Global Instruction Scheduling In Machine SUIF

Gang Chen and Michael D. Smith
Division of Engineering and Applied Sciences, Harvard University
{cheng, smith}@eecs.harvard.edu

Abstract

Machine SUIF is a retargetable compiler backend designed by the HUBE research group at Harvard University. It extends the SUIF compilation system for machine-specific compilation and optimization. In this paper, we present one such optimization phase, global instruction scheduling for acyclic graphs. In a single scheduling framework, we are implementing both trace-based and DAG-based scheduling. Both are designed for a wide class of programs, even those without strongly biased execution paths, and both employ heuristics to avoid penalties from incorrect speculative execution. In addition, we propose the use of multiple-path code motions for further avoiding redundant compensation code. To describe accurately and efficiently the target machine, our schedulers use finite state automaton models of hardware resources. Our ultimate goal is to produce a general and expressive framework that relies on abstraction and encapsulation to facilitate the quick construction of powerful and sophisticated instruction schedulers.

1 Introduction

A close coupling of architecture design with compile-time optimization is increasing the need for retargetable compilation systems. In practice, the engineering of an optimizing compiler is a hard and time-consuming process. For a research environment, this task is even more difficult since the compiler must not only adapt to the evolution of one specific architecture family but also support the exploration and evaluation of radically innovative architectural ideas. Machine SUIF [Smit96] is a retargetable compiler backend that extends the SUIF compilation system [Wils94] for machine-specific compilation and optimization. In this paper, we describe our current and future efforts in global instruction scheduling under Machine SUIF.

Base SUIF is object-oriented and machine-independent. Machine SUIF extends SUIF by encapsulating the specifics of an architecture's instructions in one of several general SUIF instruction classes [Smit96]. Under this approach, all of the sophisticated machine-independent analysis and optimization utilities that operate on the SUIF intermediate instructions can be re-used in backend, after code generation. By encapsulating target-specific machine instructions into general instruction classes, we can build Machine SUIF passes that perform machine-specific optimizations in a machine-independent manner. For example, we have a single piece of code for register allocation[1] that sees only the salient aspects of the target architecture. Unnecessary information, like the actual specifier of the first general-purpose temporary register in a particular architecture, is hidden. This approach made it very easy for us to add new compilation targets without having to change any of the code in the register allocation pass.

Like register allocation, instruction scheduling appears at first glance to be so closely coupled to the microarchitecture of the compilation target that one would have to develop a unique piece of code for each target. Under more detailed investigation however, one can separate instruction scheduling into two components: a machine-independent program transformation that considers orderings of instructions while maintaining program semantics, and a machine-specific resource manager that directs the mapping of instructions onto the target hardware.

Viewed as program transformation, instruction scheduling is simply a game of moving instructions around to reorder the instruction sequences and reduce the overall execution time. Certainly there are rules to this game that must be followed to maintain correct program semantics. First and foremost, the transformation must maintain true data dependences no matter whether we schedule locally (within a basic block) or globally (across basic blocks). For global instruction scheduling, since instructions can be moved across basic block boundaries, more complicated control flow and data flow constraints must be considered. For instance, moving an instruction across a control-flow graph (CFG) join point requires duplication of the instruction in every predecessor of the join point. Moving an instruction above a CFG branch point must be legal and safe [Smit92], i.e. the movement should not violate any data dependences that extend above the branch point and should not cause exceptions that would not have appeared in the original instruction ordering. According to the shape of regions on which transformations are performed, global instruction scheduling within an acyclic CFG can be classified as *trace-based* scheduling and *DAG-based* (or region-based) scheduling. Trace-based scheduling picks a linear sequence of basic blocks as the scheduling region, while DAG-based scheduling picks an arbitrary acyclic region of basic blocks.

Viewed as mapping function, instruction scheduling is conducted with respect to a target machine model. Usually the architecture features directly represented by instruction sets are not enough for building such a model. Instruction schedulers need further knowledge about how every instruction is executed, such as how many pipeline stages it takes, what hardware resources it uses at every stage, and what structural hazards need to be avoided. A good abstraction for such architecture features is hardware resource usage vectors. Traditionally, instruction schedulers have used hardware resource usage vectors directly, and the hardware resource usage states are simulated with resource reservation tables. In Machine SUIF, we adopt a different approach introduced by Bala and Rubin [BaRu95]. Hardware resource usage vectors are preprocessed into finite state automata (FSA) that encode all the possible machine resource usage states. At compile time, the instruction scheduler is guided through simple querying of the corresponding automaton model. As described by Bala and Rubin, this approach has significant time and space advantages over the traditional approach [BaRu95]. From our point of view, this approach is also noteworthy for its clear-cut separation between the scheduling algorithm and the implementation specifics of the target architecture.

In the next section, we present the context necessary to understand our global instruction scheduling algorithms. Section 3 describes our implementation of a trace-based scheduler, while Section 4 contains an evaluation of this scheduler. Though our compiler currently is missing several features that would improve the code produced by our trace-based scheduler, it is clear from an inspection of

1. We base our register allocator on the work of George and Appel [GeAp96].

the scheduled code that we could do significantly better if the global scheduler were able to fill issue slots from blocks other than those on the currently-selected trace. With this as motivation, Section 5 discusses the replacement of a few parts of our trace-based scheduler to produce a DAG-based scheduler. Finally, Section 6 summarizes the status of our project and outlines our continuing work in this area.

2 Instruction Scheduling in Machine SUIF

Figure 1 shows the general flow of the SUIF compilation process, and in particular, the structure of the backend. The Machine SUIF portion of the compilation process begins with code generation where individual code generators are needed for each compilation target. These code generators translate SUIF intermediate instructions into target-specific instructions.[2] However, no matter which specific instruction set architecture is targeted, the output intermediate form is always the same, Machine SUIF. The common intermediate representation for the machine-specific instructions enables us to build machine-specific optimization passes, such as instruction scheduling and register allocation, that are written independent of the unnecessary machine target details.

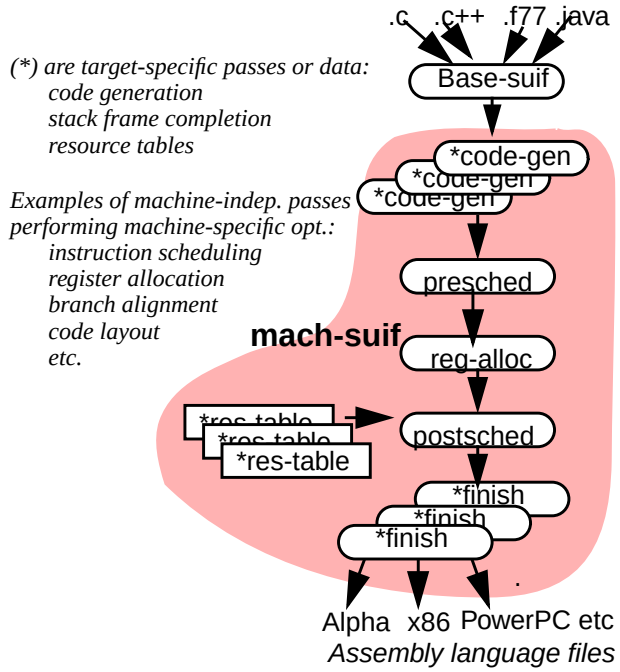

Figure 1. Structure of compilation flow in Machine SUIF

In general, we design SUIF analysis and optimization passes, including those in the backend, to make it very easy to reorder and repeat passes. We currently run instruction scheduling after register allocation, however, as Figure 1 shows, we could run instruction scheduling both as a pass before and as a pass after register allocation, as suggested by Hwu et al. [Chan95]. Structurally, the only difference between the pre- and post-scheduling passes is the heuristics used to drive the selection of instructions (e.g. pre-pass schedulers typically try to perform code motions without increasing register pressure).

Global instruction scheduling consists of several basic steps. First, the scheduler selects a region to schedule. For a trace-based scheduler, this region will comprise a linear, acyclic sequence of basic blocks. For a DAG-based scheduler, this region comprises an arbitrary acyclic graph of blocks. In both cases, a region typically is seeded by the unscheduled block with the highest probability of execution and typically ends at loop-back edges, low-frequency edges, and edges to already-scheduled basic blocks. Once a region is selected, the next step is to summarize the data dependence and control dependence constraints that must be maintained during any code motion. We describe these structures in more detail in the later sections. At this point, we are ready to begin scheduling. Our algorithm, like most current algorithms, schedules the region one basic block at a time. Though there are several techniques for scheduling a basic block, most are based on a variant of list scheduling [Davi81]. In particular, we schedule blocks in a top-down, cycle-scheduling approach that is efficient in terms of compile-time and lends itself nicely to the upward movement of instructions. In each cycle, the algorithm selects the highest-priority, data-ready, unscheduled instruction and then checks to see if the current cycle has the resources available to execute this instruction. If the resources are available, the instruction is marked as scheduled in this cycle, and the resources required are marked as busy for the latency of the instruction. If not, the algorithm simply moves on to consider the next highest-priority, data-ready, unscheduled instruction. We repeat this operation until all of the resources in the current cycle are busy or all of the data-ready instructions have been considered. At this point, the algorithm advances to schedule the next cycle. This process is repeated until all of the instructions are scheduled.

In Machine SUIF, the output of our scheduling pass is a list of SUIF instructions (containing machine-specific instructions) with an annotation[3] that records the result of the scheduling process. Figure 2 below illustrates the structure of our scheduling annotation. We discuss the details of the fields after we overview the mechanism for maintaining resource information. One important point, however, is that our scheduling pass does not insert NOPs into the instruction list. Instead, the information about necessary NOPs is recorded in the annotations. There are several benefits to this approach. First, the result of instruction scheduling can be transparent to other passes. Thus, it is trivial to run instruction scheduling either before or after register allocation. Furthermore, these annotations make it trivial to translate the instruction list into the assembly-language listing for a VLIW machine. This idea is similar to the one used by Moon and Ebcioglu [MoEb92] to target the same scheduler and optimizer at multiple different microarchitectures.

As mentioned earlier, we employ the FSA technique described by Bala and Rubin [BaRu95] to maintain and manage the machine resource information. Briefly, this technique involves the generation of two sets of automata. One set (the *forward* automata) simulates the transition of hardware resource usage states in instruction execution order. The other set (the *reverse* automata) simulates the transition of hardware resource usage states in an order opposite to the instruction execution order. These automata are built only once, when the compiler is constructed, for each target architecture. During the compilation process as shown in Figure 1, we simply load the transition table part of the automata as a part of the machine description information.

Figure 2 shows the structure of the scheduling annotation associated with every Machine SUIF instruction. For each instruction, we record *fstate_bot*, the forward state after this instruction is issued,

2. We currently have code generators for the Digital Alpha, MIPS, and PowerPC instruction set architectures, and we are working on an x86 code generator.

3. An annotation is just a SUIF mechanism for attaching a piece of information to a SUIF object, like an instruction.

and *rstate_top*, the reverse state before this instruction is issued. For the instruction at the end of a cycle, we include extra information that records the NOPs which need to be packed before the cycle can actually advance (*nop_pad*) and information about the forward state after the cycle advances (*fstate_bot_e*). Similarly, for the instruction at the beginning of a cycle, we include extra information to record how many empty cycles will elapse before this cycle starts (*cycle_pad*) and information about the reverse state before these empty cycles (*rstate_bot_e)*. Unfortunately, we do not have the space to provide a full explanation of the use of this extra information.

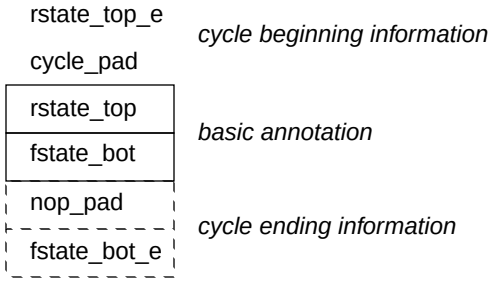

Figure 2. Structure of our scheduling annotation. This annotation is attached to each scheduled instruction.

There are several advantages to the use of FSA and the corresponding scheduling annotation. If we had used hardware resource reservation tables, we would have had to record a resource reservation bit matrix with every instruction. Using automata is much more space efficient. For Alpha 21064, we require only 32 bits to record a forward or reverse machine state. Furthermore, checking for available resources in a cycle, a frequent operation in the innermost loop of our scheduling algorithm, is simply time-efficient table lookup. The low space overhead allows us to keep the machine state at every instruction point, and this then frees us to play with the way in which we create the final instruction schedule. As we describe in the later sections, it is very useful to be able to place an instruction into an empty slot in an already scheduled instruction sequence. As proposed by Bala and Rubin [BaRu95], the scheduler can decide if a new instruction can fit in an empty slot by simply checking to determine if there is legal transition from the forward state above the empty slot and legal transition from the reverse state below the empty slot.

3 Trace-based Scheduling

Within the same scheduling framework, we are implementing two scheduling approaches, trace-based and DAG-based scheduling. Currently, these schedulers perform global code motions only within acyclic CFG regions; however in the future, we will extend them to support code motions across loop-back edges through loop unrolling and software pipelining.

We implement the trace-based scheduling algorithm presented by Smith [Smit92]. The central idea of this algorithm is to consider the effects of a global code motion while scheduling. It is well known that, although global instruction motion may shorten the being-scheduled path, it may also impose penalties on other execution paths. The classical trace scheduling approach [Fish81] performs global code motions without considering the penalty to off-trace paths. This approach strongly biases the resulting schedule toward those paths selected early in the scheduling pass. This may turn out to be a poor policy for applications without strongly biased execution paths.

Briefly, our trace-based scheduling algorithm begins by the seletion of a trace. In Figure 3(a), the algorithm has picked a trace which includes blocks 1, 2, 3 and 4. Once a trace is selected, we construct two data structures. One is the Data Dependence Graph (DDG), which summarizes all of the data dependences among the on-trace instructions as well as any imposed constraints among on-trace instructions, such as the forced order of branch instructions. Our other structure summarizes the control dependence information required during scheduling. In other words, it summarizes the off-trace constraints required to maintain program semantics during a global code motion. For example, in Figure 3(b), since block 2 has an off-trace successor, its off-trace constraints include the off-trace liveness information.

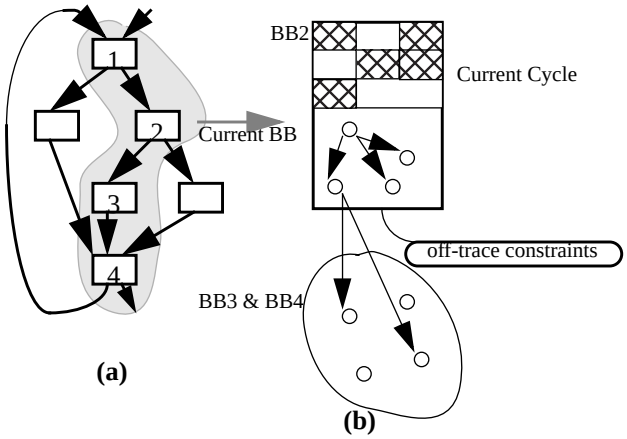

Figure 3. Trace-based Scheduling

Code motion only in need: On-trace basic blocks are scheduled one by one in a top-down order. As shown in Figure 3(b), assume that block 1 has already been scheduled and that Block 2 is being scheduled. As stated earlier, the scheduling of block 2 proceeds cycle by cycle. During the scheduling of each cycle, we give higher priority to those instructions that are local to the current block. In Figure 3, these are the nodes drawn inside the box representing block 2. Therefore, when there is an empty instruction issue slot, we first try to fill it with a local instruction. We consider global instructions, the nodes drawn in the cloud labeled as BB3 & BB4, only when no local instruction fits. This heuristic limits unnecessary upward global code motions that may penalize other paths.

In addition to this heuristic, our trace-based algorithm includes two optimizations that also help to reduce the execution-time penalty of compensation code. First, as shown in Figure 4, we introduce a new code motion call *Jump* to avoid unnecessary duplication in some program graphs. Our algorithm recognizes those program graphs where the original basic block containing the moving instruction is *control equivalent* with the basic block currently being scheduled. Two blocks are control equivalent if the first dominates the second, and the second post-dominates the first. In these cases, we can move an instruction without duplication if neither its sources nor its destination are written in the blocks between and its destination is not live at the top of the subgraph.

Second, we always absorb duplicated instructions into the off-trace sequences. When an instruction is moved above a join point, a duplication needs to be put into all other off-trace blocks which connect to this join point. If such a block is not yet scheduled, this duplicated instruction will be scheduled into the off-trace sequence later. However, if such a block is already scheduled, rescheduling is expensive. One traditional choice is simply forbidding this motion; the other is to put the duplication at the end of the block in its own

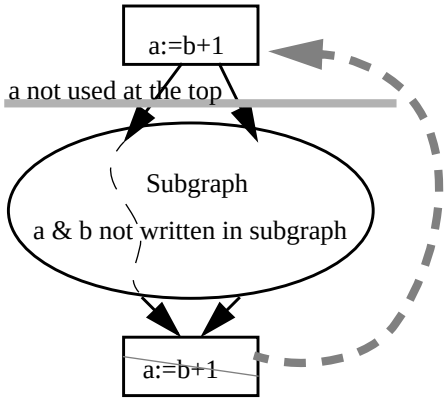

Figure 4. Jump Operation

cycle. This may lead to an inefficient schedule. Using the FSA information attached to every scheduled instruction, it is very easy for our algorithm to determine if an instruction can be inserted into an empty slot in the scheduled code [BaRu95]. We take advantage of this to insert compensation code into already scheduled blocks whenever possible.

4 Experimental Results

We have implemented a prototype of our trace-based scheduler, and we have performed some preliminary experiments to evaluate the current prototype. This section presents the results of that testing and highlights the shortcomings of our current implementation.

Program	Without Global Scheduling			With Glb.Sched. **%diff**		
	Cycles	Instrs	CPI	Cycles	Instrs	CPI
wc	1.8644 M	1.0495 M	1.7764	-1.72	3.8	-5.3
compress	18.459 M	11.232 M	1.6434	-3.1	4.2	-7.0
uuencode	3.1033 M	1.9563 M	1.5863	-3.4	3.5	-6.7
li	35.515 M	22.222 M	1.5982	-0.8	2.3	-3.0
powell	279319	144449	1.9337	-1.3	0.6	-2.0

TABLE 1. Cycle and instruction count effects of the trace-based scheduler.

The testing set includes two SPEC integer benchmarks, two small UNIX utilities and one floating point computation program selected from text book [Press et al.]. Table 1 shows measurements on an Alpha 21064 using the ATOM-Zippy tools from Digital. Here since the purpose is to expose the effect of global instruction scheduling instead of measuring the overall performance, a perfect memory system model is used. Therefore ATOM-Zippy is actually just used as a fast instruction simulator. One group of results shows code generated without the global instruction scheduling phase; the other group shows code with global instruction scheduling. Notice that, the output of Machine SUIF is Alpha assembly code, so the final executables are generated using native Alpha assembler. Since the Alpha assembler does local instruction scheduling in both cases, we are comparing trace-based global instruction scheduling with local instruction scheduling.

For each program, Table 1 shows the number of instructions executed, and the number of cycles taken, and the CPI. First of all, these results show consistent result: the number of instructions increases after instruction scheduling because of compensation code. On the other hand, the total number of cycles decreases after

Program.Procedure	Instrs	Height	Average
wc.main	78	71	0.9103
compress.compress	411	692	1.683698
li.xlsave	123	123	1.0
li.xleval	87	133	1.5287
uuencode.encode	359	423	1.1783
powell.brent	310	524	1.6903

TABLE 2. Limited parallelism in data dependence graphs

global instruction scheduling. However, the cycle improvement is currently small. .

Cycle improvements were limited by low parallelism in the current DDG. Table 2 shows some statistics about the DDGs constructed for traces. Several procedures are selected from the testing set. The first column shows the total number of instructions in each procedure; the second column shows the sum of height of DDGs constructed in each procedure. The third column shows the arithmetic average of the height over the number of instructions. Practically these numbers can give a rough measurement about the DDG; they are very thin. This is caused by several problems. First, we are testing instruction scheduling after register allocation, and register allocation adds significant amount of data dependence. Second, almost no memory disambiguation is applied during data dependence analysis.

However, no register renaming or operation combining is applied during instruction scheduling and code motion. Register renaming and operation combining are very important in eliminating some data dependences imposed by register allocation. Figure 5 shows a code segment, and the lists inside boxes show the sequence before instruction scheduling. Block 7-8-9-10-14-15-18 is the first trace chosen on this subgraph. Without register renaming and operation combining, the scheduler is not able to make significant changes. The root of the trouble is instruction 18. It can not jump to block 7 because register 2 is used in block 14. And all other instructions below instruction 18 depend on it. The sequences outside the boxes and without italics shows the on-trace schedule if register renaming and operation combining are used. Instruction 18 moves to block 7 after its destination is renamed from register 2 to register 8. A register copy instruction is left at B18. Then instruction 19 is moved to block 9 after combining with the newly generated copy instruction. So one of its sources is changed from register 2 to register 8. Instruction 20 is moved to B14 after renaming. Instructions 21 and 22 are scheduled locally with their sources changed because of combining. Dead copy instructions can be deleted later.

There are several other issues which also impact scheduling results:

- Currently, our compiler schedules assembly language instructions. Some of these instructions are actually macro instructions that are expanded by the assembler into multiple machine instructions. These expansions are obviously not considered by our scheduler. In the future, we will perform these expansions in Machine SUIF so that we always schedule machine instructions.
- Traces are picked using simple policies without profile information. We select the fall through path for forward branches and the taken path for backward branches.

- There is no general approach to eliminate redundant compensation code. The jump described earlier suppresses compensation code for one special case.

Even if the above problems are solved, there are other problems preventing trace-based scheduling from exploiting more ILP. Continuing the example from Figure 5, after the trace 7-8-9-10-14-15-18 is scheduled, the remaining CFG is cut into two small regions. Further trace selection and scheduling will be confined in each region, which has very limited ILP. Trace-based scheduling never allows code motion across trace boundaries, i.e. we can not move instructions from these small regions into the holes of already-scheduled basic blocks, or from one small region to another. On Figure 5, italicized instructions show possible improvement if scheduling is not confined to these two small regions. Instruction 8 can replace the nop in block 9 after renaming. Instruction 16 can be moved up from B17 with its destination renamed, then filled in the holes of block 10, 12 and 13. On the other hand, in block 13 and 17, the newly generated copy instructions can be scheduled in the same cycle with local instructions after combining.

We would prefer to schedule all the blocks in an acyclic CFG subgraph together. This can expose more ILP than a trace because it moves up instructions from multiple execution paths. Section 5 presents the DAG-based scheduling approach we are implementing.

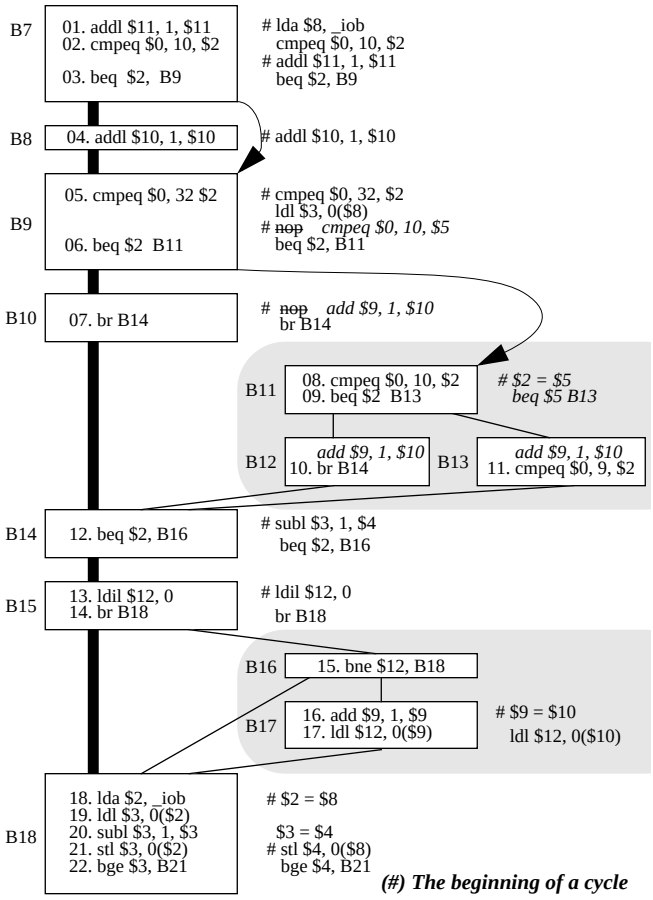

Figure 5. Trace sched. with register renaming and operation combining

5 DAG-based scheduling

As a by-product of implementing DAG-based scheduler, a common scheduling framework has been extracted from the trace-based scheduler, which provides the following classes:

- Control flow graph for scheduling
- Data flow analysis
- Simple data dependence analysis
- Data dependence graph
- Resource usage FSA
- Instruction latency matrix
- Instruction list with scheduling annotations

Deriving such a common scheduling library enables us put more focus on new algorithm when constructing and experimenting with the scheduler, and ultimately facilitates research in this area.

Similar to our trace-based scheduler, each time our DAG-based scheduler picks an unscheduled acyclic subgraph in the CFG, it schedules this DAG region one basic block at a time in top-down order. The scheduling of each basic block is divided into two phases. Phase 1 is called the local scheduling phase. At this phase, A DDG including all local instructions is constructed and then scheduled. The output of this phase is a rearranged local instruction list with scheduling annotations. Right now a top-down cycle scheduling is used for simplicity. However, this phase is modular, so in the future, it can be easily replaced with more effective scheduling approach. For example, we may use bottom-up cycle scheduling, which can take care of architectures with branch-delay slots. (A major difficulty expected for bottom-up scheduling is merging the machine state at the top of the current block)

Phase 2 is called the global scheduling phase. At this phase, the scheduler walks through the scheduled local instruction list from top to bottom again, trying to fill empty slots in the middle with global instructions. So we use the same heuristics as the trace-based scheduling: global code motion only in need. The critical computation in DAG-based scheduling is finding available global instructions.

The approach used for computing available global instructions is based on that of Moon and Ebcioglu [MoEb92]. Finding globally available instructions can be treated as a backward data-flow problem, which in fact simulates all the possible global code motions statically. While upward code motion is simulated, only right hand sides(rhs) of instructions are considered because the left hand side(lhs) of instructions can be renamed in order to eliminate anti-dependence and output dependence. Therefore we use *rhs_out[B]* denoting all the right hand sides that can be moved up to the bottom of block B, and *rhs_in[B]* denoting all the right hand sides that can be move up and through block B to its top.

Like all other backward data-flow problem, this problem is solved by applying two operations to every basic block repeatedly:

a) Merging: Compute *rhs_out[B]* from *rhs_in[x]*, x stands for successors of block B.

b) Raising: Compute *rhs_in[B]* from *rhs_out[B]*.

While raising rhs set from the bottom of block B to the top of block B, some new right hand sides are added, while other right hand sides may be removed because of data dependence. Register renaming and operation combining can be performed during this process to eliminate some code motion constraints.

While merging rhs sets at the top of successors of block B to the bottom of block B, rhs sets are unioned. The same right hand sides which may be raised from different paths are unified into one. However, the merging is not simply an union. First, the validation of the original left hand side needs to be checked e.g. When a rhs is raised from one path, if the original left hand side is live along other paths, then the lhs will no longer be valid (need renaming at code motion time). Second, the priority parameters associated with each rhs need to be updated, for example, the Degree of Speculation(DOS) [MoEb92].

Figure 6 illustrates the idea of DAG scheduling. First, conduct backward dataflow computation, calculating the initial values of rhs_in and rhs_out for each block in the DAG. Then, as discussed before, schedule each basic block locally, then try to fill the holes in the local instruction list with global instructions. For each empty slot, the global candidate is selected from the rhs_out of the current block according to priority parameters. For the chosen candidate, the scheduling action is described as follows:

1. Check the availability of the chosen rhs according to the DDG containing all the scheduled instructions (this is the DDG used in local scheduling, which is held and grown when global instructions are inserted); See if the data dependences have been met for this rhs at the given cycle.

2. Check the current machine state, the forward state above this position and the reverse state below this position. If this rhs cannot fit in this position without resource conflicts, this motion fails.

3. Once a good candidate is found, such as i in Figure 6(b). Trace back the moving paths, find out the original instructions from which this rhs is generated. Notice that one rhs can have multiple sources. Mark the paths from the current basic block to all those source instructions, that can be viewed as if all these sources are raised simultaneously along those paths. This generalizes the idea of jump operation shown in Figure 1, which in fact is just a special case of moving instructions simultaneously along multiple paths.

4. Try the transformation, make sure it will not abort in the middle. It has been tested that this rhs can be inserted into the empty slot. However, extra duplications may be needed for those basic blocks, such as block B5 in Figure 6, which are not on the moving paths, but join to the moving paths, so this step tests if all these duplications can be inserted into those basic blocks. If not, a simply solution is to give up this motion. The general solution is to truncate a subset of the moving paths (give up == truncate all), select the remaining subset as the feasible moving paths. However this general approach is complicated and deserves further investigation because the decision may need to be made based upon profile information.

5. Commit the transformation: put the rhs in the empty slot, insert duplications into off-trace blocks and remove those source instructions if they have the same destination as the newly generated instruction, or replace them with copy instructions. See figure 6(c)

6. Data flow update. Since some instructions are moved up, it may make more global instructions available. Notice that changes only affect those blocks on the moving paths. So we only need incremental data flow update along the moving paths.

This algorithm is very close to the algorithm described by Moon and Ebcioglu in [MoEb92]. The major differences are that we do global code motion only in need while they move all possible instructions. Also, we are proposing a more general transformation, moving instruction along multiple paths to eliminate redundant duplications, while they move up one instruction along single path.

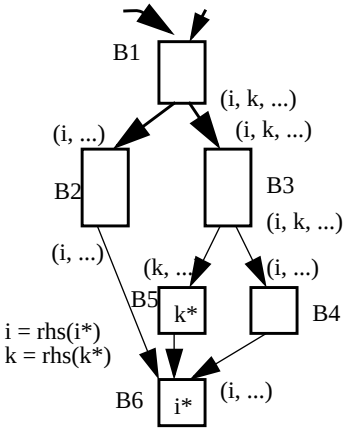

Figure 6(a) Compute rhs_in sets

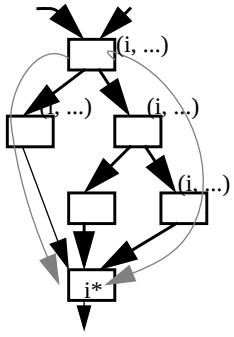

Figure 6(b) Trace back

a) Instruction *i** is originally in Block 6. Instruction *k** is originally in Block 5. i is rhs of i*, and k is rhs of k*. Assume, for some reason, i can not be moved across *B5*. Both *i* and *k* can be moved to the bottom of *B1* through other paths.

b) Pick up *i* as the candidate. Trace back, look at the *rhs_in* of every block. Find two paths to move *i* from *B6* to *B1*.

c) *i* is moved up from *B6* to *B1* along two paths. One duplication is left in *B5* because *B5* is out of the traveling paths.

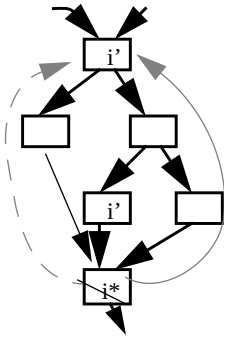

Figure 6(c) Move up

Figure 6. Several steps of DAG-based scheduling

6 Summary

Up to now, we have constructed a common scheduling framework, implemented a trace-based scheduler upon this framework, and are implementing the proposed DAG-based scheduler using the same framework.

After finishing the initial version of DAG-based scheduler, in the next step, we will continue enhancing the scheduling framework:

- Integrate the array dependence analysis, alias analysis and interprocedural analysis of High SUIF, and add more powerful data dependence analysis support.
- Incorporate profiling; add support for profile-driven scheduling.
- Extract a common local instruction scheduling module.

We will continue enhancing trace-based and DAG-based scheduling:

- Pick trace and DAG based upon profile information.
- Select the global available instruction and code motion path based upon profile information.

Furthermore, we will investigate the following problems:

- How to extend the DAG-based scheduling to loop scheduling so that we can provide an integrated instruction scheduling solution.
- How to integrate instruction scheduling with register allocation.

- Applying our instruction scheduling techniques on innovative architectures, studying the interaction between new architecture features and compilers.

7 Acknowledgments

The authors would particularly like to thank Vas Bala and Norm Rubin for their guidance and suggestions, and for providing the resource usage vectors of Alpha and HPPA architectures. Particularly thank Cliff Young for providing a lot of valuable comments and helping us refining the paper. Cliff also tested the trace-based scheduler on the PowerPC platform. The earlier version of our control-flow-graph library was provided by Bob Wilson at Stanford. It was modified and reorganized by Cliff and Tony DeWitt for use under Machine SUIF. Glenn Holloway implemented our register allocator and helped to implement our data-flow analysis routines. Other members of HUBE research group at Harvard contributed useful suggestions.

This research was sponsored in part by grants from AMD, Digital Equipment, Hewlett-Packard, IBM, and Intel. Michael D. Smith is supported by a National Science Foundation Young Investigator award, grant number CCR-9457779.

8 References

[BaRu95] V. Bala and N. Rubin. "Efficient Instruction Scheduling Using Finite State Automata," *Proc. MICRO-28*, pp. 46–56, Nov. 1995.

[Chan95] P. Chang, et al. "The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processor," *IEEE Trans. on Computers*, 44(3):353–370, Mar. 1995.

[Davi81] S. Davidson, et al. "Some Experiments in Local Microcode Compaction for Horizontal Machines," *IEEE Trans. on Computers*, 30(7):460–477, Jul. 1981.

[GeAp96] L. George and A. Appel. "Iterated Register Coalescing," *ACM Trans. on Prog. Lang. and Systems*, 18(3):300–324, May 1996.

[GeAp96] L. George and A. Appel. "Iterated Register Coalescing," *ACM Trans. on Prog. Lang. and Systems*, 18(3):300–324, May 1996.

[Fish81] J. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. on Computers*, 30(7):478–490, Jul. 1981.

[MoEb92] S. Moon and K. Ebcioglu. "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW processors," *Proc. MICRO-25*, pp. 55–71, Dec. 1992.

[Pres88] W. Press, et al. *Numerical Recipes in C*, Cambridge Press, 1988.

[Smit92] M. Smith. "Architectural Support for Compile-Time Speculation," *The Interaction of Compilation Technology and Computer Architecture*, edited by David Lilja and Peter Bird, Kluwer Academic Publishers, pages 13–49, 1994.

[Smit96] M. Smith. "Extending SUIF for Machine-dependent Optimizations," *Proceedings of the First SUIF Compiler Workshop*, Stanford, CA, pages 14–25, Jan. 1996.

[Wils94] R. Wilson, et al. "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," *ACM SIGPLAN Notices*, 29(12):31-37, Dec. 1994.