Global Program Optimization: Register Allocation of Static Scalar Objects

Andrea G. M. Cilio Henk Corporaal

Department of Electrical Engineering,
Delft University of Technology,
P.O. Box 5031, 2628 CD Delft, The Netherlands
A.Cilio@et.tudelft.nl heco@cs.tudelft.nl

Keywords: Program optimization, Compiler technology, SUIF, Linkage, Register Allocation

Abstract

Performing certain optimizations on whole programs offers huge potential for code improvement. To exploit this possibility, we implemented a framework that allows sophisticated analysis and transformation of machine code, during or after linkage. This paper presents one of the many applications of our framework: the allocation of global scalar variables into registers. We present the algorithms that select the variables and remove the load and store instructions associated with them. The simulations of the unscheduled code of our benchmarks show that up to 10% of the execution cycles is saved by applying this technique.

1 Introduction

A traditional compilation path is split into three phases: compilation proper (front-end plus back-end), assembly and linkage. To support the many, complex code transformations performed during compilation, a rich internal intermediate representation, which usually is machine-independent and preserves most of the source-level information, is used. In the second phase, the assembler parses the assembly file and translates the code into a binary format of the target machine. Finally, the linkage phase merges one or more object files and libraries into an executable binary file. During linkage, symbolic references to data objects or instructions are *resolved*, i.e., replaced with the actual physical address of the referenced symbol.

This well-established code generation path (see Fig.1) offers three advantages for general purpose target processors. First, it is *efficient*: the binary format is a compact representation; linking modules and libraries does not require expensive code transformations. Second, it becomes possible to distribute library code without the source files. Third, since assembly syntax and binary format are standardized with respect to a target architecture and an operating system, it promotes *interchangeability* of the tools that perform assembly and linkage, provided that they comply to these standard formats.

However, this approach to compilation presents also severe drawbacks. During the translation of the compiler's intermediate format into assembly code, most of the source-level information is lost. This information is essential for state-of-the-art, machine-dependent code optimization. In particular, optimizations applied to *whole programs*, after linkage, have a great potential for code improvement. The binary formats commonly used at link time are awkward for applying sophisticated code transformations. Also, this approach is hardly acceptable in the context of automatic architecture exploration, common for hardware-software co-design. As we showed in [1], the traditional approach requires to compile a version of the library code for each point of the target space being explored.

These drawbacks lead us to revisit the traditional compilation trajectory. In this paper we present one of the many potential optimizations that are enable by this new trajectory: the assignment of global scalar variables to registers.

Other potential optimizations, like inter-procedural register allocation and automatic floating- to fixed-point conversion of library code at link-time, are being researched as well.

The rest of this paper is organized as follows. Section 2 presents a general description of our compilation system and states the problem. Section 3 details the algorithms that select the global scalar variables that are candidates for register allocation. In Sec.4 we estimate the potential performance improvement

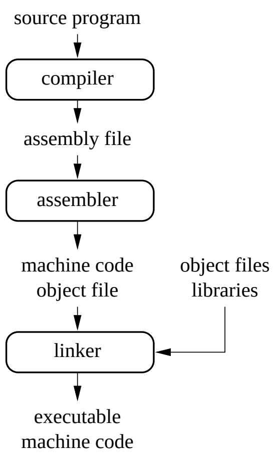

Figure 1: The traditional code generation trajectory.

given by our optimization on two benchmarks. Section 5 reviews related work. Finally, Sec.6 concludes this paper.

2 The Compilation Trajectory

Figure 2 shows our code generation system. It generates code for a templated architecture for Application Specific Instruction-Set Processors called *Move*. This architecture offers explicitly programmed instruction-level parallelism, in a fashion similar to that of VLIW architectures [2]. The code generation is coarsely split in two phases: (1) compilation to a generic-machine instruction set and (2) target-specific instruction scheduler, which integrates also register allocation [3].

Notice that our approach to register allocation of global variables, as described in Sec.3, can be applied to any general-purpose target processor, although the traditional binary format complicates code transformation.

2.1 The SUIF Representation

The intermediate representation used in the first phase of code generation is *SUIF*, the Stanford University Intermediate Format [4]. The SUIF infrastructure includes a suite of optimizations. All code optimization and analysis passes are implemented as SUIF-to-SUIF transformations.

Instead of generating the traditional assembly textual output, the compiler generates and maintains a structured representation of the machine code in *MachSUIF* [5], a format derived from SUIF. MachSUIF maintains all source-level information, as well as any other piece of information gathered during SUIF analysis passes. This information is essential for our code optimization: disambiguation of global scalar variables and allocation to registers thereof.

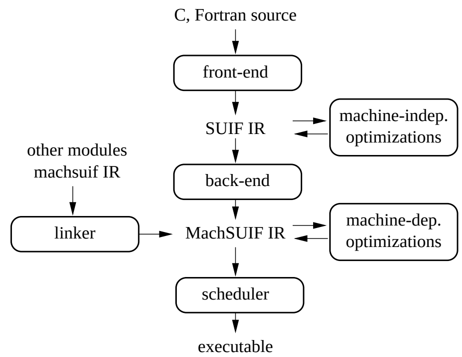

Figure 2: The new code generation trajectory.

2.2 Motivation for Allocation of Global Variables into Registers

Which is more convenient to assign to a register: a rarely accessed local variable of a leaf procedure or a global variable v that is frequently accessed in many points of the program? It is this question that motivates this work. Under some conditions, e.g. v is not aliased in *any of the program modules*, allocating v to a register and spilling a local value may yield faster code.

Since a compiler handles one translation unit (or module) at a time, even if a global variable is not aliased in a module, the compiler must conservatively assume that the variable is aliased in other modules. A global variable can be guaranteed not to be accessed through pointers only after all program modules have been linked together. This means that only after linkage we can assign a global variable to a register and remove all load and store instructions that are associated with it.

2.3 Interaction with Register Allocation

In the Move framework, register allocation is performed during scheduling. The integrated register allocator, called *Register on Demand* (RoD) analyzes all the live ranges of the unscheduled code. These live ranges are represented by *virtual registers*; a virtual register number uniquely identifies a live range.

The purpose of our Global Register Allocation optimization is to replace memory references to a global variable with a single virtual register that can be accessed in the whole program. This replacement does not imply that the variable will be certainly allocated in a register; it simply presents more live ranges, and thus more candidates for register allocation, to RoD.

Currently, RoD works on one procedure at a time and does not handle inter-procedural live ranges. The introduction of global live ranges requires some adaptations to register allocation. The decision whether to spill or assign to a register a global live range is taken "on demand", locally to a procedure. Nevertheless, this decision can greatly influence the rest of the program. This means that the heuristics used to direct a decision must take into account its global cost.

The cost terms themselves require adaptations. For example, the cost of spilling a register that holds a global variable v is lower than normal spilling [3]:

$$\begin{aligned} C_{spill}(v) &= \sum_{\text{uses of } v} C(ld) f(use) + \\ &+ \sum_{\text{defs of } v} C(st) f(def) \end{aligned}$$

where $C(i)$ is a cost estimate for performing instruction i and $f(\cdot)$ is the frequency of an operation. Spilling is cheaper because global variables should be spilled to a static address shared among all procedures, while local variables are spilled to the stack, and this requires also an explicit address computation (an add operation).

3 Implementing Register Allocation for Global Variables

The SUIF format includes a flag for each variable which answers the question: "*could* the address of this variable be taken?". This indicates whether the variable might be aliased or not. We can build on this information to decide which scalar variables are safe candidates to be allocated to registers. The final selection of candidates is performed after linkage, and the code is modified accordingly.

3.1 Global Alias Analysis

The SUIF front-end does not perform alias analysis of global variables: it simply assumes that all global variables have their address taken, therefore they cannot be allocated in registers. Before being linked together, the global variables in each module must be analyzed to check for possible uses through aliases. This analysis answer a more useful question: "is the address of this variable taken *in this module*?". During linkage, the flags of a variable used in several modules are combined; the variable is aliased if its address is taken in at least one of the modules. In this way we obtain accurate global alias information on all scalar variables and we are enabled to allocate unaliased global variables to registers.

3.2 Variable Selection and Code Transformation

The algorithm selects the global variables that are legal candidates for register allocation and removes load and store instructions associated with them. This transformation is applied after linkage and comprises two steps.

```
Procedure Select(symtab)
beginproc
l = ∅
foreach variable v in symtab do
   if not address_taken(v) and is_scalar(v)
      add a new entry e in l
      place v in e, assign a new register r to e
   endif
   if v is initialized
      let i be the initialization value of v
   else
      let i = 0
      initialize v with 0
   endif
   if type(v) is integral
      place instruction 'mov r,i' in e
   elseif type(v) is float
      place instruction 'lds r,addr(v)' in e
   elseif type(v) is double
      place instruction 'ldd r,addr(v)' in e
   endif
end
return l
endproc
```

Figure 3: Algorithm for selecting candidate variables for register allocation out of a symbol table.

First, the global and the *file* symbol tables are scanned and scalar variables are selected. The global symbol table contains symbols with external linkage, while the file symbol table contains static symbols with internal linkage. Figure 3 presents the procedure *Select*, that detects the candidates for register allocation and records them in a list. A global list of candidate variables, L, is maintained. For each variable, a virtual register associated with the live range and the instruction needed for its initialization are recorded in L.

Initialization needs special treatment. A global or static variable gets a defined initial value from the DATA or BSS sections of the program. Since selected variables live in (virtual) registers, we must ensure that these registers are initialized prior to any use. To accomplish this, we create a reserved initialization function `__init_scalar`, to be invoked by the startup-code before calling `main`. This function acts as a container for all instructions that initialize the registers.

The second part of the algorithm scans the procedure symbol tables and adds candidate static variables to L, then it replaces load and store instructions of variables recorded in L with uses and definitions of the associated registers, respectively. During this process, the section offsets of procedures and labels are updated to account for the instructions removed. The instructions for initialization, maintained in L,

Procedure Transform(*file*)
beginproc
install_proc($initproc$, "__init_scalar")
foreach procedure p in *file* **do**
  $L = L \cup$ Select(symtab(p))
  Transform(p)
end
foreach instruction i in L **do**
  append($i, initproc$)
end
endproc

Procedure Transform(*proc*)
beginproc
foreach instruction i in *proc* **do**
  if i does not access memory **continue**
  let v be the variable addressed by i
  r = lookup(L, v)
  if r
    replace(i, r)
  endif
end
endproc

Figure 4: Algorithm for selecting local candidates to register allocation and for modifying the code.

Variable	Cycles	Percentage
`in_count`	79318	3.1%
`maxbits`	39666	1.5%
`offset`	37476	1.5%
`free_ent`	35963	1.4%
`n_bits`	23988	0.9%
`out_count`	23977	0.9%
`maxcode`	11994	0.5%
`clear_flg`	11989	0.5%
`maxmaxcode`	11988	0.5%
total	279384	10.9%

Figure 5: Load/store operation executed for the 9 most frequently accessed global variables in `compress`.

Variable	Cycles	Percentage
`bits_left`	57890	1.6%
`get_buffer`	31703	0.9%
`dcinfo`	3754	0.1%
`dcinfo`	3754	0.1%
`row_buffer`	251	0.0%
total	3496919	2.8%

Figure 6: Load/store operation executed for the 5 most frequently accessed global variables in `djpeg` while decoding an input image into *ppm* format. Note `dcinfo`, declared in different source files with the same name.

are added to `__init_scalar`. Figure 4 presents this algorithm in detail.

4 Performance Potentials

We performed our measurements, in a semi-manual fashion, on a Unix benchmark, `compress`, and on `djpeg`, a standard *jpeg* decoder. We manually performed alias analysis of global exported variables, while file-global variables were automatically detected by a portotype implementation of our algorithm. The results show that 22 global scalar variables are used in `compress` and 49 in `djpeg`. Interestingly, all these variables are *not* aliased and could be selected as candidates for register allocation.[1] In addition, we searched for global scalar variables within library code linked to the benchmarks. Actually, very few global *library* variables are scalars. We found 9 scalar library variables in `compress` and 11 in `djpeg`. However, in this case 3 of these variables are aliased, in both benchmarks.

Next, we simulated the execution of the benchmarks. The library variables were not considered, as their influence on the execution cycles is negligible. Then, we scanned the profiled code obtained and searched for occurrences of load/store instructions referring to the previously selected variables. For each occurrence found we summed its execution frequency to the total of cycles that could be saved by allocating that variable in a register.

The results show that in `compress` 9 global variables cover more than 98% of the load/store operations involving selected global variables. Up to 10.9% of the total dynamic cycle count could be avoided by allocating these variables in registers. Figure 5 gives a more detailed picture of the results. The first column contains the names of the global variables; the second column contains the number of cycles spent loading or storing each variable; the third column translates these figures into percentage of the total dynamic cycle count.

In `djpeg`, only 4 global variables are sufficient to cover over 99% of the load/store operations involving global scalar variables when the output format selected is *ppm*. However, for *gif* output format the profile of global references is very different, as can be seen by comparing Figg.6 and 7: the 10 most-referenced variables cover only 86% of the load/store operations.

In both simulations the potential for improvement is scarce; load/store operations on global scalar variables contribute to only 2.8% *(ppm)* and 2.3% *(gif)* of the dynamic execution count.

These benchmarks were compiled with optimization level '-O2'; the results for the version obtained with '–O1' are almost identical. This is easily ex-

[1] As global variables are accessible from any procedure, they need not be passed as reference (i.e. through their pointer) to functions, hence their address is usually not taken.

Variable	Cycles	Percentage
`waiting_code`	62645	0.3%
`bits_left`	57890	0.3%
`cur_bits`	50986	0.2%
`cur_accum`	50985	0.2%
`bytesinpkt`	44859	0.2%
`histogram`	37111	0.2%
`free_code`	31848	0.1%
`get_buffer`	31703	0.1%
`hash_code`	31638	0.1%
`hash_prefix`	25166	0.1%
total	21848134	2.3%

Figure 7: Load/store operation executed for the 10 most frequently accessed global variables in `djpeg` while decoding an input image into *gif* format.

plained by the fact that the compiler does not perform any inter-procedural optimization even with '–O2'.

The average number of registers that are live in a procedure is 7.3 in `compress` and 20.8 to 32 in `djpeg`. This suggests that for modern processors with 32 registers or more, many registers may be left unused most of the time and would be available for global live ranges. For `compress`, 9 registers are sufficient to achieve nearly all the potential improvement.

These results are conservative in some sense. The variable selection and the code transformation were applied to the benchmark code only and do not consider the library code, while the simulation results include the cycles spent in library functions. However, this effect is negligible, as demonstrated by a later analysis of library code performed by our program.

The simulations were performed on unscheduled code, which is comparable to code of a very simple RISC instruction set. The simulation assumes that all unscheduled instructions take one cycle. Only after scheduling, the simulator uses a realistic model of the target machine, which includes the actual latency of the instructions. We expect that instruction scheduling and the realistic model used by the simulator will result in an increase of the performance gain delivered by our technique for two reasons:

- the latency of load operations[2] is typically longer than the average CPI, and
- loads tend to represent a scheduling bottleneck, therefore by removing load instructions we shorten the dependency chain.

5 Related work

Wall [6] developed a global register allocation scheme split in two phases. During local register allocation, the compiler assigns registers for expression temporaries using an algorithm based on graph coloring. Immediately before linkage, register allocation is completed with the allocation of global variables and function arguments passed by register. This removes most of the caller/callee save and restore code. An essential drawback of this approach is that the architectural registers are partitioned in two sets: one is reserved to temporary live ranges managed by the compiler, and one to function arguments and global variables, managed at link time. In contrast, our approach does not impose any presumptive partition on the register set, and has thus chances to perform better.

In his work, Wall remarks the large potential of code optimizations applied to whole programs at link time or thereupon. The need for an effective transformation engine that enable code transformations after linkage is also stressed. In [7] the authors propose *OM*, a system that enables code analysis and transformation at link-time. A significant drawback of this approach is its cost: OM must *undo* the work performed by the assembler and *redo* analyses that were already performed by the compiler. More importantly, a substantial part of source-level information has been irretrievably lost when the binary code was generated. In contrast, in our compilation system nearly no information is lost. The compiler's intermediate representation is augmented to represent the specifics of the machine instruction set. Thanks to this, we can take advantage of information gathered during early compilation stages, like alias analysis, for transformations on machine code.

6 Conclusions and Future Work

We have developed a compilation system that maintains a rich intermediate representation of the program down to the machine level. This representation enables many whole-program optimizations at link-time or immediately thereafter. In this paper we presented one of these optimization. As the whole program is available, no conservative assumption is needed for variables used across modules and register allocation of global scalar variables is possible.

The preliminary results show a large potential for improving the code. Moreover, since these tests were performed on unscheduled, RISC-like code, the values obtained represent an interesting indication for many current processors, based on a RISC core.

In our work we considered only global scalar variable and ignored allocation of registers across function calls. More improvement is possible if interprocedural allocation based on call-graph is considered. Another promising optimization based on call-graph is the allocation of local, stack-living objects into static areas, to save stack address computation.

[2] The dominant type of reference to a global scalar object.

References

[1] Andrea G.M. Cilio and Henk Corporaal. A linker for effective whole-program optimizations. In *Proceedings of HPCN*, Amsterdam, The Netherlands, April 1999.

[2] Henk Corporaal. *Microprocessor Architectures; from VLIW to TTA*. John Wiley, 1997. ISBN 0-471-97157-X.

[3] Johan Janssen and Henk Corporaal. Registers on demand: an integrated region scheduler and register allocator. In *Conference on Compiler Construction*, April 1998.

[4] Stanford Compiler Group. *The SUIF Library*. Stanford University, 1994.

[5] Michael D. Smith. Extending SUIF for Machine-dependent Optimizations. In *Proceedings of the First SUIF Workshop*, January 1996.

[6] David W. Wall. Global register allocation at link time. Technical Report 6, Western Research Laboratory, Digital Equipment Corporation, October 1986.

[7] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. Technical Report 6, Western Research Laboratory, Digital Equipment Corporation, December 1992.