MACHINE-INDEPENDENT REGISTER ALLOCATION

Richard L. Sites

Dept. of Electrical Engineering and Computer Sciences C-014
University of California at San Diego
La Jolla, CA 92093

Introduction.

The context of this paper is a machine-independent Pascal optimizer that transforms an intermediate stack-machine pseudo-code program into a generally smaller and faster pseudo-code program. The emphasis of this current paper is on the approach taken for mapping registers and storage, using an abstract but practical definition of the target machine's storage hierarchy. A companion paper [7] describes the overall optimizer project. After starting on this project, additional input became available from a Fortran-to-Pcode compiler written at Stanford [2]. Our storage mapping design is sufficiently robust that the addition of Fortran EQUIVALENCE and COMMON statements required no changes.

Our particular allocation algorithm is not unique or original, but the abstract description of the storage hierarchy, and its inclusion of practical considerations we feel to be an original contribution. The description framework has so far stood the test of describing the storage hierarchy of many commercially-available machines without needing to be extended or embellished. We hope that this is a good predictor for the future.

The purpose of a machine-independent storage (and hence register) allocator is to accept a set of variable names, their corresponding live ranges and frequency of use, and to (re-)map these names into a usually smaller set of names, such that frequently-used variables are mapped into names that are intended to correspond with high-speed access in the final target machine, and such that two variables with disjoint live ranges are allowed to be mapped into the same output name. The effect of such a mapping is to speed up the execution of the final program, while also reducing the amount of run-time storage needed. The rest of this paper describes the problems encountered and the storage description used in our mapping.

The Input.

Our allocation phase runs before final target-machine code generation, but after initial (compiler) mapping of activation records and after a global common subexpression optimizer that creates new temporaries. Running after initial storage mapping allows us to treat overlapping variables (from Pascal variant records and Fortran EQUIVALENCE), and also allows us to detect nested fields within records or arrays, so that only the largest aggregate is mapped as a single unit. Running before target-machine code generation allows us to be machine independent, but requires a partitioning of the registers -- part for us and part left over for code generation. This contrasts with the approach taken in the original Bliss compiler [8] and in the current PQCC project [6].

The input to the storage allocator consists of a Universal P-code program, where P-code is a machine language for a

hypothetical stack machine. Variables (and compiler/optimizer generated temporaries) are referenced by quintuples:

<data type,memory type,block no,offset,length>

The data type might be Integer, Real, Boolean, Array, etc. The memory type is one of the target machine memory hierarchy levels described below. The block number is a unique activation record number for each procedure in the program. The offset and length describe a particular set of storage locations within the sub-activation record <memory type, block number>. The unit of storage (word, byte, bit, etc.) is not specified -- the only constraint is that all offsets and lengths must be consistent. Typically, the unit would be the smallest amount of storage allocated to a field in a PACKED record.

The input P-code has storage mappings within each procedure's activation record already resolved, usually by simple first-declared-first-allocated algorithms in the compiler. In particular, mapping of fields within records and boundary alignment of fields and variables have been done. Fields in variant records, which may overlap or share storage, have been mapped into overlapping or identical <offset, length> pairs. Similarly, Fortran COMMON and EQUIVALENCE statements have been resolved into offsets and lengths within activation records (COMMON blocks or groups of local variables).

A P-code program consists of many procedures, not just one, and each of these consists of many basic blocks (pieces of straight-line code). The input to the storage allocator includes both a P-code program and an indication of the live range of each input name. The live ranges can be gathered by quite traditional flow-analysis techniques, which are not further discussed in this paper.

The Output.

The output of the storage allocator is a P-code program with the variable references re-arranged within each activation record, i.e. the memory type and offset fields may be changed, but the data type, block number, and length fields will remain unchanged. The output P-code will eventually be translated to machine language for some target machine. In general, the target machine may have a storage hierarchy, for example some fast registers, some directly addressable memory, and some indirectly addressable memory.

Our current allocation algorithm is similar to those described by Day [2] and Freiburghouse [3]. It maps as many variables as possible (weighted by frequency of use, and negatively weighted by length) into offset 0 of the top level of the hierarchy, then more variables into the next offset, and eventually next level, until all variables have been mapped. No variable, array, or record is split across levels. The bias toward low offsets is quite deliberate, since it is a good match to machines with short- and long-address instruction formats, it encourages cache locality, it allows use of block load/store instructions for registers, and it doesn't hurt on other machines.

This algorithm is in distinct contrast to optimal-coloring algorithms [4,5], which exhibit no preference for low-numbered locations, and which do not adopt well to choosing which variables to place in each level of the hierartchy when not all fit in the topmost level. An optimal algorithm might need 10 registers for a set of variables when only 9 are available. If so, the optimal coloring gives no hints about the best variables to spill. Our algorithm will put the most heavily used variables in the 9 registers, even though this might cost us a non-optimal mapping that requires more than 10 locations total. It is not clear which approach is better for real programs.

To the extent that a wide range of target machine hierarchies can be accurately described via a small table of parameters, we can use a single storage mapping algorithm to do good register/storage allocation for all of those machines. This is the main thrust of our current work.

The Storage Hierarchy Description.

Our "small table of parameters" consists of an ordered list of descriptions of each level of the hierarchy, starting with the most desirable level (fastest), and ending

with the least desirable (usually main memory). Each level is described with five parameters:

Memory type. A single letter corresponding to the memory type field of output quintuples.

Maximum size. The maximum number of storage units available for this level (i.e. the maximum value of offset + length in an output quintuple with this memory type.

Minimum allocation. The minimum number of storage units to be allocated at once. All variables allocated will have offsets that are a multiple of this number.

Maximum alignment. The maximum interesting storage alignment boundary for the target machine's hardware. Input variables that are aligned to such a boundary (or a simple fraction of one) will remain so aligned.

Set of data types. A list of all the datatypes allowed to be allocated to this level of the storage hierarchy.

We will give an example of a storage hierarchy description, then justify our choice of parameters, then demonstrate the power and flexibility of this particular choice.

Example. A description of the IBM 370 storage hierarchy, with 1 storage unit = 1 bit, 9 general-purpose registers available for storage of variables, 1 floating-point register available, 3800 bytes of directly-addressable memory available, and 16,000,000 bytes of indirectly-addressable memory available.

Memtype	Maxsize	Minalloc	Maxalign	Datatypes
G	9*32	32	32	A,B,C,I
F	1*64	64	64	R
D	3800*8	8	64	A-Z
M	16000000*8	8	64	A-Z

Memtypes:
G - general purpose registers (GPRs)
F - floating-point registers (FPRs)
D - directly-addressable main memory (within the first 4K bytes of an activation record -- assumes that some base or display register will point to the first memory byte of each activation record)
M - the rest of main memory, i.e. large offsets in an activation record, which must be accessed via indirect pointers in the first 4K bytes.

Datatypes:
A - address (pointer)
B - boolean
C - character
I - integer
R - real
Q - record
M - array
etc.

The above description specifies the four interesting pieces of the IBM 370 storage hierarchy, and supplies some details about each piece. The sizes are shown as products, rather than raw bit counts, only for clarity in this example: in the actual P-code, 9*32 would be 288. The minimum allocation size is the size of each register(32 or 64 bits), or the minimum addressable unit (8 bits) in memory. The maximum interesting alignment in the GPRs is one register, and the same in the FPRs. The maximum interesting alignment in memory on the IBM 370 is doubleword alignment -- multiples of 64 bits. The sets of datatypes allowed at each level are carefully choosen to avoid (1) allocation of a real number to a GPR or an integer to an FPR, and (2) allocation of any array to the registers, since it is not practical to calculate a register number as a variable subscript.

Justification.

The maximum size parameter is needed to delineate exactly how much of the target machine resources are available for storage of variables and temporaries. The remaining resources (i.e. the other 7 GPRs, 3 FPRs, and 296 bytes of directly-addressable memory above) are reserved for linkage conventions, indirect pointers, expression evaluation pseudo-stack, and other needs of the target machine code generator.

The minimum allocation parameter makes it possible to respect target machine addressing boundaries, which may vary from level to level. One of the subtleties addressed by the minimum allocation idea is that two 32-bit real numbers must not be allocated to the same 64-bit FPR, while adjacent allocation into 64 bits of main memory is all right.

The maximum alignment parameter is needed so that the storage allocator can preserve incoming boundary alignment of a variable, without "over-preserving" it -- an incoming

variable with an offset that just happens to be a multiple of 1024 should not be constrained to such a boundary if the target machine only cares about multiples of 64. Our alignment-preserving algorithm is:

New Align = GCD(Maxalign,input offset,length).

Length is included so that a 16-bit item is not forced to be on a 64-bit boundary.

The set of datatypes is probably the most important parameter, in terms of making sure that a variable is not allocated in a place that is entirely inappropriate to its representation or to the set of operations to be performed on it. The ability to use datatypes to prevent arrays from being allocated to non-subscriptable registers is also important, and indicates the robustness of the description.

Flexibility.

The above description style has been used to describe the storage hierarcy of the Lawrence Livermore Labs S-1 computer, a 36-bit number-cruncher that vaguely resembles the PDP-10. The machine has 32 36-bit registers, and byte (9-bit) address resolution in main memory. The registers are a subset of the memory address space, so it is possible to reference them as memory addresses. Instructions can address the first 64 words of offset from an address in a register by using only a few address bits, while larger offsets require extra instruction words. In the hierarchy description,the size parameter is straightforward. The minimum allocation parameter is used to allow byte-size variables to be packed together in adjacent bytes in memory, but to allocate registers 4 bytes at a time. The maximum alignment parameter for this machine is 36 bits (not a power of two). Variables are preserved on 36-bit boundaries, and also 18-, 12-, 9-, 6-, 4-, 3-, 2- and 1- bit boundaries. Because of cache line considerations, it may prove desirable to align to 72- or 144-bit boundaries. This is trivially expressed by changing the Maxalign parameter.

In describing the storage hierarchy of the Cray-1, it is necessary to keep large (64-bit) items from being allocated to the small (24-bit) address registers, while still allowing small integers to use those registers. This is accomplished by having two data types -- I for long integers, and J for short integers. Variables of type J can be allocated anywhere, but the short registers are higher in the hierarchy than the long ones, so the J's preferentially go into the short registers if there is room, and into the long registers next (instead of defaulting immediately to main memory). Variables of type I are allowed only in the long registers and main memory.

Also on the Cray-1, it was decided to allow a few large set variables, up to 4096 bits long, to be allocated directly to some of the eight vector registers (instead of only having large sets stored in main memory). By specifying 12K bits of memory type V, and a minimum allocation of 4K bits, it turned out to be easy to describe 3 vector registers for set temporaries, without allowing two shorter sets in the same register, and without allowing, say, a short integer to accidently occupy an entire vector register.

Memory hierarchy descriptions have also been done for the D.E.C. PDP-11, Univac 1108, and Burroughs B6700.

Summary.

The description mechanism appears to be flexible enough to make realistic descriptions of commercially available architectures, and this in turn allows a single machine-independent algorithm to do a reasonable job of allocating storage for any number of target machines. Through this mechanism, it is also possible to read in a program that has been mapped for one storage hierarchy, and convert it to a program matching another storage hierarchy. This in turn allows expermentation in computer architecture designs -- exploring the effects of having various numbers of registers, or exploring the range of offsets needed in a three- level hierarchy vs. the range of offsets in a single-level hierarchy, etc. We plan to do research in this area in the coming year.

Acknowledgements.

This research was supported by Los Alamos Scientific Laboratory under contract number NP8-2322E-01, and by Lawrence Livermore Laboratory under contract number 2462109.

References.

[1] Chow, F. 1979. "UCFORT," Stanford Artificial Intelligence Laboratory, Stanford, CA.

[2] Day, W.H.E. 1970. "Compiler assignment of data items to registers," IBM Systems Journal 9:4, pp. 281-317.

[3] Freiburghouse, R.A. 1974. "Register allocation via usage counts," Communications of the ACM 17:11, pp. 638-642.

[4] Harrison, W. 1975. "A class of register allocation algorithms," IBM Research Report RC-5342.

[5] Johnsson, R.K. 1975. An approach to global register allocation, PhD. thesis, Carnegie-Mellon University.

[6] Leverett, B.W. et al. 1979. "An overview of the production quality compiler-compiler project," Computer science report CS-79-105, Carnegie-Mellon University.

[7] Perkins, D.R. and Sites, R.L. 1979. "Machine-independent pascal code optimization," Proceedings of the SIGPLAN Symposium on Compiler Construction, Boulder, CO.

[8] Wulf, W., Johnsson, R.K., Weinstock, C.B., Hobbs, S.O., and Gesche, C.M. 1975. The design of an optimizing compiler, Elsevier, New York.