1994

Register Allocation for Accurate Garbage Collection of C++

S. Satishkumar

Register Allocation for Accurate Garbage Collection of C++

Abstract

Register Allocation for Accurate Garbage Collection of C++ S. Satishkumar* M.S. Creative Component Accurate garbage collection of C++ requires that every memory location and every register be known to contain either a pointer or a non-pointer. In order to minimize the run-time overhead of tagging memory locations and registers, techniques for partitioning memory and registers into separate classes dedicated independently to the representation of pointers and non-pointers respectively have been developed. This paper describes the implementation and performance of a specially designed register allocator for the GNU g++ compiler. * Portions of this paper were excerpted from "Code Generation to Support Efficient Accurate Garbage Collection of C++ on Stock Hardware", a paper currently being prepared for publication by Kelvin Nilsen, Ravichandran Ganesan, Satish Guggilla, Satish Kumar, and Kannan Narasimhan

Disciplines

Systems Architecture | Theory and Algorithms

Register Allocation for Accurate Garbage Collection of C++

S. Satishkumar[1]

Department of Computer Science
Iowa State University
Ames, IA 50011

ABSTRACT

Accurate garbage collection of C++ requires that every memory location and every register be known to contain either a pointer or a non-pointer. In order to minimize the run-time overhead of tagging memory locations and registers, techniques for partitioning memory and registers into separate classes dedicated independently to the representation of pointers and non-pointers respectively have been developed. This paper describes the implementation and performance of a specially designed register allocator for the GNU g++ compiler.

1. Introduction

Accurate garbage collection of C++ improves programmer productivity, simplifies the development of reusable software components, and offers the potential of supporting more reliable and predictable dynamic memory management performance than is possible using traditional C++ dynamic memory management techniques. But previous implementations of accurate C++ garbage collectors require special hardware and incur a run-time overhead of up to 30%, even with the aid of the special hardware support. Further, the only previous implementation of C++ garbage collection [10] does not make use of traditional code optimizations, including global register allocation, because of bugs in the compiler's optimizer.

In this paper, we describe the garbage-collection compatible techniques that we have developed to allow efficient global register allocation and other optimizations to be performed by the GNU GCC code generator.

Related Work

Most of the previous research on code generation techniques to support garbage collection has focused on languages designed to incorporate garbage collection. In these languages, clean semantic models and cooperative implementation techniques make the implementation of efficient garbage collection much easier than in languages such as C and C++ which were not originally designed to make use of automatic garbage collection. As this paper describes garbage collection of C++, our survey of related work concentrates on code generation techniques for uncooperative languages.

Existing literature describes only two alternative techniques for garbage collection of the full C++ language. Conservative garbage collection, designed by Boehm and his colleagues, works, for the most part, with traditional C and C++ compilers by assuming that every word of memory may contain a pointer. It treats as a pointer any register or memory cell containing a value that is a valid heap memory address [1].

[1] Portions of this paper were excerpted from *Code Generation to Support Efficient Accurate Garbage Collection of C++ on Stock Hardware*, a paper currently being prepared for publication by Kelvin Nilsen, Ravichandran Ganesan, Satish Guggilla, Satish Kumar, and Kannan Narasimhan.

Conservative garbage collection has been shown to provide good time and space performance on a wide variety of real-world applications [12]. The alternative available garbage collection technique for C++ is accurate garbage collection, as described in references 9 and 8. Accurate garbage collection requires cooperation from the compiler's code generator in order to distinguish pointers from non-pointers within memory and registers.

Though the design of conservative garbage collection was intended to avoid the need for modifications to compiler code generators, a number of traditional code optimization techniques have been found to be incompatible with conservative garbage collection. For example, certain induction variable optimizations may convert the only pointer to a particular heap object into an integer offset relative to some other pointer (which refers to a different heap object). If garbage collection is triggered while the loop is executing, the garbage collector will not recognize the referenced object as live. Another example of problematic code generation results from addition of a pointer variable and a large integer constant. This may translate into a two-instruction sequence, the first instruction of which adds a constant to the high-order bits of the pointer so that the resulting value points beyond the borders of the referent object. Boehm and Chase have studied a number of different compilers and have characterized the sorts of problems that their code generation strategies present to the conservative garbage collection technique. They have addressed these problems in reference 2.

Since conservative garbage collectors are never sure of whether a particular value is a pointer or a non-pointer, they are not able to relocate objects in order to reduce memory fragmentation. This is because a relocating garbage collector needs to update all pointers to reflect the new locations of the objects they refer to. If a particular word that resembles a pointer is being used by the program as a pointer, then it would be acceptable for the garbage collector to move the referent object and modify the pointer. But suppose the memory really represents an integer which just happens to hold a value that resembles a heap address. If the conservative garbage collector modifies this word as a consequence of relocating the referenced object, then the integer's value becomes corrupted. Thus, relocating garbage collectors need to be accurate garbage collectors, in the sense that they must have full knowledge of exactly which memory cells and registers represent pointers.

In reference 3, Diwan, Moss, and Hudson of the Object Systems Laboratory at the University of Massachusetts at Amherst describe the modifications they made to the GNU code generator in order to support relocating garbage collection of Modula-3[2]. Modula-3, unlike C++, was designed to cooperate with automatic garbage collection. Thus, Diwan's techniques do not generalize directly to garbage collection of C++. Nevertheless, it is instructive to consider the methods Diwant has developed. Diwan's garbage collector requires a base pointer to the beginning of every heap object. Pointers that refer to internal fields of heap objects must be converted into base pointers in order for the garbage collector to process them. Like Boehm and Chase, Diwan points out that standard compiler optimizations may result in pointers that do not point directly to the bases of objects, and in pointers that refer to addresses beyond the boundaries of the objects that they belong to. Diwan calls these derived pointers, because they are derived from the base pointers by some sequence of arithmetic operations. Unlike Boehm and Chase, Diwan needs to be able to accurately identify every pointer in the system, and must be able to map every pointer to the base address from which it was derived.

Diwan's technique is to select *gc-points* within each function, which identify control points at which garbage collection might begin. At each gc-point, the compiler builds a number of tables that characterize the contents of all registers and activation frame slots at that execution point. Two of these tables identify which registers hold pointers, and which activation frame slots hold pointers, respectively. A third table provides formulas for calculating base pointers from the derived pointers held in certain registers and stack locations. In order to enable these calculations, it is occasionally necessary to extend the lifetimes of certain variables beyond the point at which a traditional register allocator would consider them dead.

[2] The GNU code generator serves as a portable back-end for a number of different languages, including C, C++, Smalltalk, and Modula-3.

The increased register pressure may result in worse generated code under some circumstances, but Diwan reports that they found no differences between the optimized code generated by the traditional compiler and the code generated by the revised system for the benchmark applications that were evaluated in reference 3.

Previous work by Schmidt and Nilsen differs from Diwan's work in that the target language is lower level than Modula-3 [9, 10]. Since the target language is C++, there is no hope of tracking base pointers for all heap-allocated objects. In C++, derived pointers may be assigned to programmer-defined variables and passed as arguments to other functions. Therefore, the garbage collector (rather than the compiler) takes responsibility for computing base pointers from derived pointers. Special circuitry provides the functionality in the hardware-assisted real-time garbage collection system [7]. In the stock-hardware garbage collection system, this functionality is implemented in software [5]. Thus, Schmidt's compiler did not need to preserve base pointers, nor did it need to find ways to compute base addresses of objects from derived pointers. Another difference between Schmidt's compiler and the other two studies cited in this section is that Schmidt's compiler did not support optimizations.

Terminology

In the discussion that follows, we use the term *descriptor* to denote a pointer. By pointing to objects allocated elsewhere, each descriptor is capable of "describing" all conceivable kinds of information. We use the adjective *terminal* to characterize memory locations known not to contain pointers. If all live memory is represented as a directed graph in which nodes represent dynamically allocated objects and directed edges represent pointers from one object to another, the terminal nodes are those from which no directed edges emanate. The source nodes in this directed graph are pointers residing outside the garbage-collected heap. These source pointers, which are under direct control of the CPU, are called *root descriptors*.

2. Sparc Code Generation in the GNU GCC Compiler

We have used the implementation of GNU CC as a foundation upon which to build a customized compiler which supports automatic garbage collection of the C++ heap. We have focused on the SPARC architecture, though the techniques we have used can be generalized to other targets in the future. Most of the target-dependent code and data structures are found in the following two files:

md: md is an abbreviation for machine description. This file contains an abstract representation of all the instructions supported by the target machine.

sparc.h:
This file provides definitions of data structures and macros which convey information about the target architecture. These macros are accessed from various parts of the compiler.

Most of the work performed by the compiler is done using an intermediate representation called the *register transfer language* (*rtl*). Within the md file, machine instructions are represented as rtl patterns. The compiler's code generator compares instruction patterns against rtl expressions (*rtx*) in order to select efficient translations of the rtx representations of the source code that it is translating. For a more complete description of the above files and of rtl, see the GNU CC manual [11].

Registers in the SPARC Architecture

SPARC describes an architecture with many possible implementations. The SPARC architecture uses register windows. An implementation of SPARC may have from 40 to 520 general purpose registers, called r registers, of which only 32 registers are visible in the register window at any one time. Additionally, the SPARC supports 32 floating point registers, called f registers, which are not windowed. The 32-register window is comprised of 8 global registers, 8 local registers, 8 input registers, and 8 output registers. The register window shifts with each function invocation. When it shifts, the output registers of the current window became the input registers of the new window. This provides an efficient mechanism for parameter passing through registers. The current function's local and input registers are shifted out of the window, and a set of 16 registers is allocated to represent the new window's local and output registers. The global registers are shared between all functions. The figure below illustrates the abstract organization of register windows within the SPARC system:

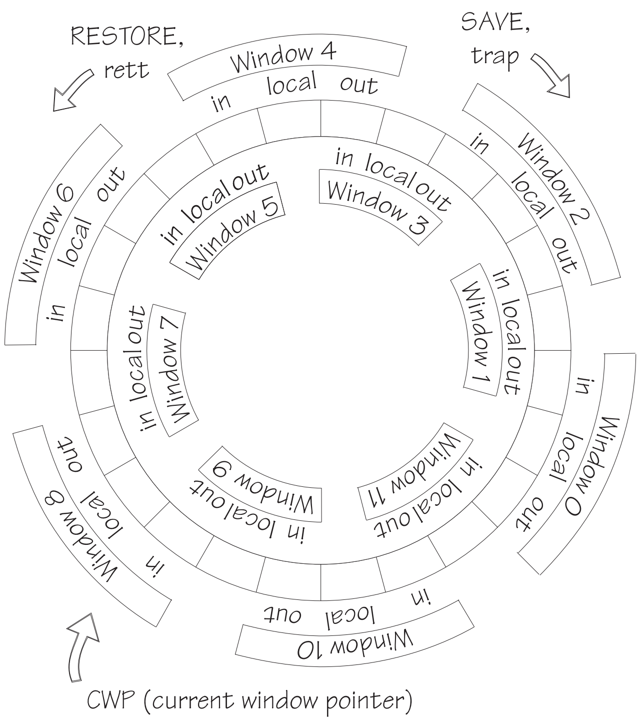

Figure 2.1: SPARC Register Windows

Note that deeply nested function calls may overflow the register window hardware. Each function's activation frame reserves space on the run-time stack to hold its sixteen local and input registers in case the register window allocator needs to reuse a particular register window frame. By convention, the i6 register points to the function's activation frame, and the register window save area is always found at the same fixed offset relative to the beginning of the activation frame. Thus, the trap handlers for register window overflow (and underflow, which occurs when returning to a function whose register window has been written to memory) know the precise mappings between particular register windows and the memory regions to which the register windows map.

Windowed Register Address	*Register Names*
global[0] − global[7]	r0 − r7
out[0] − out[7]	r8 − r15
local[0] − local[7]	r16 − r23
in[0] − in[7]	r24 − r31

By convention, parameters are passed in the out registers of the caller, which become the in registers of the callee. However, if there are more parameters to be passed than fit within these registers, the extra parameters are passed on the stack. Also by convention, structures are passed using an implicit level of indirection. The caller reserves space for the structure argument within its activation frame, and passes a pointer to the structure's location as an argument to the called function. Refer to the SPARC architecture manual for a more detailed description of the SPARC architecture [6].

The genoutput program

The GNU compiler's code generator is designed to be portable between a large number of target architectures. Within the GNU compiler, much of the machine-independent analysis and optimization is performed on an intermediate code called *rtl*. The C type of an expression written in rtl is declared *rtx*.

The machine-specific code-generation details are isolated within a small number of configuration files. The file md holds the machine description. This description includes, among other things, patterns describing the functionality provided by each of the host architecture's machine instructions. The genoutput program is a utility that reads the md file and produces C code to be incorporated into the compiler's code generator. genoutput is compiled from the source file genoutput.c. The output from genoutput is the C source file insn-output.c, which forms part of the compiler.

The insn-output.c file contains arrays that provide information about the various instruction templates found in the md file. The code generator matches rtl expressions against the machine instruction templates in order to select efficient translations. The tables declared in insn-output.c represent the instruction names, constraints on operands, and the costs of particular machine-instruction translations of particular rtl instructions.

Some of the arrays constructed as part of the file insn-output.c are as follows:

insn_n_operands []
: As instructions are read in from the md file, they are assigned instruction code numbers. Instruction code numbers are assigned in increasing order, starting with zero for the first instruction. insn_n_operands is an array of integers, indexed by an rtl instruction code number. insn_n_operands[i] is the number of operands required by instruction i.

insn_operand_constraints [][]
: This is a two-dimensional array of operand constraint strings. An operand constraint string describes the constraints imposed on each operand. For example, certain instructions require the operand to be in a particular class of registers. Other instructions allow the operand to reside in memory. This array is indexed by the machine instruction number in the first dimension, and the operand number in the second.

insn_n_alternatives []
: insn_n_alternatives[i] is the number of constraint alternatives associated with the operands of instruction whose code number is i.

genoutput's main() function reads the machine description file, passing each instruction template to the scan_operands() function. scan_operands() analyzes the instruction, checking for internal consistency, and storing its representation in global data structures. These data structures are declared globally so that the output_epilogue() function can examine their contents. output_epilogue() has the responsibility of outputting the various tables discussed above to the insn-output.c file. A pseudo-code prototype for scan_operands() is provided below:

```
static void scan_operands(part, this_address_p, this_strict_low)
  part: The rtx template scanned.
  this_address_p: Boolean indicating if the containing rtx is
                  an address.
  this_strict_low: Boolean indicating if the containing rtx is a
                  strict low part.
```

A strict-low instruction is an instruction that updates only the low-order bits of a particular destination pseudo-register while ensuring that the higher-order bits remain unchanged.

Register Allocation

When generating intermediate code (rtl), the GNU CC compiler assumes the existence of an unlimited number of registers. These registers are called pseudo-registers. The job of the register allocator is to map pseudo-registers to real machine registers, which are called hard registers.

The GNU C/C++ register allocator supports two modes of register allocation:

1. Stupid (naive) register allocation is based on very minimal dataflow analysis. This is the register allocator used by default, when optimization is not explicitly enabled on the gcc command line.
2. Optimized register allocation is based on detailed data-flow analysis in order to perform efficient local and global register allocation.

The following sections of this paper describe the various phases of analysis that are performed as part of the register allocation effort.

Register Scan

This phase of the compiler is used to find the first and last use of every pseudo-register. This information is recorded in the vectors regno_first_uid and regno_last_uid, which are indexed by pseudo-register number. The value represented by regno_first_uid[r] is the unique integer identifier (*uid*) associated with the first instruction that uses register r. Similarly, the value represented by regno_last_uid[r] is the uid of the last instruction using register r.

These vectors are essential for performing jump optimizations, loop optimizations and common subexpression elimination. These phases are postponed until after the register scan phase has been performed.

The source code for register scanning is found in the file regclass.c. The entry point for this pass is the function reg_scan(), whose prototype is shown below:

```
void reg_scan (f, nregs, repeat)
  f: Chain of rtl instructions
  nregs: No of pseudo-registers
  repeat: Zero value indicates first entry to the function
```

reg_scan performs initialization and then calls the function reg_scan_mark_refs(), shown below, for each rtl instruction.

```
void reg_scan_mark_refs(x, insn)
  x: The pattern of the rtl instruction
  insn: The rtl instruction itself
```

This function comprises the main portion of the work performed in this pass: it obtains and records the regno_first_uid and regno_last_uid information for each register by examining each rtl instruction.

Register Class Preferencing

Many machines support a variety of register classes. For example, modern processors are known to have address registers, data registers, floating-point registers, index registers, and base registers. Certain machine instructions require their operands to be registers of a particular class. Accordingly, the registers are classified into various categories called register classes. And some registers may belong to several classes. The class ALL_REGS contains all the hard registers defined for the machine. The class GENERAL_REGS is assumed to be defined for all machines. If an operand allows multiple classes, the union of these classes should itself be defined as a named class in the abstract model for the machine. Otherwise, the code generator may obtain an invalid class for an operand during its computation of instruction costs.

Register class preferences are specified in the md file. Each instruction's pattern template is accompanied by operand constraints that specify the allowed register classes for each operand. The cost of moving data between register classes and between registers and memory is also defined within the sparc.h file, and these costs are consulted by the instruction matcher in order to obtain the least-cost translation of rtl to machine code. The cost of assigning a pseudo-register to a hard register of a particular class is calculated as the the sum of the costs associated with the use of this operand in all of the instructions that make reference to this particular pseudo-register.

The register-class pass of the compiler is used to compute the preferred class in which to allocate each pseudo-register. This information is used in the register allocation routines to find a suitable hard register for a pseudo-register. Its code is found in the file regclass.c. The entry point of the pass is the function regclass().

The instructions are scanned and the cost of putting a pseudo-register in different register classes is computed. Savings resulting from putting a register in a certain class are recorded. Instructions inside a loop are assumed to be executed three times more frequently than instructions outside. The various alternatives suggested in the register constraints in rtl are considered. The cost of using each of the alternatives is computed assuming that the registers are in the suitable class. Then, the cost of placing the registers in the right class is computed. Thus the total cost of the alternative is obtained. The alternative with the best cost is utilized.

For each register, we compute how desirable each class is. The most preferred class is recorded in prefclass[regno]. The alternate class is also recorded in altclass[regno]. The alternate class is a class whose use is preferable to using memory. In the absence of such a class, NO_REGS is recorded.

These values are accessed using the functions prototyped below:

```
enum reg_class reg_preferred_class (regno);
enum reg_class reg_alternate_class (regno);
```

If these functions are called before the arrays prefclass[] and altclass[] are computed, then GENERAL_REGS is returned.

Stupid Register Allocation

''Stupid register allocation'' is invoked by default (when the -O optimize option is not provided on the command line). It can be explicitly invoked by providing the -noreg switch on the gcc command line. Stupid register allocation does very limited data-flow analysis. Only programmer-declared register variables and compiler-generated temporaries are placed in registers. All other variables are assigned stack slots instead of pseudo-registers during the generation of rtl intermediate code.

When performing stupid register allocation, the register-scan phase is omitted. However, the register-class pass of the compiler is still performed, because the compiler must determine which class of register to assign to each programmer-declared register variable.

The lifetime of programmer-defined register variables begin at the variables' declarations and end at the end of the enclosing function, even for register variables declared within nested blocks. This interpretation simplifies analysis in the absence of the register-scan phase. Other pseudo-registers are said to die at the point of last reference.

After the lifetimes of pseudo-registers are computed, the pseudo-registers are prioritized according to length of the lifetime, the number of references to the pseudo-register, and the register number, in that order. The register number is of no particular significance; it serves only to resolve prioritization ties between pseudo-registers that happen to have identical lifetimes and references. Hard registers are assigned to pseudo-registers in order of decreasing priority.

The entry point for stupid register allocation is the function `stupid_life_analysis`, prototyped below:

```
void stupid_life_analysis (f, nregs, file)
  f: Chain of instructions
  nregs: Number of pseudo-registers allocated
  file: File for outputting debugging dump
```

Local Register Allocation

When code optimization is enabled, allocation of hard registers to pseudo-registers is done in two passes. Local allocation is performed first, to allocate pseudo-registers whose lifetimes are contained entirely within particular basic blocks. Then, global allocation is performed, to assign hard registers to pseudo-registers that span multiple basic blocks. Generally, local register allocation is simpler than global register allocation. The pseudo-to-hard register mappings are recorded in a vector `reg_renumber[]`. `reg_renumber[N]` is the hard register assigned to pseudo-register `N` or −1 if none has been assigned. `reg_renumber[N]` is `N`, if `N` is a hard register number. In a subsequent code generation phase, the rtl code is modified to reflect the physical resources (hard registers or slots in the stack activation frame) assigned to each pseudo-register.

When the compiler detects that one pseudo-register's lifetime ends at the same moment that another pseudo-register's lifetime begins, and the second pseudo-register receives its initial value by copying the contents of the first, the register allocator groups the two (or possibly more) pseudo-registers into a *quantity*. The quantity represents a block of pseudo-registers, all of which are to be assigned to the same hard register if possible. The compiler refrains from grouping pseudo-registers with different register class preferences into a single quantity. Quantities are prioritized for register allocation in terms of the following criteria: duration of lifetime (shorter lifetimes are preferred over longer lifetimes), number of references to the pseudo-registers represented by the quantity, and the unique integer assigned to identify the quantity.

The function `local_alloc()`, shown below, is the entry point for local register allocation.

```
void local_alloc() {

  update_equiv_regs();
  // Reduce the priority of registers which maintain a constant
  //  value throughout.  If it has a single reference, replace the
  //  register by the value.

  Determine the registers which are used only in a single block;
  // These are eligible for allocation here if their preferred
  //  register set has more than one register.  This restriction is
  //  present as at least one register of the class needs to be
  //  reserved for use as a reload register.

  Perform the allocation for each basic block;
```

```
    block_alloc();

    Scan the basic block instructions tying registers into quantities
      whenever appropriate.
    // Instead of assigning the same quantity for a hard register and a
    //   pseudo-register, specify the hard register as a suggested allocation.

    Prioritize the quantities, and allocate registers to them
    // For each quantity, try to allocate a suggested physical
    //   register for it.  If this fails, attempt to find a suitable
    //   register from the preferred class (and if this fails,
    //   from an alternate class).

    // Upon completion, the mapping of every pseudo-register to a hard
    //   register is set in reg_renumber[], based on the physical register
    //   suggested for its quantity.
}
```

Global Register Allocation

This pass of the compiler is responsible for performing the allocation of hard registers to those pseudo-registers which have not been accounted for by the local allocation phase.

Again, the allocations are recorded in the array reg_renumber[]. Preparatory to performing global register allocation, the compiler assigns *allocation numbers* to all of the pseudo-registers that have not yet been allocated hard registers, with a unique allocation number assigned to each pseudo-register. A mapping between pseudo-registers and allocation numbers is provided by the C arrays reg_allocno and allocno_reg. Allocation numbers are used to simplify the global allocation process. allocno_reg[allocno] holds a pseudo-register to be allocated for every valid allocation number (to be valid, the allocation number must be less than a particular maximum value). The allocation number can index into the allocno_order array, which specifies the desired order of allocation, and into the allocno_size array, which gives the number of consecutive hard registers required by the corresponding pseudo-register. All processing is performed using the allocation numbers, and these are converted to pseudo-register numbers only at the end of this phase.

Note that local register allocation did not make any changes to the rtl code. Thus, the global register allocator is free to undo a local allocation if it decides that it is able to make better use of a particular hard register than did the local allocation. Whenever the global allocator changes local allocations, it updates the reg_renumber[] array to reflect the modification. After local and global register allocation have both completed, the compiler performs a reload pass which modifies the rtl code to properly handle any register spills that are created by the global allocator.

The entry point for global register allocation is the function global_alloc(). Its pseudo-code is supplied below:

```
int global_alloc(file) {

    Create a list of hard registers already in use (either assigned by
      local allocation, or possibly busy because they are specified to
      be callee saved)
    Find pseudo-registers not allocated so far, establish mappings
      between allocation numbers and pseudo-registers.

    Track usage of hard registers allocated to pseudo-registers by the
      local allocator.
    //  This information is used later to nullify an
    //    assignment made by local allocation if a more efficient
    //    utilization can be made by global allocation.

    global_conflicts();
    //  Scan the rtl code and compute conflicts between allocnos and
    //    between allocnos and hard registers.  A conflict indicates that the
    //    respective entities are live at the same time and hence cannot
    //    remain in the same hard registers.

    Obtain register preferences.

    expand_preferences();
    //  If one allocation number dies in an instruction and another is
    //    set in the same one, the register preferences are merged.

    Determine an order for allocating the remaining pseudo-registers.
    //  The priority is computed similar to the local allocation pass.

    prune_preferences();
    //  From the list of preferred registers, remove those registers
    //     which cannot be used due to difference in register types or due to
    //     conflicts.  Compute the preferences of each conflicting lower
    //     priority quantity and attempt to avoid its preferences.  The intention
    //     is to provide lower priority allocation numbers the preference
    //     for these registers.

    find_reg();
    //  Attempt to allocate registers using the preferred class, and if this
    //     fails use the alternate class.

    reload():
    //  Call the reload case to handle spills and ensure correctness of the
    //     resulting rtl code.
  }
```

Reload Pass

The reload pass of the compiler is executed after both register allocation passes have completed. The reload pass is responsible for ensuring the validity of each instruction. It checks that an operand which needs to be in a register is in fact in a register of the correct class. Invalid instructions are fixed up by temporarily copying values into registers as needed.

At the beginning of this pass, the rtl instructions still make reference to pseudo-registers. The mapping to hard registers is found in the array reg_renumber[]. The reload pass modifies the rtl by replacing pseudo-registers with the corresponding hard registers.

Those pseudo-registers that are not assigned to hard registers are assigned instead to memory slots within the function’s activation frame. Since RISC architectures require instruction operands to reside in registers, the register allocators refrain from allocating all of the target machine's registers. A small number of registers of each register class is set aside to serve as reload registers. Whenever a value which was allocated to an activation-frame slot is required as an operand, the reload pass modifies the rtl code to reflect the need to copy the operand from the stack slot into one of the reload registers. In some cases, it is necessary for the reload pass to copy (spill) the previous contents of a reload register to some other hard register or to a stack slot prior to using the it as a reload register.

The source code for the reload pass is found in two files: reload.c and reload1.c. The entry point for the reload pass is the function reload(), whose prototype is shown below:

```
int reload (first, global, dumpfile)
  first: First instruction in the linked list of instructions representing
           this function.
  global: Non-zero value indicates that reload was called from
           global_alloc, (not stupid allocation) so we have
           sufficient information to reallocate spilled
           pseudo-registers.
  dumpfile: Used for debugging dump.
```

3. Design and Implementation of the Default Partition

In order to allow accurate garbage collection to be triggered at arbitrary execution points (as might be necessary in a multi-threaded C or C++ execution environment), the run-time system must always be prepared to distinguish between registers that hold pointers and those that don't. This is required so that the garbage collection can use this information at the time of a flip. We satisfy this need by reserving a subset of the registers exclusively for descriptors (pointers), using the remaining registers to represent terminal data (non-pointers). Support for this partitioning of registers required changes to a large number of different parts of the compiler.

3.1. Register Classes

There are two main classes of registers provided in the original implementation of the gcc backend, namely GENERAL_REGS which comprise the general purpose registers r0 − r31, and FP_REGS, which comprises the floating point registers. Supersets and subsets of the different register classes are also provided. For example, the symbol ALL_REGS represents the union of GENERAL_REGS and FP_REGS.

In our backend implementation, we have created two subclasses of GENERAL_REGS. They are named ADDR_REGS, which represents registers known to contain pointers, and DATA_REGS, which represents registers known not to contain pointers. We created several additional classes to represent unions of the newly created classes, in order to provide consistency with the original implementation. These modifications had far reaching impact on the compiler, requiring new instruction templates to differentiate between instruction operating on pointers and those operating on non-pointers, and requiring more careful tracking of pseudo-register modes based on the types of the values they represent.

The GNU compiler was designed to be machine independent, and its implementation is designed to support machines with different register classes. Hence, most of the changes required to implement the register partitions were isolated to machine-dependent components of the software system. In particular, most of the necessary changes were made to the file sparc.h. These changes affect the operation of other parts of the register allocator which access the data structures and macros defined in sparc.h.

3.2. The Default Register Partition

As indicated above, the general register set has been partitioned into data and pointer registers. The partition used by default for allocating registers for the functions is called the default partition. We would look at a modification of this partition for improved allocation in a later chapter.

The default partition splits GENERAL_REGS into two sets of 16 registers each for the data and address register set. Each of the four groups (global, output, input and local registers) is divided evenly into 4 registers each. The first four registers of each group form part of the data register set and the last four registers are placed in the address register partition. Note that this partition is consistent with the data types already associated with dedicated registers by the SPARC ISA (instruction set architecture) specification:

Register Name	*Aliased Name*	*Description*
r0	g0	Always has value 0
r15	o7	The call instruction writes the return address to this register.
r17, r18	l1, l2	When a trap occurs, the PC and nPC (next program counter) are copied into these registers of the trap handler's register window.

The partition is only partially compatible with the SPARC ABI (Application Binary Interface) specification, which reserves the following registers [6]:

Register Name	*Aliased Name*	*Description*
r0	g0	Always zero.
r1	g1	A temporary value (caller saved).
r5-r7	g5-g7	Undefined, reserved for future use.
r8	o0	Value returned from callee.
r8-r13	o0-o5	Outgoing parameters 1-6.
r14	o6	Stack pointer.
r15	o7	Temporary value within function body / Return address associated with call instruction (the callee returns to i7, its alias for this register).
r24	i0	Value to be returned to caller.
r24-r29	i0-i5	Incoming parameters 1-5.
r30	i6	Frame pointer
r31	i7	Return address

Note that register o0 is specified to hold the return value from a called function. Once registers are partitioned, o0 is not allowed to hold pointer data. So with the partitioned register set, we adopt the convention that functions returning pointer values return their result in register o4 instead of o0. Similar conventions guide the assignment of parameters to the available registers. o0-o3 (aliased with i0-i3 in the callee) represent the first four non-pointer parameters, and o4-o5 (aliased with i4-i5 in the callee) represent the first two pointer parameters.

Because certain registers are permanently dedicated to particular special functions, the register partitions do not generally provide equal numbers of allocatable pointer and non-pointer registers. For example, there are four allocatable non-pointer registers in the input register set, but there are only two allocatable pointer registers in the input set. Though it might have been desirable to partition the registers differently in order to have equal numbers of allocatable registers in each class, we are motivated to retain the partition as originally designed by another concern. In particular, the entire run-time stack is tagged prior to execution as a repeating sequence of four descriptor words followed by four terminal words. Function activation frames are comprised of as many consecutive repetitions of this pattern as are needed to represent the local variables of the function. Using this technique, function calls and returns are nearly as efficient for the garbage-collected system as they are for traditional code, unlike Schmidt's design [10] which incurred a minimum of 9 instruction of additional overhead on every function call for the garbage-collected system. In case the register windows overflow, the input and local registers for the offending function will be copied into the activation frame for the function. The stack locations reserved for the representation of these variables have already been tagged, four words at a time, according to the special pattern with which the entire stack was tagged prior to run time. Thus, it is necessary for the registers that are saved into the run-time stack to hold values that are consistent with the tagged memory locations to which these register values may need to be stored. Even though the number of allocatable registers of each register class is not the same, there are plenty of available registers of each class to satisfy the register needs of most of the workloads that we have studied. This is corroborated by the measurements reported in section 5. To summarize, the default register partition is tabulated in table 3.2.1.

Table 3.2.1: Default register partition

Data Registers	*Address Registers*
g0 − g3	g4 − g7
o0 − o3	o4 − o7
l0 − l3	l4 − l7
i0 − i3	i4 − i7

3.3. Function Return Values

According to the standard protocol, return values are always placed by the callee into its register i0. However, the register partition requires that i0 only hold non-pointer values. Thus, register i0 can only hold return values if the function's return type is a non-pointer. For functions returning pointer data types, we have chosen to use register i4 to hold the return value. This change is implemented by modifying the related macros in the file sparc.h. Similar changes were implemented in order to pass function parameters in registers that are consistent with the register partitions.

3.4. Operand Constraints

As indicated above, the md file contains a list of instruction patterns for all available instructions in the target architecture. The patterns specify constraints on the arguments to each instruction. For example, instructions that require operands to be in registers specify this as a operand constraint. Constraints may also be specified to require that operands reside within registers of a particular class. When the compiler generates code by matching machine-instruction patterns against rtl expressions, the compiler inserts whatever additional instructions are necessary to move operands into locations that satisfy the selected machine instruction's operand constraints.

Within the md file, constraints are specified as comma-separated strings of characters, each string representing a different allowed operand type. For example, the r constraint means that an operand may appear in a general-purpose register; and a g constraint means that an operand may appear in memory, a register, or as an immediate operand of the machine instruction. If the constraints are specified as null, any operand type is permissible. For example, consider the following rtl instruction pattern:

```
(define_insn ""
  [(set (match_operand:SI 0 "reg_or_nonsymb_mem_operand" "=r,f,r,r,f,Q,Q")
     (match_operand:SI 1 "move_operand" "rI,!f,K,Q,!Q,rJ,!f"))]
      "register_operand (operands[0], SImode)
       || register_operand (operands[1], SImode)
      || operands[1] == const0_rtx"
```

This instruction has two operands; call them *op0* and *op1*. Both have constraint strings. The constraint string for *op0* is "=r,f,r,r,f,Q,Q" and the constraint string for *op1* is "rI,!f,K,Q,!Q,rJ,!f". If *op0* satisfies the constraint =r, then *op1* must satisfy the constraint rI. And if *op0* satisfies the constraint f, then op1 must satisfy the constraint !f. Of the seven allowed combinations of operand constraints, the code generator selects the version that results in the least expensive translation.

It is important in our revised system to constrain the register allocator so that it distinguishes at all times between operands representing pointers and those representing non-pointers. For example, if an rtl instructions adds a pointer value to an integer, the result is a new pointer value. In the original md file, there was a single add instruction, which expected two register sources and produced a register result. In the revised machine description, there are three different patterns representing the add instruction. One pattern accepts two data register source operands and produces a data register result. The other two patterns accept a combination of one data register and one address register as source operands and produce an address register result.

We considered two alternative techniques for revising all of the operand constraints for machine instructions:

1. Modifying the md file directly. Since there are several hundreds of instructions described in this file, and the patterns characterizing each instruction are rather complicated, we viewed this as error prone and tedious.
2. Modifying the genoutput program which has the responsibility of reading the md file and converting its contents into tables for use by the code generator. These changes are then reflected in the insn-output.c file, which is produced by genoutput.

We chose the latter option, mainly because it appeared to require less work than the first[3]. However, we feel that a better long-term solution would be to modify the md file. The problem with the current implementation is that our technique required us to insert machine-specific code into program components that were originally designed to be machine independent. Architectures such as the Motorola 68000 already distinguish between address and data registers. The md files for whatever targets to which we want to port the garbage collection system should be modified to represent the same sorts of restrictions that are present in the 68000 specification.

We have modified genoutput to give special handling to all occurences of the following constraints:

r: In the original code generation model, an r constraint suggests that any general register operand is acceptable. In the revised system, we replicate the instruction's pattern, with different constraints for each copy. The r constraint is replaced in the instruction copies with a or d constraints, meaning address register or data register respectively. The choice is based on the mode of the operand (i.e PSImode implies an address register).

g: In the original code generation model, a g constraint means that any memory, immediate operand, or register is acceptable. We replace this constraint in the replicated patterns with the strings mia and mid, which mean memory (or register) representing address and memory (or register) representing

[3] We were not entirely sure whether we would be satisfied with the results of register partitioning. The purpose of our research was to investigate this question. If the results of our research were to conclude that the performance of register-partitioned code is unacceptably poor, we wanted to discover this problem with minimal investment of time and energy.

data respectively. (mia stands for memory, immediate, address register. mid stands for memory, immediate, data register.)

null: In the original code generation model, a constraint of null means that any operand is permissible. However, in the revised system, these constraints must be replaced so as to narrow the possibilities to either data or address locations only for each instance of the instruction pattern. It was not possible to specify a general constraint that represents the desired restrictions. Thus, rather than modify the instruction constraints, we searched for all uses of the null operand constraint in the code generator's instruction matching software and modified each of these instances appropriately.

Mapping of operand constraints by the genoutput program is implemented by the correct_constraints subroutine found in the source file genoutput.c. Its prototype is shown below.

```
static void correct_constraints(mode, opno)
  mode: Machine mode of the operand whose constraints are under
          modification.
  opno: The operand number - this is required in order to
          access the constraints.
```

3.5. Function Calls

Registers play a major role in the processing of function calls in the SPARC architecture. They are used for passing parameters and returning values. We have already seen the convention employed in the use of registers for function return values and the changes needed in them for our implementation. Let us now consider how the parameters are handled.

The caller uses the output registers for passing the parameters and the callee function uses its input registers (which are simply aliases for the caller's output registers) for reading the same. In the standard code generation model, the first six words of parameter data are passed in registers. If there are more parameters than fit in the six available registers, the additional parameters are passed on the stack.

In the revised implementation, it is necessary to differentiate between registers holding pointers and registers holding non-pointers. We have adopted the convention that the first four output registers (o0 − o3) are used for passing the first four terminal parameters and the next two output registers (o4 and o5) are used for passing the first two descriptor parameters. In case the number of words of terminal parameters exceeds four, the additional terminal parameters are passed in stack locations reserved for this purpose. In case the number of words of descriptor parameters exceeds two, the additional descriptor parameters are passed in stack locations reserved for this purpose. Note that an argument list consisting entirely of six non-pointer words passes the last two parameters in stack locations, leaving registers o4 and o5 unused. Further, note that the stack locations reserved for the overflow of terminal parameters are different than the stack locations reserved for the overflow of descriptor parameters. This is because locations within stack activation frames are tagged as to their type prior to execution of the program.

The above design tends to penalize functions with more than two words of pointer parameters. This design was chosen so as to simplify the design of the stack activation frame. Diverging from an even split in registers would have complicated the implementation of activation frames. See reference 4 for a characterization of the costs of this implementation choice for the workloads that have been studied so far.

This change to the original design is manifest in two places:

assign_parms()
: assign_parms(), found in function.c, generates rtl code to access actual parameters from within a called function, and

expand_calls()
: expand_calls(), found in calls.c, generates rtl code to invoke a function. Included in the generated rtl is code to pass parameters to the called function.

Several macros were changed in sparc.h to reflect these changes in the passing of parameters. These macros are exercised within assign_parms() and expand_calls().

Even though the input and output registers were designed specifically for parameter passing, these are general purpose registers which the register allocator is free to allocate to particular needs within the function body. Any registers not assigned to parameters are candidates for allocation. Additionally, registers representing parameters can be allocated by spilling their contents to stack locations. Redundant stack locations are reserved within the activation frame of a caller to hold register parameters that need to be spilled. These six register spill locations are always found at the same fixed offset relative to the caller function's activation frame.

Special handling is also required for variable argument lists. In the traditional code generation model, the first six arguments are placed in registers and subsequent arguments are placed in known stack locations. Support for unraveling the variable arguments simply requires that the run-time system remember the location and type of the most recently processed argument. In the revised system, the run-time support must keep track of slightly more information. In particular, it must remember the type and location of the last descriptor argument, and also must remember the type and location of the last terminal argument. This is described more completely in reference 4.

3.6. Pseudo-Register Mode

A hard register can be allocated only to pseudo-registers with compatible modes. For example, hard register r1 cannot be assigned to a pseudo-register whose mode is PSImode, because this mode indicates that the pseudo-register represents a descriptor. r1 is a register dedicated to representing terminal data. The restrictions are specified by the array hard_regno_modes_ok[], found in the source file aux-output.c. This array is indexed by a hard register number. The value stored in hard_regno_modes_ok[r] is the integer representing the register class that represents the mode (or modes) that may be represented by register r.

4. Non-Standard Register Partitions

4.1. Motivation

In section 3, we motivated and described the use of a default partition which was designed to distinguish between descriptor and terminal data. In the default partition, the general register set is divided into two sets of 16 registers each. All allocations in a certain class are made from registers suitable for that class. It is not possible to allocate a register of one class to satisfy a need for a register of the other class. For most programs, this default partition provides good performance (see section 5). However, in some cases, the default partition results in poor utilization of the architecture's available registers. For example, suppose a particular program has need of 24 terminal registers, but only two descriptor registers. Under these circumstances, it is desirable to make use of a nonstandard register partition.

4.2. Scope of the modification

At the boundary between function calls, it is necessary to honor the default partition, since the analysis and generated code for each function is performed in isolation of all other functions in the system. However, within a function, it is desirable and possible to deviate from the default partition if this enables more efficient code to be generated. If, while generating code based on the default partition, we determine that either the set of terminal registers or the set of descriptor registers has been exhaused, while noting that the other set of registers has several unused registers, we adjust the register partition appropriately and regenerate the translation of this function using the revised register partition. In our current implementation of register repartitioning, we have chosen to repartition only the global and local registers. If garbage collection were triggered while a function is executing with a non-standard partition, the garbage collector must be able to efficiently determine which non-standard registers represent pointers. The registers which hold pointers are identified to the garbage collector by a 32-bit mask stored in memory. This mask is initialized to represent the default partition. However, upon entry into a function that uses an alternative partition, this mask is updated. Similarly, whenever a function using an alternative partition calls another function, it is the calling function's responsibility to restore its register usage to the default partition (by copying certain registers to reserved stack locations and/or by overwriting descriptor registers temporarily assigned to represent terminal data with zeros).

4.3. Implementation Summary

Lifetime analysis and register allocation is initially attempted using the default partition. If the results of generating code based on the default partition suggest that better code would be possible using an alternative partition, the register allocator is informed of the change in register partitioning, and the rtl is reanalyzed and retranslated. All of the following compiler passes are repeated whenever a function is retranslated based on a non-standard register partition: Register Class Preferencing, Register Allocation, Flow Analysis, Instruction Combination, Instruction Scheduling, and Reload.

The array reg_class_contents[] represents the assignment of registers to different classes. The registers are reassigned by adjusting the values of reg_class_contents[DATAREGS] and reg_class_contents[ADDRREGS]. This is done in the function change_reg_sets() which is called at the end of the reload phase. A call to the function init_reg_sets() is also made in order to recompute various arrays like the register class size, and membership in superclasses and subclasses subclasses.

The SPARC front-end has been modified so that the modes of rtl variables indicate whether the variables represent pointers or non-pointers. We use PSImode to represent pointers, SImode to represent integers, DImode to represent double integers, and so on. Our use of PSImode is unique to our implementation of the SPARC compiler. Our use of other modes is the same as in the traditional SPARC compiler implementation. Hard registers which can contain data of a certain class must support the corresponding modes. This is indicated by an array hard_regno_mode_ok[], which is defined in the file aux-output.c. Whenever we modify the register partition, we also modify this array to reflect the change. This is done through the function set_hard_reg_mode_ok().

```
void change_reg_sets(nregs)
  int nregs;      /* Number of registers to be switched between
                   * classes in the partition - Positive indicates
                   * data partition increase and negative indicates
                   * address partition increase
                   */
```

In the code for the default partition, GC_data_map and GC_addr_map were constants providing bitmaps for data and address register class contents. These are utilized in various macros in the file sparc.h. In order to support a variable partition, we have changes these to variables of type HARD_REG_SET which is used in GCC to manipulate register sets. The function init_reg_sets() sets these variables to appropriate values.

4.4. Copy of rtl Code

Since several passes of the compiler (e.g. reload) modify the rtl code, it is necessary to make a private copy of the rtl prior to code generation in order to allow the code to be reanalyzed using a different register partition when desired[4]. Whenever the compiler decides to generate code for a particular function based on a nonstandard register partition, it discards the results of the original analysis and code generation, and repeats all of the relevant analysis and code generation steps using this private copy of the original rtl code. The function make_copy(), defined in emit-rtl.c, makes the rtl copy.

[4] Certainly, this is not the ideal implementation. However, copying the rtl and regenerating code when necessary was the easiest way we were able to devise for supporting repartitioned register sets.

```
// make_copy returns a copy of a chain of rtx instructions, given
//  that first_insn is the head of the original chain to be copied.
rtx make_copy(rtx first_insn)
```

The body of each instruction is copied using the function gc_copy_rtx(). The function gc_copy_rtx() makes a copy of every rtl expression except PC and CC0, which can be shared. Everything else needs a separate copy, especially the REG rtx's. An important concern in the copying of the REG rtx is the generation of the regno_reg_rtx_copy[] array. This is similar to the regno_reg_rtx[] array in the original code. regno_reg_rtx[] is an array of rtl expressions providing the register expressions corresponding to each pseudo-register. There is a unique rtx expression for each pseudo-register. This is essential so that the change in the REG rtx can be made in only one place. Data like the pseudo-register mode can be obtained based on the contents of this array. regno_reg_rtx_copy[] is the corresponding array for the copy of the rtl chain.

```
// gc_copy_rtx returns a copy of a single rtl expression (rtx), recursively
//  copying any rtx operands to the original rtx.  PC and CC0 expressions
//  are not copied; instead pointers to their shared representation are
//  returned.  orig is the rtx to be copied.
rtx gc_copy_rtx(rtx orig)
  register rtx orig;
```

The globals first_insn and last_insn corresponding to the first and last instructions of the rtl chain and regno_reg_rtx[], are restored by the function update_globals() which is called when the copy of the rtl is used to replace the original.

All of the above functions are defined in the file emit-rtl.c.

4.5. The .partition Assembler Directive

As indicated above, any changes to the default register partition are valid only within the current function. At run time, the garbage collector must know that particular functions are utilizing non-standard register partitions. The current register partition is maintained in a 32-bit mask which uses one bit to represent each general purpose register, with a 1 meaning that the corresponding register holds a pointer and a 0 meaning that the register holds a non-pointer. Upon entry to and exit from a function that makes use of a non-standard partition, a .partition directive is inserted into the generated code. This directive takes a single integer argument which represents the new value of the partition bit mask.

4.6. Register Saving

When a function call is encountered, the register contents must be consistent with the default partition. Those registers which do not conform to the default partition need to be saved on to stack locations prior to the call. It is also necessary to ensure that descriptor registers which were used as terminal registers in the non-standard partition are made equal to NULL. This prevents the terminal data from being treated as descriptors in case garbage collection is initiated during execution of the called routine. The additional overhead caused by this can be justified based on the improved utilization of registers made possible by a non-standard partition.

4.7. Overhead

The default partition may result in inefficient utilization of the registers in certain cases. Non-standard register partitioning is designed to make more effective use of available registers whenever a particular function requires many more terminal registers than descriptor registers, or vice-versa. Most of the overhead of repartitioning is paid by the compiler, which first has to recognize the need for repartitioning, and then has to retranslate the function based on the revised partition. The run-time overhead associated with repartitioning includes the extra instructions required to inform the garbage collector of the the change in register partitioning. These instructions are found in the prologue and epilogue, and surrounding every function invocation. We expect that the overhead will not be large, however, for the following reasons:

1. Very few functions require register partitioning, as discussed in section 5.
2. Compile-time analysis can approximate the costs and benefits associated with each proposed repartitioning of the registers. If the costs outweigh the benefits, register partitioning is not done.

5. Empirical Results

In this project, we have partitioned the register set into terminal and descriptor registers in order to facilitate the garbage collection process. We have analyzed a couple of experimental workloads (benchmarks) to see the effect of this change.

The experimental workloads are described briefly below:

cham:

Chameleon is an N-level channel router for multi-level printed circuit design.

cfrac:

Cfrac is a program to factor large integers using the continued fraction method.

Since, we needed to collect data statically, it was not necessary to execute the benchmarks. All the necessary data was collected during compilation.

The purpose of this study was to obtain the following:

1. Total number of pseudo-registers used by each function.
2. Number of hard registers utilized.
3. Number of pseudo-registers which the register allocator failed to place in hard registers.

5.1. Total Number of Pseudo-registers:

cham:

The table below summarises the data on the number of pseudo-registers used in the benchmark:

Table 5.1.1:Total Pseudo-Registers - cham

	# of Functions with pseudo-registers	
	Under 30	Total
Terminal	89	108
Descriptor	88	108

As the above table indicates, over 80% of the functions have less than 30 terminal and less than 30 descriptor pseudo-registers. This means that there is a greater chance that all pseudo-registers will get assigned hard registers. This is also obvious from the higher concentration of the points closer to the origin (Figure 5.1.1). From the graph it can be seen that the majority of the points fall close to the diagonal of the terminal/descriptor plane. Since the terminal and descriptor axes are of the same scale, this implies that functions tend to have comparable number of terminal and descriptor values to be allocated in hard registers.

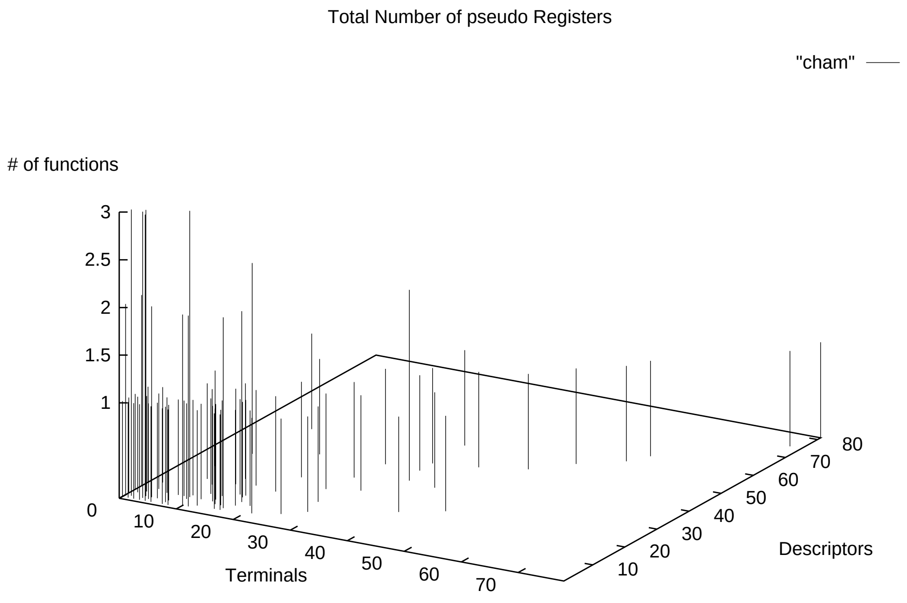

Figure 5.1.1: Total Pseudo-Registers - cham

cfrac:

The table below summarises the data on the number of pseudo-registers used in the benchmark:

Table 5.1.2:Total Pseudo-Registers - cfrac

	# of Functions with pseudo-registers	
	Under	Total
Terminal (Under 30)	40	55
Descriptor (Under 10)	40	55

Again the table, indicates that over 70% of functions have under 30 terminal and under 10 descriptor pseudo-registers. We can conclude that this benchmark has much more number of terminals to be allocated than the number of descriptors.

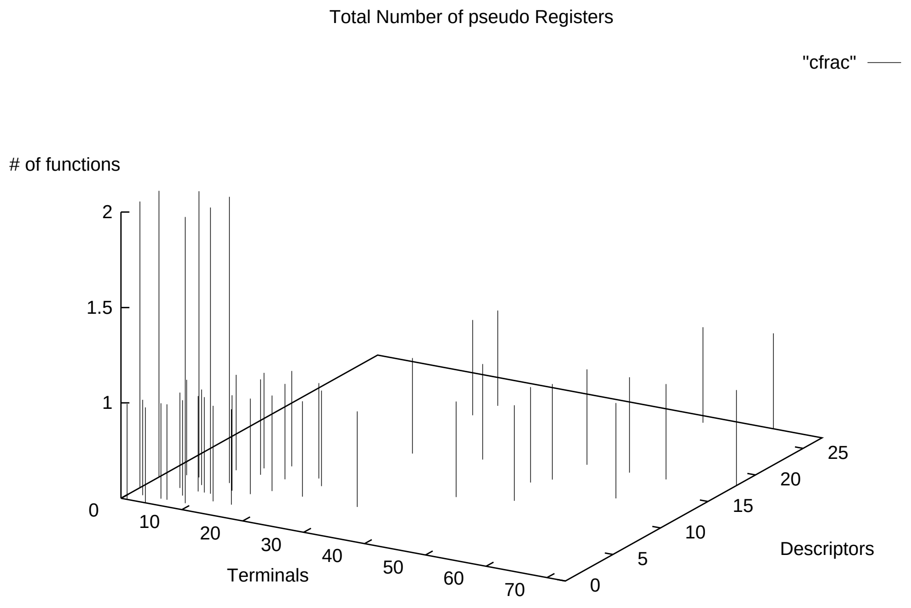

Figure 5.1.2: Total Pseudo-Registers - cfrac

5.2. Number of Hard Registers Used

cham:

The hard register utilization can be seen from the graph provided. The following table provides useful summary about the same:

Table 5.2.1:Hard Registers Used - cham		
	# of Functions with Hard Registers	
	Under 10	Total
Terminal	94	108
Descriptor	96	108

It is interesting to note that over 85% of all the functions use 10 or less terminal and 10 or less descriptor hard registers of the available 16. We can see from the graph (Figure 5.2.1) that the points are randomly distributed all over the plane, indicating that the ratio of terminal to descriptor hard register usage varies widely between functions. We note that this varies from the pseudo-register ratios which seemed to stay close to 1 for most functions. This is an indication that there is no correlation between the usage of hard registers and pseudo-registers.

Also, we can see that relatively few points are near the outer boundaries of the X-Y plane, which means that few functions tend to make full utilization of all available hard registers.

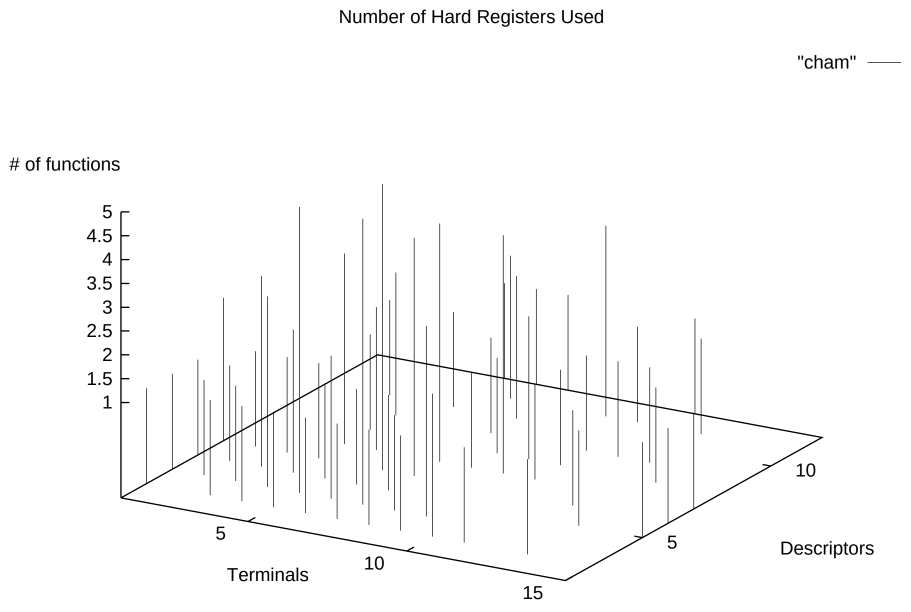

Figure 5.2.1: Hard Registers Used - cham

cfrac:

The hard register utilization can be seen from the following graph . The table below also provides a summary:

Table 5.2.2: Hard Registers Used - cfrac

	# of Functions using Hard Registers	
	Under 10	Total
Terminal	53	55
Descriptor	49	55

Again, it can be noted that about 90% of the functions use less than 10 terminal and descriptor registers. The observations made for the cham benchmark seem to hold here too. There appears to be more demand for descriptor registers than for terminals. This is in contrast with the pseudo-register distribution which was otherwise (see reference 4). However, the available number of registers seems to be sufficient to meet the application's requirements. It should be noted that some descriptor hard registers are reserved and cannot be used for general allocation (g[5], g[6], and g[7] are reserved for the environment, the stack pointer, and operating system). This increases the demand for descriptor registers.

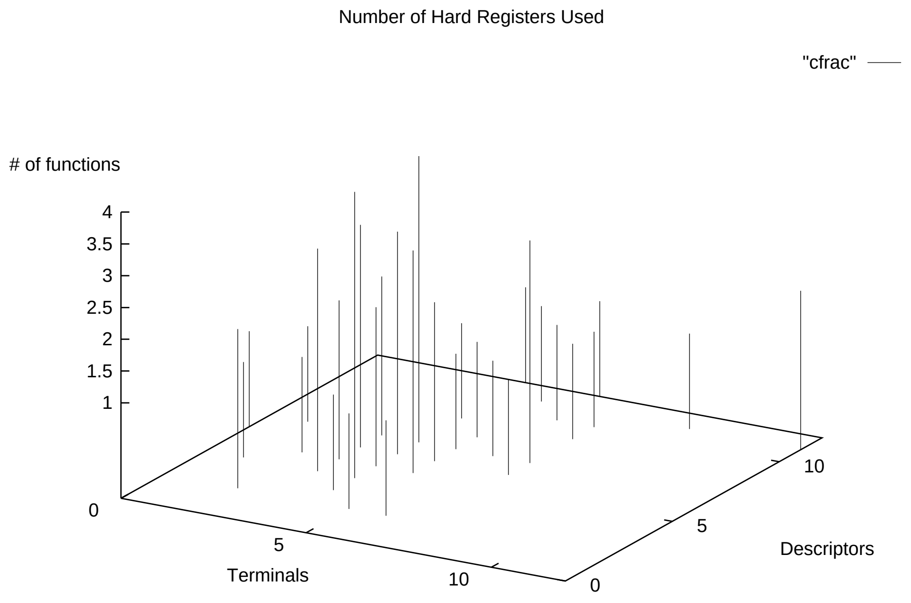

Figure 5.2.2: Hard Registers Used - cfrac

5.3. Number of Unallocated Pseudo-Registers:

cfrac:

The table below summarises the data on the number of pseudo-registers in the benchmark for which we have failed to allocate hard registers:

Table 5.3.1: Remaining Pseudo-Registers - cfrac		
	# of Functions	
		Total
Terminal	0	55
Descriptor	6	55

This indicates that most of the terminal and descriptor pseudo-registers are allocated hard registers. In fact from the following graph we can see that about 90% of all functions have no pseudo-register (terminal or descriptor) that has to be allocated a stack location in lieu of a hard register. This figure does indicate that dividing the register set into 2 classes has not adversely affected the performance of the register allocator. Where pseudo-registers are remaining, the number is restricted to four or less.

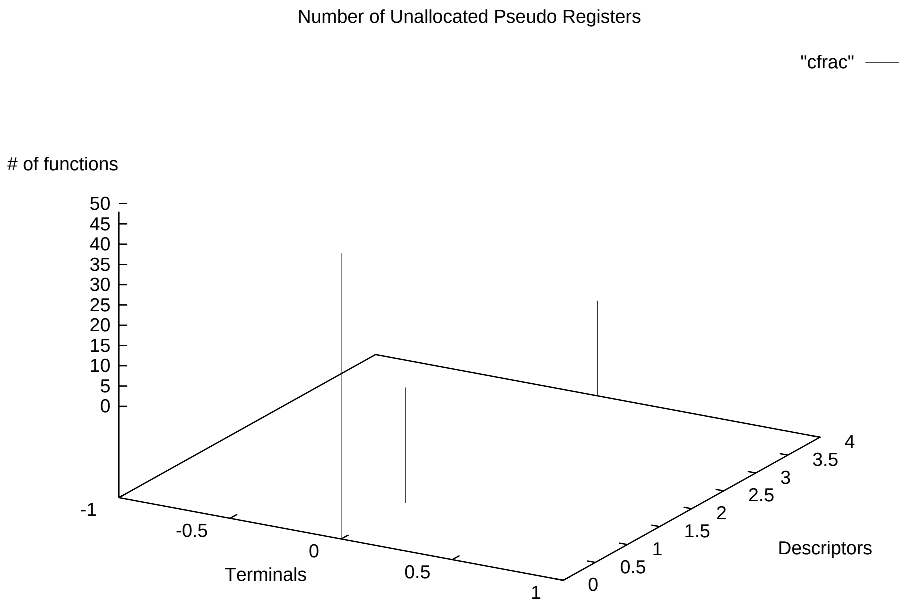

Figure 5.3.1: Remaining Pseudo-Registers - cfrac

cham:

The data of unallocated pseudo-registers in the cham benchmark is summarized in the table below:

Table 5.3.2: Remaining Pseudo-Registers - cham

	# of Functions	
		Total
Terminal	5	108
Descriptor	7	108

These figures are similar to those obtained for the cfrac benchmark. Over 95% of all the functions have their terminal pseudo-registers placed in hard registers and over 95% have their descriptor pseudo-registers placed in registers. Over 90% of the functions have their hard register needs fully satisfied.

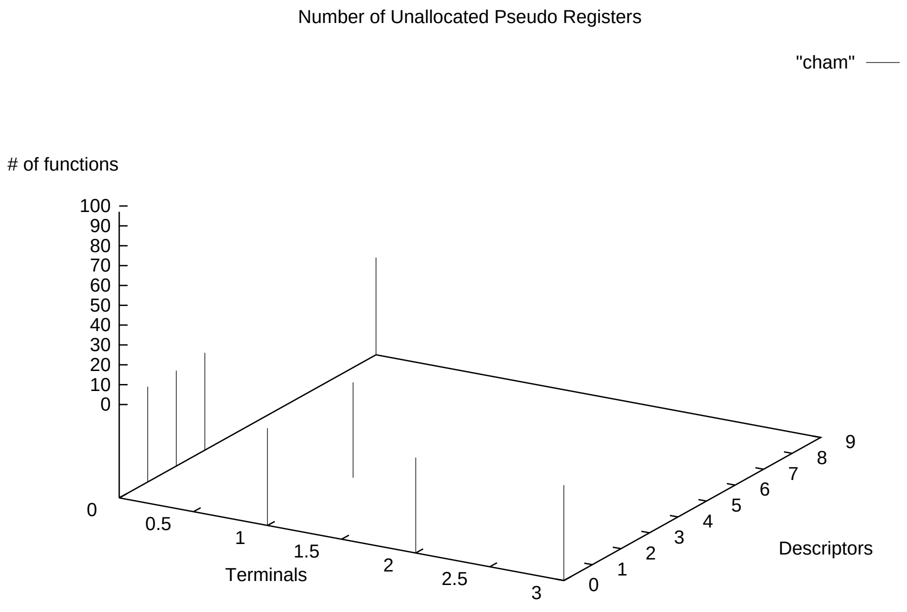

Figure 5.3.2: Remaining Pseudo-Registers - cham

5.4. Conclusions:

Based on the study of the two benchmarks, we can reach the following conclusions:

- Most functions have relatively few pseudo-registers to be allocated to hard registers. In most cases, the number remains under 30. Most of these needs are satisfied by the set of 16 terminal and 16 descriptor registers which are provided by the default register partition. The number of hard registers required by a particular program is not necessarily proportional to the number of pseudo-registers used by the program.
- In most cases, fewer than ten hard registers of each category are needed. Relatively few functions use their full quota of registers. In case one set of registers is fully utilized, some of the registers from the other set are usually available. Very few functions have unallocated pseudo-registers.

This has led to the implementation of a non-standard register partition. The default partition is changed selectively when some of the pseudo-registers cannot be assigned to hard registers based on the default partition. By transferring one or more hard registers between partitions, the register needs can be easily satisfied.

6. Acknowledgements

I would like to express my gratitude to Dr. Kelvin Nilsen, my advisor, for his support and understanding throughout this project. His constant guidance and advice was an invaluable factor in the completion of this project. He has invested countless hours in shaping my work. Working with him has been rewarding experience which I will cherish forever.

I would also like to thank Dr. Akhilesh Tyagi, Dr. Charles Wright, Dr. Gary Leavens and Dr. Gurpur Prabhu, my Program of Study Committee members, for their valuable time and co-operation.

I would never have reached this stage but for the constant encouragement provided by my parents and sister.

I would like to thank the Department of Computer Science and Dr. Kelvin Nilsen for the financial support provided to me during the course of my study. Finally, I am grateful to all my friends at Iowa State University for making my stay here fun and memorable.

7. References

1. H. Boehm and M. Weiser, Garbage Collection in an Uncooperative Environment, *Software—Practice & Experience 18*, 9 (Sep 1988), 807-820.
2. H. Boehm and D. Chase, A Proposal for Garbage-Collector-Safe C Compilation, *Journal of C Language Translation 4*, 2 (Dec 1992), 126-141.
3. A. Diwan, E. Moss and R. Hudson, Compiler Support for Garbage Collection in a Statically Typed Language, *ACM SIGPLAN Notices Conference on Programming Language Design and Implementation*, June 1992, 273-282.
4. R. Ganesan and K. Nilsen, The Design and Implementation of SPARC Activation Frames to Facilitate Accurate Garbage Collection of C++.
5. S. K. Guggilla, Generational Garbage Collection for C++ Targeted to Sparc Architectures, Master's Degree.
6. S. International, *The SPARC Architecture Manual, Version 8*, Prentice Hall, Inc., Menlo Park, CA, 1992.
7. K. Nilsen and W. J. Schmidt, Cost-Effective Object-Space Management for Hardware-Assisted Real-Time Garbage Collection, *ACM Letters on Prog. Lang. and Systems 1*, 4 (Dec. 1992), 338-354.
8. K. Nilsen, Reliable Real-Time Garbage Collection of C++, *Computing Systems 7*, 4 (Fall 1994), .
9. K. D. Nilsen and W. J. Schmidt, A High-Performance Hardware-Assisted Real-Time Garbage Collection System, *Journal of Programming Languages*, To appear.
10. W. J. Schmidt, Issues in the Design and Implementation of a Real-Time Garbage Collection Architecture, Ph.D. Dissertation, Iowa State Univ. Tech. Rep. 92-25, 1992.
11. R. M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, Inc., Cambridge, MA, 1989.
12. B. Zorn, The Measured Cost of Conservative Garbage Collection, *Software—Practice & Experience 23*, 7 (Jul 1993), 773-756.