ISSN No. 0976-5697

Volume 2, No. 5, Sept-Oct 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Register Allocation and Instruction Scheduling for an Efficient Retargetable Compiler

Dr. Manoj Kumar Jain* and Veena Ramnani
Department of Computer Science
Mohanlal Sukhadia University Udaipur,
Rajasthan, India
manoj@cse.iitd.ernet.in, ramnaniv@yahoo.com

Abstract: System designers increasingly employ compilers not only for pure application programming after an ASIP's architecture is fixed but also for architecture exploration. During exploration, the designer tunes the initial architecture for a given application or application set. This tuning requires an iterative, profiling based methodology, by which the designer evaluates the cost-performance ratios of many potential architecture configurations. If C or C++ application programming is intended, the designer should apply a compiler-in-the-loop type of architecture exploration, thus avoiding a compiler and architecture mismatch.

The compiler designers often have difficulty ensuring good code quality because an instruction set designed primarily from a hardware designer's viewpoint fails to support their efforts. This problem is taken care by retargetable compilers. These compilers take processor description as input so that the machine code can be generated according to this description. Like traditional compilers, the code quality of retargetable compilers depends on the back end of the compiler. This paper describes a new register allocation and instruction scheduling technique.

***KeyWords* :** ASIP Design**,** Retargetable compilers, register allocation , instruction scheduling.

I. INTRODUCTION

Modern system-level design libraries frequently consists of Intellectual Property (IP) blocks such as processor cores that span a spectrum of architectural styles , ranging from traditional DSPs and superscalar RISC , to VLIWs and hybrid ASIPs. Embedded systems facilitate easy re-design of processor-memory based systems. The designer can incorporate modifications in the behavior and operation aspect of the architecture late in the design stage. ASIP are a compromise between the non-programmable ASICs and general purpose processors (GPP).

ASIP design [1][2][3][4] allows a wide range of memory organizations and hierarchies to be explored and customized for the specific embedded application. The ASIP designer is faced with the task of rapidly exploring and evaluating different architectural and memory configurations. Furthermore, shrinking time-to-market has created an urgent need to automatically generate compiler/simulator tool-kit.
There are two approaches for performance estimation using ASIP design: scheduler based approach and simulator based approach.

A. Scheduler Based Approach: In scheduler based approach the problem is formulated as a resource constrained scheduling problem. The application is scheduled to generate an estimate of the cycle count.
B. Simulator Based Approach: A retargetable compiler is constructed for each architecture to be evaluated. This compiler is used to generate code. This generated code is given as input to a retargetable simulator which is also designed for the same architecture under evaluation. This simulator generates the performance estimates and other statistics.

II. RETARGETABLE COMPILERS

Retargetable compilers are a promising approach for automatic compiler generation. A compiler is said to be 'retargetable' if it can be used to generate code for different processor architectures by reusing significant compiler source code. This has resulted in a paradigm shift towards a language-based design methodology using Architecture Description Language (ADL) for embedded System-on-Chip (SOC) optimization, exploration of architecture /compiler co-designs and automatic compiler/simulator generation. However, whatever approach is used, the performance depends on the back end of the compiler i.e. instruction selection, register allocation and instruction scheduling.

In this paper, we have discussed new approaches for both register allocation and instruction scheduling.

III. REGISTER ALLOCATION

The register allocation phase lies between optimization and the final code generation. When generating intermediate code, we freely use as many variables as required. Afterwards, we simply translate variables in the intermediate language one-to-one into registers in the machine language. Processors, however, do not have an unlimited number of registers, so we need register allocation to handle this conflict. The purpose of register allocation is to map a large number of variables into a small number of registers. This can often be done by letting several variables share a single register, but sometimes there are simply not enough registers in the processor. In this case, some of the variables must be temporarily stored in memory. This is called spilling.

Register allocation can be done in the intermediate language prior to machine code generation, or it can be done in the machine language. In the latter case, the machine code initially uses symbolic names for registers, which the register allocation turns into register numbers. Doing register allocation in the intermediate language has the advantage that the same register allocator can easily be used for several target machines (i.e. for retarget ability: it just needs to be parameterized with the set of available registers).

However, there may be advantages to postponing register allocation after machine code has been generated. In machine code generation phase, several instructions may be combined to a single instruction, and in the process a variable may disappear. Then, there is no need to allocate a register to this variable, but if we do register allocation in the intermediate language we will do so. Furthermore, when an intermediate language instruction needs to be translated into a sequence of machine-code instructions, the machine code may need an extra register (or two) for storing temporary values.

We have used a variation of linear scan algorithm [13] which is basically a global register allocation algorithm, and is not based on graph coloring.

A. Existing Register Allocation Approaches:

Global register allocation has been studied extensively in the literature. The predominant approach, first proposed by Chaitin in [5], is to abstract the register allocation problem as a graph coloring problem. Nodes in the graph represent *live ranges* (variables, temporaries, virtual/symbolic registers) that are candidates for register allocation. Edges connect live ranges that *interfere*, i.e., live ranges that are simultaneously live at atleast one program point. Register allocation then reduces to the graph coloring problem in which colors (registers) are assigned to the nodes such that two nodes connected by an edge do not receive the same color. If the graph is not colorable, some nodes are deleted from the graph until the reduced graph becomes colorable. The deleted nodes are said to be *spilled* because they are not assigned to registers. The basic goal of register allocation by graph coloring is to find a legal coloring after deleting the minimum number of nodes.

Chaitin's algorithm also features *coalescing*, a technique that can be used to eliminate redundant moves. When the source and destination of a move instruction do not share an edge in the interference graph, the corresponding nodes can be coalesced into one, and the move eliminated. Unfortunately, aggressive coalescing can lead to uncolorable graphs, in which additional live ranges need to be spilled to memory. [6] and [7] have focused on graph coloring and removed unnecessary moves in a conservative manner so as to avoid introducing spills.

Some simpler heuristic solutions also exist for the global register allocation problem. For example, lcc [8] allocates registers to the variables with the highest estimated usage counts, places all others on the stack, and allocates temporary registers within an expression by doing a tree traversal. Linear scan can be viewed as a global extension of a special class of local register allocation algorithms that have been considered in the literature [8] [9] [10] [11] which in turn takes their inspiration from an optimal off-line replacement algorithm that was studied for virtual memory [12]. Since the original description of the linear scan algorithm in [13], Traub in [14] have proposed a more complex linear scan algorithm, which they call *second-chance bin packing*. At a high level, the binpacking schemes are similar to linear scan, but they invest more time in compilation in an attempt to generate better code. The second-chance binpacking algorithm both makes allocation decisions and rewrites code in one pass. The algorithm allows a variable's lifetime to be split multiple times, so that the variable resides in a register in some parts of the program and in memory in other parts. It takes a lazy approach to spilling, and never emits a store if a variable is not live at that particular point or if the register and memory values of the variable are consistent.

Binpacking can emit better code than linear scan, but it does more work at compile time. Unlike linear scan, binpacking keeps track of the “lifetime holes" of variables and registers (intervals when a variable maintains no useful value, or when a register can be used to store a value), and maintains information about the consistency of the memory and register values of a reloaded variable. The algorithm analyzes all this information whenever it makes allocation or spilling decisions.

B. The Revised Linear Scan Algorithm:

We have used a variation of linear scan algorithm proposed by Poletto in [13]. The original algorithm is used for global allocation, but we have employed the same methodology for local allocation as well, taking care of the variables which are being used across the block. We shall be performing the register allocation considering the intermediate representations i.e. three address code already generated. We also assume some kind of ordering on this three-address code.

For every basic block, we calculate the starting point of use and the last point of use of the variables, constants and temporary variables. As we scan each line to find the use of the above, we cross the block boundary to find the last use. For every variable and constant we note down the use in the basic block as well as the last use in the whole program. Calculating the last use in the program, gives us the last basic block where the variable will be used. The advantage of finding the last use in block and last use in program helps the register allocator decide on two things:

a. If the variable is being used in the current block and has no use outside the block , the register allocator will free the register after the point of last use in the basic block
b. If the variable has the last use in program outside the current basic block , the register allocator will not free the register immediately on exit from the block

We are not keeping separate registers for global and local allocation. Initially, all the registers are used for local allocation. On exit from the basic block, only the registers holding values which have no next use will be freed and rest registers will be maintained as it is. Demarking the registers for local and global allocation leads to under utilization of registers and unnecessary spilling. Priority of register allocation is given to variables to be used within the current basic block i.e. given a set of registers; the allocator would try to keep all the variables to be used within a basic block in the registers. In the process, it may spill variables if they are not required in the current block.

For local allocation, when we need to allocate a register to some variable/constant/temporary and there is no free register, then the register allocator calls a procedure free_register, which will search –

a. For register holding variable/constant/temporary which are dead i.e. which are no longer required locally or globally
b. If no such register is found, then it searches for register holding variable/constant not required locally but globally and they will be freed i.e. the contents are removed and not spilled.
c. If the register allocator fails to find such a register satisfying (1) and (2) i.e. the register allocator is not able to find a register that can be freed, then spilling is performed.

As can be observed, spilling is the last option. Spilling will select a register based on the usage count of the variable/constant/temporary it is holding. The one with the lowest usage count will be spilled.

At the end of each basic block or at the entry of the next basic block , we will not load all the variables with last use in program in following blocks, in the registers because this approach may not leave empty registers for local variables/constants/temporaries. But, definitely , we would free the registers holding values which are dead.

Global allocation does not require another algorithm.It is being taken care of as we are allocating registers for local values. Since we have calculated the last use of the value in program , we know which variables/constants will be used outside the basic block and we try to keep them in register as long as possible , till the register is required for local allocation. In that case, we spill these values to accommodate local values and these global values will be loaded as and when required like local variables.

At the time of spilling , if we have more than one global value that has so next use in the current basic block , but is live on exit , we use the following criteria for selecting the register to spill :

i. Lowest usage count
ii. Distance of next use from the current line

Generally , we have two terms register allocation and register assignment.Register allocation is to decide which values will reside in registers and Register assignment is in which value should each value reside.

In our approach , there is no such thing like register allocation as every variable/constant/temporary is being assigned a register and if a free register is not available either a register is freed or spilled.Also , we have mentioned initially that we are not demarking registers for local and global allocation , hence every register can be used to hold every variable/constant/temporary ,whether inside the basic block or across the block boundary for global use. It is observed that this approach minimizes unnecessary memory loads and stores , which reduces the code size and cycle count drastically. Also , not reserving registers for global and local usage helps proper utilization of registers.

The formal algorithm for the above methodology is given below :

```
Procedure last_use_info ()
For every basic block do
     For every variable/constant/temporary do
       Begin
         Calculate the last use in basic block
           Calculate the last use in program
       End
End Procedure
Procedure get_free_register ()
For all the registers in the register_file do
Begin
If the register is holding a value which is dead, mark it as "empty"
If the register is holding a value that will not be used in the current block
But will be required globally, mark it as "empty"
End
Return the empty register
End Procedure
Procedure register_allocator()
    Call last_use_info
    Initialize all registers to "empty"
     For every basic block do
      begin
        For every variable/constant/temporary do
          Begin
         Reg=Get_free_register ()
         if there is a free register then
             Allocate reg to the variable/constant/temporary
         else
           Find registers holding values which will not be used in the block
           Select the one with least usage count
                   Spill the value in the selected register
                 End if
            End
        At the end of the basic block:
            Free the registers, which are holding values which are dead
      End
End Procedure
```

IV. INSTRUCTION SCHEDULING

Instruction scheduling, an important optimization in modern compilers, attempts to minimize the execution time for a set of instructions by orchestrating the order of their execution. Scheduling is particularly important for wide-issue machines, where instruction level parallelism (ILP) is a key source of performance, as it is responsible for presenting sets of instructions for concurrent execution.

In order to generate high-quality code for modern processors, a compiler must aggressively schedule instructions, maximizing resource utilization for execution efficiency. For a compiler to produce such code, it must avoid structural hazards by being aware of the processor's available resources and of how these resources are utilized by each instruction.

The problem facing an instruction scheduler is to reorder machine code *instructions* to minimize the total number of *cycles* required to execute a particular instruction sequence. Unfortunately, sequential code executing on a pipelined processor inherently contains *dependencies* between some instructions. Any transformations performed during instruction scheduling must preserve these dependencies in order to maintain the logic of the code being scheduled. In addition, instruction schedulers often have a secondary goal of minimizing register lifetimes.

Because of the overlap of the execution of instructions in a pipeline structure, the results of an instruction issued in cycle *i* will not be available until cycle *i*+*n*, where *n* is the length of the appropriate pipeline constraint (i.e.: the latency of the instruction). If an instruction issued between cycles *i* and *i+n* attempts to reference the result of the instruction issued at cycle *i*, a data dependency has been broken. An interlocked pipeline will detect this situation and *stall* the execution of the offending instruction until cycle *i+n*, wasting valuable processor cycles. Even worse, on a pipeline without interlocks the code will produce incorrect results, since the value being referenced will be the result of some other operation.

Automatically finding an *optimal* schedule for a code sequence is NP-complete in its most general form (with arbitrary pipeline constraints), since all legal schedules must be examined to determine the optimal one.

a) Types of Dependencies:

In practice, three types of data dependency need to be considered.

i. A *true* or *read-after-write* (RAW) dependency occurs when one instruction reads the result written by another. The read instruction must follow the write instruction by a suitable number of cycles for the result to be read without stalling.
ii. An *anti* or *write-after-read* (WAR) dependency occurs when one instruction writes over the operand of another. The read instruction must occur before the write instruction by a suitable number of cycles for the value to be safely read without stalling the write instruction.
iii. An *output* or *write-after-write* (WAW) dependency occurs if two instructions write to the same target, with the result of the logically first instruction never being used.

The first two types of dependencies – *read-after-write* and *write-after-read* dependencies – are "real" in the sense that they represent the genuine logic of the program. Any *write-after-write* dependencies are usually artifacts of either the compilation process or structured programming practices, and can often be removed by dataflow optimizations such as dead-code elimination or partial redundancy elimination.

In addition to data dependencies, it is important to remember that programs also contain *control* dependencies which specify the logical structure of the program and the order in which certain operations must be performed. For example, the *then* part of an *if* statement should not be executed until the test has been performed, and then only if the test was true. Loops, conditionals, function calls and so on all represent control dependencies.

A. Existing Approaches for Instruction Scheduling:

There have been many heuristic attacks on the instruction scheduling problem. Most of these have followed the general approach set down by John Hennessy and Thomas Gross in 1983 [15] is based on standard list scheduling. Another notable approach is the modification of Sethi-Ullman expression evaluation to compensate for some forms of pipeline delays [16], however that approach cannot easily be extended to handle the more complex scheduling problems presented by modern superscalar processors.

List scheduling approaches work by maintaining a *ready list* which contains all the instructions that can legally be executed at a particular point in time. The scheduler repeatedly chooses an instruction from the ready list, removes it from the list and issues it (inserts it into the final schedule). This in turn makes other instructions ready, and these are added to the list. The key to the whole process is the set of heuristics used to select the *best* instruction to issue.

In general, scheduling heuristics are used in combinations, called *priority functions*, to try to select the best instruction to be issued for a particular situation. Although there are literally hundreds of popular heuristics in use today, they can be grouped into a few broad categories as in [17].

B. The Revised List Scheduling Algorithm:

In our approach we have used a variation of list scheduling algorithm, in the sense we have combined register allocation with list scheduling. The three-address code is converted to the machine code; the instruction selection phase selects appropriate opcode from the instruction set and the registers are allocated to the operands. Once the machine code is generated, we perform instruction scheduling.

In order to perform scheduling, we need to calculate data dependencies for which we create dependency graph or data precedence graph (dpg). The nodes of the data precedence graph are the machine code instructions. The dependencies are calculated between instruction operands i.e. the registers and the memory addresses. The data dependencies are depicted by drawing edges between the nodes. The dependency is calculated between register and memory operands and not the actual operands. Since, instruction scheduling is only about reordering the instructions, so as to produce an optimal sequence; we don’t need to perform register allocation again.

Using the above approach solves the phase ordering problem as well. The phase ordering problem is to decide whether to perform instruction scheduling before or after register allocation. In our approach, we are performing instruction scheduling after register allocation. Since the scheduling is performed considering register and memory operands, no false dependencies are created, hence there is no need to perform repeated register allocation and scheduling.

First, the dpg (data precedence graph) is built, each instruction is a node and the data dependency between instructions is shown by drawing edges between them. Next, priorities are assigned to each node in the graph. There are several different heuristics that can be used to assign priorities. A common and effective strategy is to use the latency-weighted depth of the node. The depth of a node n is the length (number of nodes) of the longest path in the dpg from n to some leaf (including n and the leaf). The latency-weighted depth is computed the same way, but the nodes along the path are weighted using the latency of the operation the node represents.

The formula below shows how the priority of a node is calculated:

$$\text{priority}(n) = \begin{cases} \text{latency}(n) & \text{if } n \text{ is a leaf.} \\ \max\left(\text{latency}(n) + \max_{(m,n)\in E}(\text{priority}(m))\right) & \\ \text{otherwise} & \end{cases}$$

If two or more nodes have the same priority as calculated with the above formula, we calculate the no.of dependents of the nodes and the one with highest no. of dependents is selected, because delaying the node (or instruction) with large number of dependents will result in a longer schedule.

The algorithm maintains two lists: *ready* and *active*. *Ready* consists of all the instructions that are ready to be scheduled i.e. all its predecessors have been scheduled. *Active* consists of instructions that are being implemented. Initially, *ready* consists of nodes with no predecessors. Starting at cycle 1, the list scheduler places operations into the schedule cycle by cycle. Any operation that is "ready" at cycle X (i.e. all its operands have been computed and satisfies resource constraints), is a candidate to be scheduled at cycle X. The priorities computed in the previous step are used to determine which ready operation to schedule, by selecting the highest priority operation first. Any tie in the priority of two operations is broken in favor of the instruction with more no. of dependents. The instruction is selected, removed from *ready* and moved to *active*.

At every clock cycle, we check whether any instruction already in *active* list has completed execution, then we free the resources assigned to that instruction and remove it from *active* list. If this instruction has instructions depending on it, they are now placed in the *ready* list as they can now be scheduled. This process continues till *ready* and *active* have no elements or in other words there are no instructions to be scheduled.

The formal algorithm is given below:

Input: Data Precedence Graph (DPG) with priorities assigned to each node

Output: A schedule containing all nodes in the graph that satisfies the precedence constraints in the DPG and the resource constraints of the machine

Algorithm:

```
Cycle = 1
Ready = Leaves of DPG
Active = ϕ
While (Ready U Active <> ϕ)
{
        For op= (all nodes in Ready in descending priority order)
                If (a functional unit exists for 'op' to start at 'cycle')
                {
                -remove 'op' from Ready and add 'op' to Active
                - add 'op' to schedule at time 'cycle'
                - make operands available in registers and allocate a
register for target
                }

        End for
        Cycle = cycle +1
        Update the Ready Queue
}

For op= (all nodes in Active)
        If ('op' finishes at time 'cycle')
        {
                -remove 'op' from Active
                - Check nodes waiting for 'op' in DPG and add to
'ready' – if all operand are       available
        }
End for
```

V. CONCLUSIONS

In this paper we have given new approaches for register allocation and instruction scheduling. The objective for these two approaches is to generate efficient code in terms of reduced cycle count .In the approach for register allocation ,there is nothing like allocation i.e. selection of values which will reside in registers, as our algorithm allocates register to every value. Our algorithm also does not require def-use analysis. Instead, we linearly scan the code to know the last use of a value in the program. We try to retain a value in register till its last use point is crossed. The above approach reduces spilling and unnecessary moves. The approach employed for list scheduling avoids the phase coupling problem by combining register allocation with scheduling.

VI. REFERENCES

[1]. Jain,M.K., Kumar,A., Balakrishnan,M. and Gangwar,A.(2005) Customizing Embedded Processors for Specific Applications, In proceedings of Recent Trends in Practice and Theory of Information Technology, Proc. of NRB Seminar, 10-11 January 2005, NPOL, Cochin, pp. 261-284

[2]. Jain,M.K., Balakrishnan,M.and Kumar,A.(2001) ASIP Design Methodologies: Survey and Issues, In proceedings of the Fourteenth International Conference on VLSI Design, 2001, 3-7 Jan. 2001, Pages: 76-81

[3]. Jain,M.K., Balakrishnan,M. and Kumar A.(2004), Efficient Technique for Exploring Register File Size in ASIP Design', IEEE TCAD of VLSI, vol. 23, No. 12, pp. 1693-1699, Dec. 2004.

[4]. Jain,M.K. and Gaur,D.(2011)ASIP Design Space Exploration :Survey and Issues ,International Journal of Computer Science and Information Security,ISSN – 19475500,Volume 9, Issue 4 ,pg. 141-145

[5]. Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. (1981). Register allocation via coloring. Computer Languages 6, 47-57.

[6]. Briggs, P., Cooper, K., and Torczon, L. (1994). Improvements to graph coloring register allocation.ACM Transactions on Programming Languages and Systems 16, 3 (May), 428-455.

[7]. George, L. and Appel, A. (1996). Iterated register coalescing. ACM Transactions on Programming Languages and Systems 18, 3 (May), 300-324.

[8]. Fraser, C. W. and Hanson, D. R. (1995). A Retargetable C Compiler: Design and Implementation. Benjamin/Cummings, Redwood City, CA.

[9]. Freiburghouse, R. A. (1974). Register allocation via usage counts. Communications of the ACM 17, 11 (November), 638-642.

[10]. Hsu, W.-C., Fischer, C. N., and Goodman, J. R. (1989). On the minimization of loads and stores in local register allocation. IEEE Transactions on Software Engineering 15, 10 (October), 1252-1260.

[11]. Motwani, R., Palem, K. V., Sarkar, V., and Reyen, S. 1995. Combining Register Allocation and Instruction Scheduling (Technical Summary). Tech. rep., Courant Institute, New York University. July. TR 698.

[12]. Belady, L. A. 1966. A study of replacement algorithms for a virtual storage computer. IBM Systems Journal 5, 2, 78-101.

[13]. Poletto, M., Engler, D. R., and Kaashoek, M. F. 1997. tcc: A system for fast, flexible, and high-level dynamic code generation. In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation. Las Vegas, NV, 109-121.

[14]. Traub, O., Holloway, G., and Smith, M. D. 1998. Quality and speed in linear-scan register allocation. In Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation.

[15]. Hennessy, J.,Jouppi, N.,Przybylski,S.,Rowen,C.and Gross,T.(1983) Design of a high performance VLSI processor , Technical Report No. 236 ,Computer systems Laboratory , Departments of Electrical Engineering and Computer Science , Stanford University,C

[16]. Proebsting, T.A. and Fischer, C.A. (1991) Linear-Time optimal code Scheduling for delayed-Load architecture, PLDI'

[17]. Krishnamurthy, M. (1990) A brief survey of papers on scheduling for pipelined processors, Sigplan, 25(7):97-106