* $\Delta; \Gamma \vdash o : p$, *then* $o = r$, *or* $o = l$. *If* $o = r$, *then* $r \in \mathsf{domain}(R)$, *else if* $o = l$, *then* $l \in \mathsf{domain}(D)$.
3. *If* $\Delta; \Gamma \vdash o : \mathsf{Const}$, *then* $o = \bullet$.

Theorem 2. (Progress) *If* $\vdash M$, *then* M *is a final state, or there exists* M' *such that* $M \mapsto M'$.

Corollary 1. (Soundness) *If* $\vdash M$, *then* M *cannot go wrong.*

We have checked the proof using Twelf [32], and this proof can be found at `http://compilers.cs.ucla.edu/ralf/twelf/`

3.4 Preservation of callee-save registers

Our type system is intra-procedural, and it assumes that a function call preserves callee-save registers. This must be verified for each procedure, after its type-checking phase, when every instruction has a well know type. If we assume that a machine register is either caller-save or callee-save, this verification step can be done via a simple test. Let L_0 be the label of the first instruction in the procedure, and let L_e be an exit point. Let $\Psi(L_0) = (\Gamma_0 \times \Delta_0)$, and let $\Psi(L_e) = (\Gamma_e \times \Delta_e)$. Callee-save registers are preserved at exit point L_e, if $\Gamma_e \diamond \mathsf{CallerSave} \leq \Gamma_0 \diamond \mathsf{CallerSave}$.

Along with the preservation of callee-save registers, our type systems has a set of consistency requirements, which can be carried out as a sequence of table lookup verifications. These checks are explained in the full version of this paper, available at `http://compiler.cs.ucla.edu/ralf`.

4 Experimental Results

Figure 1 presents a high-level block diagram of RALF. RALF interfaces the transformation between the RTL and LTL intermediate representations used by gcc. RALF's front end consists mainly of gcc's parser, gcc's optimization phases, and code to produce a MIRA program from a RTL program. The back end consists mostly of a type checker, code for producing LTL instructions, and gcc's code generation engine. RALF interacts with the implementation of a register

allocator via ordinary ASCII files. Given a RTL program P, RALF translates P into a MIRA ASCII program $\mathcal{M}$, which is then fed to the plugged-in register allocator. The register allocator outputs a set of FORD directives $\mathcal{F}$, which are then given back to RALF. RALF checks that $(\mathcal{M}, \mathcal{F})$ is a correct mapping via the type system described in Section 3, applies the directives $\mathcal{F}$ on the original RTL program, and generates gcc's LTL code. RALF's back end does not need the original MIRA file in order to produce the LTL program. When producing the RTL code, RALF inserts loads and stores for callee save registers at the entrance and exit of each function (a smart register allocator might decide to take on that responsibility itself and RALF has an option for that).

We have tested RALF with eight different register allocators: (1: gcc -O2) the allocator present in the gcc compiler, which has two main phases: (a) aggressive register allocation for local variables within basic blocks, (b) conservative allocation for the whole function. (2: Naive) The *naive* register allocator, which spills all the pseudos. (3: UBC) usage count based register allocator [11], (4: IRC) iterated register coalescing [13], (5: Chordal) register allocation via coloring of chordal graphs [27], (6: LS) linear scan [28], (7: RA) integer linear program (ILP) [23], (8: SARA) stack location allocation combined with register allocation (SARA) [23].

Five of the register allocators have been implemented in Java (2, 3, 4, 5 and 6). Algorithms 7 and 8 have been implemented in AMPL [10]. The interface provided by RALF is extremely simple, and most of the code used to parse and output MIRA/FORD files could be reused among the different implementations. Table 7 compares the size of each implementation; (J) stands for Java, and (A) for AMPL. We do not compare the execution speed of the allocators, because they have been implemented in different languages.

RA	#LOC	
	RA	Interface
Naive	48 (J)	773 (J)
UBC	2766 (J)	773 (J)
IRC	3538 (J)	773 (J)
Chordal	4134 (J)	773 (J)
LS	385 (J)	1100 (J)
RA	495 (A)	298 (A)
SARA	731 (A)	400 (A)

Fig. 7. Comparison between different register allocators plugged on RALF

We have plugged each of the eight algorithms into RALF and then tested the produced code on a StrongARM/XScale processor, with 64MB SDRAM, and no cache. We have drawn our benchmark programs from a variety of sources. We chose these benchmarks in part because the more traditional ones (for example,

SPEC) have a huge memory print, and cannot be run in our resource constrained ARM hardware.

	LOC	RTL	gcc-O2		LS		RA		SARA		Chordal		IRC		UCB		Naive	
Bench			mem	csr	mem	csr	mem	csr	mem	csr	mem	csr	mem	csr	mem	csr	mem	csr
Stanford	307	1082	20	81	171	107	22	63	24	69	34	134	70	100	44	133	854	0
yacr2	3979	10838	1078	289	3035	335	1003	123	1109	142	2121	357	1957	314	2200	361	8181	0
ft	2155	3218	299	130	538	151	225	87	230	106	371	162	561	100	360	169	2184	0
c4	897	40948	187	123	715	301	176	145	179	151	416	170	453	145	394	170	3531	0
mm	885	3388	386	92	2494	68	375	116	380	92	591	93	648	93	687	93	2590	0
url	652	1264	102	54	313	16	120	56	120	58	155	61	201	51	203	63	860	0
md5	790	3464	519	110	1869	433	500	120	500	120	570	228	697	174	580	229	2714	0

Fig. 8. Compile time statistics.

- Stanford Benchmark suite: a collection of seven programs that test recursive calls and array indexing.
- NetBench [19]: url is a network related benchmark that implements HTTP based switching; md5 is a typical cryptographic algorithm.
- Pointer-intensive benchmark [7]: This benchmark suite is a collection of pointer-intensive benchmarks. Yacr2 is an implementation of a maze solver and Ft is an implementation of a minimum spanning tree algorithm.
- c4 and mm are taken from the comp.benchmarks USENET newsgroup at http://www.cs.wisc.edu/~thomas/comp.benchmarks.FAQ.html. c4 is an implementation of the connect-4 game, and mm is a matrix multiplication benchmark.

The number of lines of C code (LOC) and the number of instructions in the RTL for these benchmarks are presented in Figure 8. These benchmarks are non-floating point programs. (We had to edit few of the programs 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, Figure 8 presents two static compile time statistics: the number of memory accesses (mem) due to spill/reload instructions, and the number of callee save registers (csr) used by the register allocator (leads to more memory accesses).

The chart in Figure 9 compares the execution times of the programs produced by each of the register allocators. The execution times have been normalized against the time obtained by programs compiled with gcc at the -O2 optimization level. It can be noted that, although the gcc algorithm is heavily tuned for the StrongARM architecture, Chordal, UBC, and IRC present comparative performances. Also Chordal and IRC's performances are similar, which confirms the results found by Pereira et al [27]. Figure 9 suggests an upper limit on the gains that any register allocator can make. Even in the most extreme case, the code generated by the naive allocator is worse by a factor of 2.5 (as compared to the optimal solution found by the ILP based allocator). An important point

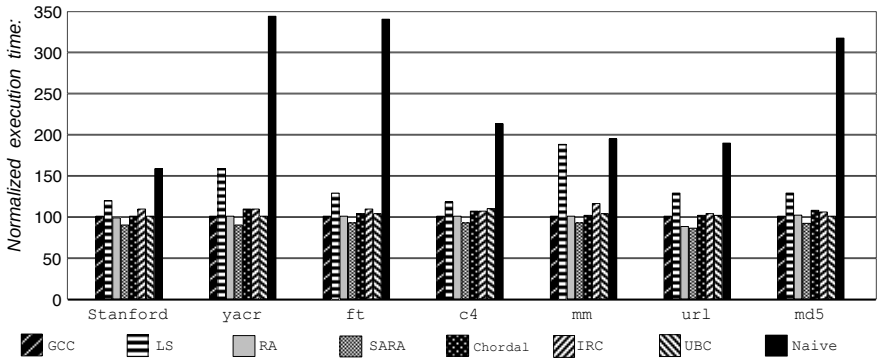

Fig. 9. Comparison of different register allocators using execution time of benchmarks as the metric.

is that most of these benchmarks deal with structures and arrays that require compulsory memory accesses, and it seems that these accesses overshadow the spill cost and hence such a small (2.5 times) improvement. Another conclusion is that the obtained execution time given by optimal solutions (SARA and RA), that run in worst-case exponential time, is not much lower than that obtained by polynomial time heuristics. It should be pointed that our experiments compare specific implementations of the allocators, and are run on a specific target machine; however, it was our objective to be as faithful as possible to the original description of each algorithm.

5 Conclusion

We have presented a framework that facilitates the development of register allocation algorithms. Our framework consists of the two description languages MIRA and FORD, plus a type system. Our framework is easy to use, and versatile enough to support a wide variety of register allocation paradigms. In order to validate this claim, we have developed RALF, an implementation of our framework for the StrongARM architecture, and used it to compare eight different register allocators.

Our framework has several limitations which may be overcome in future work: FORD does not permit the register allocator to modify the control flow graph of the target program; the grammar of MIRA supports only intra-procedural register allocation; the validation algorithm does not handle bitwidth aware register allocations [29]; and FORD does not support the concepts of re-materialization or code-motion. The implementation of RALF itself has the limitations that it targets only the ARM architecture, and RALF does not handle pseudos of type float or double (we have opted for this restriction to keep the implementation simple). So far we have experimented with only the C front-end of gcc; RALF

can be seamlessly used with any language supported by gcc. Currently, we are extending RALF to allow the register allocator to do bitwidth-sensitive-analysis.

RALF, our benchmarks, our Twelf proof, and a collection of tools that we have developed to aid in the design and test of register allocators are publicly available at `http://compilers.cs.ucla.edu/ralf`.

5.1 acknowledgments

We thank the anonymous reviewers for comments on a draft of the paper. Fernando Pereira is sponsored by the Brazilian Ministry of Education under grant number 218603-9.

References

1. Standard ML of New Jersey. 2000. `http://www.smlnj.org/`
2. GNU C compiler. 2005. `http://gcc.gnu.org`
3. Ali-Reza Adl-Tabatabai, Thomas Gross, and Guei-Yuan Lueh. Code reuse in an optimizing compiler. In *OOPSLA*, pages 51–68. ACM Press, 1996.
4. Johan Agat. Types for register allocation. *Lecture Notes in Computer Science*, 1467:92–111, 1997.
5. Christian Andersson. Register allocation by optimal graph coloring. In *CC*, pages 34–45. Springer, 2003.
6. Andrew W Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI*, pages 243–253. ACM Press, 2001.
7. Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, pages 290–301, 1994.
8. G. J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982.
9. K.M. Elleithy and E.G. Abd-El-Fattah. A genetic algorithm for register allocation. In *Ninth Great Lakes Symposium on VLSI*, pages 226–227, 1999.
10. Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL A modeling language for mathematical programming.* Scientific Press, 1993. `http://www.ampl.com`.
11. R. A. Freiburghouse. Register allocation via usage counts. *Commun. ACM*, 17(11):638–642, 1974.
12. Changqing Fu and Kent Wilken. A faster optimal register allocator. In *MICRO*, pages 245–256. IEEE Computer Society Press, 2002.
13. Lal George and Andrew W. Appel. Iterated register coalescing. *TOPLAS*, 18(3):300–324, May 1996.
14. David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *SPE*, 26(8):929–968, August 1996.
15. 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.
16. Yuqiang Huang, Bruce R. Childers, and Mary Lou Soffa. Catching and identifying bugs in register allocation. In *SAS*. Springer, 2006.

17. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54. ACM Press, 2006.
18. 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.
19. G. Memik, B.Mangione-Smith, and W.Hu. Netbench: A benchmarking suite for network processors. *IEEE International Conference Computer-Aided Deisgn*, November 2001.
20. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *TOPLAS*, 21(3):527–568, 1999.
21. Mayur Naik and Jens Palsberg. Compiling with code size constraints. *Transactions on Embedded Computing Systems*, 3(1):163–181, 2004.
22. V. Krishna Nandivada and Jens Palsberg. Efficient spill code for SDRAM. In *CASES*, pages 24–31, 2003.
23. V. Krishna Nandivada and Jens Palsberg. Sara: Combining stack allocation and register allocation. In *CC*, pages 232–246, 2005.
24. George C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–95. ACM Press, 2000.
25. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *PLDI*, pages 333–344, 1998.
26. Atsuchi Ohori. Register allocation by proof transformation. *Science of Computer Programming*, 50(1-3):161–187, 2004.
27. Fernando M. Q. Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *ASPLAS*, 2005.
28. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *TOPLAS*, 21(5):895–913, 1999.
29. Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. In *POPL*, pages 85–96, 2003.
30. Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *PLDI*, pages 142–151, 1998.
31. Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, 1999.
32. Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In (CADE-16), pages 202–206, Springer, 1999.