CHAPTER 6

Compiling for Superscalar Processors

6.1 INTRODUCTION

Superscalar processors fetch, decode, and execute more than one instruction per cycle with duplicated decode/issue units, functional units, and datapaths. In order for the full performance to be extracted from these processors, techniques must be used to minimize the hardware stalls caused by the sequentiality between instructions. As the pipelining depth and the instruction issue rate increases, these stalls become more costly. As a result, it has become increasingly important for the compiler for superscalar processors to minimize sequentiality among instructions.

Traditionally, optimizing compilers improve program execution speed by reducing unnecessary instruction execution [1]. Figure 6.1 shows an overview of such a traditional optimizing compiler for scalar processors. The compiler translates high level source code into an intermediate representation. The intermediate code is transformed by optimization algorithms of various levels of sophistication to improve the execution frequency. Local optimization algorithms optimize within code segments where execution neither enter nor exit the middle of the segment. Global optimizations optimize the entire function or subroutine body. Loop optimization exploits the fact that the loop body

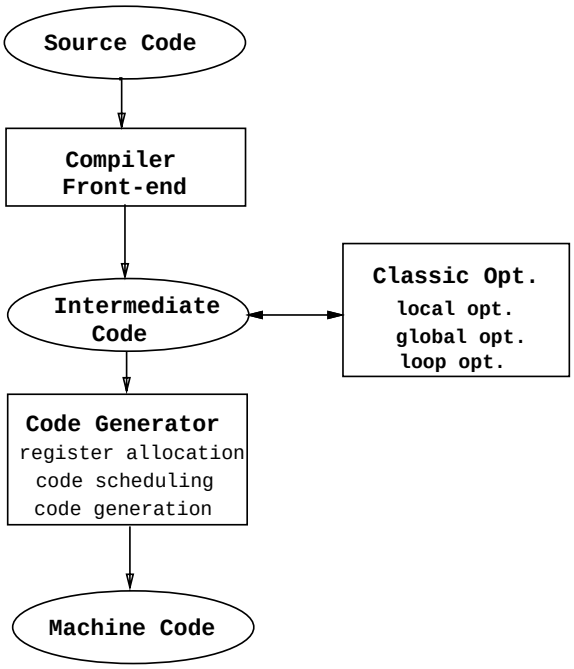

Figure 6.1 A traditional scalar compiler.

tend to execute multiple times whenever execution reaches it. All these optimization algorithms keep memory data and intermediate computation results in high speed processor registers. As a result, they reduce the program execution cycles spent waiting for the memory system and performing redundant computation. While this model of optimization will remain important to the performance of future microprocessors, the rapidly increasing hardware parallelism demands parallelization techniques that have been missing from most traditional compilers.

The code generation phase of traditional compilers translates intermediate code into machine code. The intermediate code usually assumes an infinite number of *virtual registers* to simplify the task of code optimization. During code generation, these virtual registers must be folded by a register allocation algorithm into the *physical registers* provided in the instruction set architecture of the processor. To better use the instruction pipeline of scalar processors, the code generator usually performs some limited instruction scheduling to fill load and branch delay slots. The code generation phase ends with writing actual machine code to the output file.

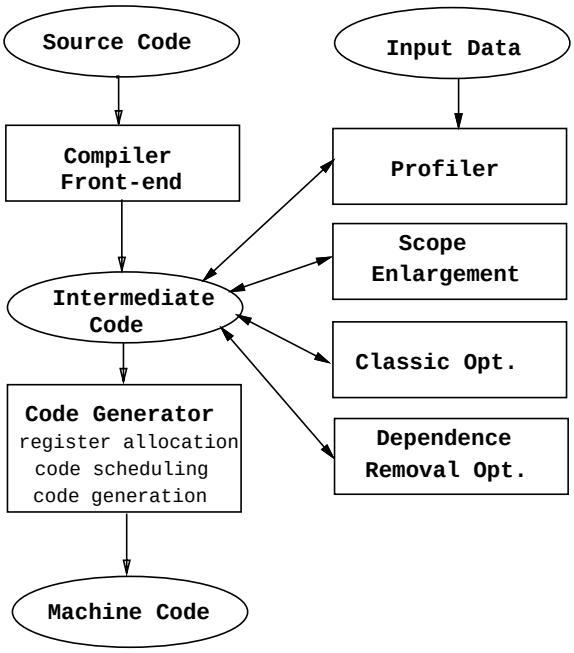

Figure 6.2 A generic superscalar compiler.

Compared to traditional scalar processors, superscalar processors impose additional responsibility on the compiler. The quality of compiler parallelization techniques can potentially make an order of magnitude difference in program execution performance for superscalar processors. With so much at stake, the industry is moving to incorporate parallelization techniques in their compilers. Figure 6.2 shows an overview of a compiler for a superscalar processor. The dependence removal optimizations transform the intermediate representation to enhance the parallelism among instructions. The scope enlargement algorithm, facilitated by the profiler, allows the optimization algorithms to efficiently handle code with frequent branches. The scope enlargement algorithm also implicitly enhances the effectiveness of code scheduling in the code generator. The objective of the chapter is to introduce the major concepts of dependence removal optimizations, scope enlargement, and code scheduling.

6.2 BASIC BLOCK COMPILATION

A basic block is a sequence of instructions where execution can only enter from a unique entrance point (top) and exit from a unique exit point (bottom). Whenever the execution reaches a basic block, it will traverse through

```
avg = 0;
weight = 0;
count = 0;
while(ptr != NIL) {
   count = count + 1;
   if(ptr->wt < 0)
      weight = weight - ptr->wt;
   else
      weight = weight + ptr->wt;
   ptr = ptr->next;
}
if(count != 0)
   avg = weight/count;
```

(a)

```
(i1)          load   r1,   _ptr
(i2)          mov    r7,   0          // avg
(i3)          mov    r2,   0          // count
(i4)          mov    r3,   0          // weight
(i5)          beq    L3,   r1,   0
(i6)    L0:   add    r2,   r2,   1
(i7)          load   r4,   0[r1]      // ptr->wt
(i8)          bge    L1,   r4,   0
(i9)          sub    r3,   r3,   r4
(i10)         br     L2
(i11)   L1:   add    r3,   r3,   r4
(i12)   L2:   load   r1,   4[r1]
(i13)         bne    L0,   r1,   0
(i14)   L3:   beq    L4,   r2,   0
(i15)         div    r7,   r3,   r2
(i16)         store  _avg  r7
(i17)   L4:
```

(b)

Figure 6.3 Code segment, (a) C source (b) intermediate code.

all the instructions before leaving at the bottom. Identifying basic blocks is an important functionality in the compilation process for both scalar and superscalar processors. Local optimizations operate within each basic block and thus require the knowledge of basic block boundaries. Global and loop optimizations are facilitated by the knowledge of control and data flow information among basic blocks. Code scheduling algorithms rely on basic block information to derive the constraints imposed by branches and other control transfer instructions. In this section, we will focus on local compiler optimization and scheduling algorithms that apply within basic blocks.

The C code segment shown in Figure 6.3(a) will be used in this chapter to illustrate the compilation algorithms. This code segment traverses a linked list and accumulates the absolute value of each element's weight into variable *weight*. After visiting all the elements, the code computes the average weight of elements in the list. During compilation, the compiler generates intermediate code for the source code. The intermediate code is shown in Figure 6.3(b). The intermediate code is based on a load-store processor model and its format is *opcode destination, source1, source2* where the number of source operands depends on the opcode.

6.2.1 Basic block construction

Figure 6.4 shows a generic algorithm for identifying basic blocks from a sequence of intermediate code instructions, referred to as operations in the algorithm. The basic idea is to find all the possible entry and exit points of the instruction sequence. An entry point is the first instruction of the sequence, the target of a control transfer instruction, or the instruction after a

```
// Find basic block entry points
for   ( oper = operation_list; oper != NULL; oper = oper->next ) {
      if ( FirstOper(oper) )
            SetEntry(oper);
      else if ( TargetOfBranch(oper) )
            SetEntry(oper);
      else if ( PrevOperIsBranch(oper) )
            SetEntry(oper);
}
// Add operations to basic blocks
for   ( oper = operation_list; oper != NULL; oper = oper->next ) {
      if ( IsEntry(oper) )
            bb = NewBB();
      AddOper(bb,oper);
}
```

Figure 6.4 A generic basic block recognition algorithm.

control transfer instruction. These entry and exit points define basic block boundaries. The output of the algorithm is a list of all the basic blocks thus identified. Instructions are inserted sequentially into the basic block to which they belong.

6.2.2 Dependence graph construction

Dependences represent constraints in ordering instructions. These constraints arise due to register accesses (register dependences), memory accesses (memory dependences), and control transfers (control dependences). They are expressed in a dependence graph. Each node in the dependence graph represents an instruction and each arc an ordering constraint between a pair of instructions. An arc from instruction A to instruction B specifies that A must be executed before B.

There are three types of register data dependences in general, as illustrated in Figure 6.5.

- Data flow dependency: the destination register of A is the same as one of the source registers of C, and C is subsequent to A. In this case, a subsequent instruction consumes the execution result of a previous instruction.

- Data anti-dependency: one of the source operands of C is the same as the

```
A   r2 ← (r1)[0]
B   r3 ← (r1)[1]
C   r2 ← r2 + r3
D   r3 ← (r1)[2]
```

Figure 6.5 A simple code example to illustrate register data dependences.

destination register of D, and D is subsequent to C. (C should receive result of B. However, if D is executed too soon, C may get execution result of D, which is too new.) In this case, a subsequent instruction overwrites one of the source registers of a previous instruction.

- Data output dependency: the destination register of B is the same as the destination register of D, and D is subsequent to B. In this case, a subsequent instruction overwrites the destination register of a previous instruction.

A generic register dependence graph construction algorithm is shown in Figure 6.6. Register dependence arcs are created in two passes. The first pass scans the instruction sequence forward to create the flow and output dependences. The second phase scans the instructions backwards to create the anti-dependence arcs. A *register definition table* is used to record for each register the latest instruction processed thus far that writes into the register as its destination operand.

During the forward pass, for each new instruction encountered, the algorithm looks up the register definition table to identify the latest instructions that write into its source operands. Register flow dependence arcs are drawn from these instructions to the current instruction. Similarly, an output dependence arc is added from the latest instruction that writes into the destination register of the current instruction to the current instruction

During the backward pass, the instructions are visited in the reverse order. For each new instruction encountered the algorithm checks the register definition table to identify the latest instructions to write into its source operands. Since the instructions are visited in the reverse order, these are actually the instructions to write into the source operands in the nearest future according to the original program order. Register anti-dependence arcs are created from these writes to the current instruction.

```
// Construct Register Dependence Graph for a basic block

ClearRegisterDefinitions();
// Draw forward register dependence edges
for   ( oper = FirstOp(bb); oper != NULL; oper = oper->next) {
      src1 = OperSrc1(oper);
      src2 = OperSrc2(oper);
      dest = OperDest(oper);

      // Register Flow Dependences
      if ( IsRegister(src1) && PrevDefined(src1) )
            AddFlowDepArc(DefiningOp(src1),oper);

      if ( IsRegister(src2) && PrevDefined(src2) )
            AddFlowDepArc(DefiningOp(src2),oper);

// Register Output Dependences
      if ( IsRegister(dest) && PrevDefined(dest) )
            AddOutputDepArc(DefiningOp(dest),oper);

DefineRegister(oper,dest);
}

ClearRegisterDefinitions();

// Draw backward register dependence edges
for   ( oper = LastOp(bb); oper != NULL; oper = oper->prev) {
      src1 = OperSrc1(oper);
      src2 = OperSrc2(oper);
      dest = OperDest(oper);

      // Register Anti-Dependences
      if ( IsRegister(src1) && PrevDefined(src1) )
            AddAntiDepArc(DefiningOp(src1),oper);

      if ( IsRegister(src2) && PrevDefined(src2) )
            AddAntiDepArc(DefiningOp(src2),oper);

      DefineRegister(oper,dest);
}
```

Figure 6.6 A generic basic block register dependence graph construction algorithm.

Memory dependence arcs represent instruction ordering constraints between loads and stores. For each pair of memory access instructions, if at least one of the instructions is a store, and the compiler cannot tell that the two instructions access non-overlapping locations, a memory dependence arc is added from the preceding instruction to the succeeding instruction.

Control dependence arcs express instruction ordering constraints due to control transfers. The only relevant control dependence within a basic block is that the control transfer instruction that ends a basic block must remain as the last instruction of the basic block after code transformation. This is accomplished by creating a control dependence arc from each instruction in the basic block to the control transfer instruction. As a result, the control transfer instruction will never be moved ahead of any other instruction in the basic block.

6.2.3 Machine description and dependence latency

Each dependence arc in the dependence graph has a latency that specifies the desired delay between two dependent instructions. For example, in a processor where a load instruction requires two clock cycles to execute, a register flow dependence arc between a load and another instruction will have a latency of two cycles. This indicates that the dependent instruction should be schedule two cycles after the load instruction in order to avoid interlocking.

In general, latency varies across processors. The compiler writer needs to have a flexible mechanism to specify such latencies for difference processors or different generations of the same processor family. One could attempt to specify dependence distances by specifying a single latency for each operation, such as two cycles for loads. Unfortunately, this is not sufficient when the dependence distance also depends on when the result is being used by the consuming operation. For example, address generation sometimes is done a cycle early in the processor pipeline, requiring that the address operands be available a cycle early. Additionally, for store operations, the data being stored is sometimes read a cycle late in the processor pipeline, allowing the store to be issued one cycle before this data is ready. To take this into account while scheduling, the register flow dependence distance to address operands should be increased by one cycle and for stores, the register flow dependence distance to the store's data operand should be decreased by one cycle. These types of latency variations can be modeled by describing when source operands are read and destination operands are written for each operation, where time zero is typically defined as just before an operation begins execution.

An example of specifying dependence distances through the use of operand

```
// Shift operations have a two cycle latency (dest writes in time 2)
OUT_shft(dest(2) src(0 0));

// Stores use address operands early (time -1) and the data operand late (time 1)
OUT_st   (       src(-1 -1 1));
```

Figure 6.7 An example specification of Operand Use Time (OUT).

use (read/write) times is shown in Figure 6.7. Typically, operations read their source operands at time zero (right before execution) and write their destination operands at their latency. Thus, a two cycle shift operation would be described by the entry OUT_shft shown in the figure, which writes its destination at time two and reads all of its source operands at time zero. The atypical store operation described above would then have an entry similar to OUT_st which reads its address operands at time negative one and the operand to be stored at time one. Note that these operand read and write times take into account all the forwarding and bypassing logic in the processor and may not correspond to when a pipelined processor would actually read from or write to the register file.

Using this information, the latency for a register flow dependence can be calculated using the algorithm shown in Figure 6.8. The algorithm first queries the above information for the relative time at which the destination is written and the relative read time of the register's use. The latency is then calculated by subtracting the read time from the write time. If the result is negative, the latency is set to zero (the use must follow the define). Thus if calculating the dependence latency from a two cycle shift to the first (address) source of a store, the latency would be 2 - (-1) = 3. From a two cycle shift to the last (data) sources of a store, then latency would be 2 - 1 = 1. This method of specifying the operand use times for operations handles almost all of the latency characteristics in today's processors. However, some forms of partial bypassing cannot be fully modeled using this method. The enhancements required to handle partial bypassing are beyond the scope of this chapter.

6.2.4 Code Scheduling

Code scheduling algorithms use dependence graph and dependence latencies to determine an instruction execution order that results in good performance.

```
// Calculate latency of register flow dependence
dest_write_time = WriteTime(dep->from_oper, dep->from_which_dest);
src_read_time = Read_Time(dep->to_oper, dep->to_which_src);
dep->latency = dest_write_time - src_read_time;
if (dep->latency < 0)dep->latency = 0;
```

Figure 6.8 Calculation of register flow dependence latency using operand use times.

For example, to avoid stalls due to an instruction with a long latency (such as a load or a multiply), the scheduler will attempt to move it upward in the code so that its result is ready in time for use by a subsequent instruction. While reorganizing the code, it must preserve the correctness of the original program with respect to the data and control dependences. In this work, it is assumed that instruction latencies and the type and number of functional units are visible to the code scheduler.

List scheduling is a simple and popular form of scheduling algorithm in industry. The general idea of list scheduling is to pick, from a set of dependence graph nodes (instructions) that are *ready* to be scheduled, the best combination of nodes to issue in a cycle. The best combination of nodes is determined by using heuristics which assign priorities to the ready nodes [7]. A node is ready if all of its parents in the dependence graph have been scheduled and the result produced by each parent is available.

Figure 6.9 shows a generic list scheduling algorithm for basic blocks. The scheduler first builds a dependence graph. It then assigns priorities to instructions according to the scarcity of their required resources and the number of their dependent instructions. Once the dependence graph is in place, the scheduler maintains a pool of ready instructions whose dependence latencies have been satisfied. At the beginning, the pool consists of instructions that do not depend on any other instructions in the basic block.

The scheduler starts with cycle 0. It selects instructions from the ready pool according to the priorities assigned to them (GetHighestPriorityReady-Oper). For each instruction thus selected, the scheduler checks the resource requirement of the instruction against the resource available. If the resource requirement can be satisfied, the instruction is scheduled for execution in the current cycle (ScheduleOper). All the resources required for the scheduled op-

```
// Build the dependence graph the basic block being scheduled
BuildDepGraph(bb);

// Calculate static scheduling priorities for each operation
CalcSchedPriorities(bb);

// Start scheduling in cycle 0
cycle = 0;

// Schedule all the operations in the basic block
while (AreUnscheduledReadyOpers(bb)) {
    // In priority order, try to schedule the operations ready in this cycle
    while ((oper = GetHighestPriorityReadyOper(bb,cycle)) != NULL) {
        if (ResourcesAvailable(bb,oper,cycle))
            ScheduleOper(bb,oper,cycle);
        else
            CannotSchedule(bb,oper,cycle);
    }
    cycle++;
}
```

Figure 6.9 A generic list scheduling algorithm.

eration are reserved. Scheduling an instruction can in turn enable more ready operations. If the resource requirement cannot be satisfied, the instruction is rejected for the current cycle and held for future attempts in following cycles (CannotSchedule). Once the scheduler has exhausted all the potential opportunities for the current cycle, it moves on to schedule for the next cycle. The process iterates until there are no more instructions left unscheduled.

6.2.5 Machine description and instruction resource requirements

A very important aspect of code scheduling is to detect if there are enough resources available to satisfy the requirements of a candidate instruction, shown as the ResourcesAvailable function in Figure 6.9. Once a candidate instruction is selected, the scheduler has to record all the resources consumed by the instruction so that the selection of subsequent instructions will be based on up to date resource availability information, which would be done in the ScheduleOper function shown. In order to model the processor accurately, these functions require detailed knowledge of the resources available in the processor

and knowledge of how operations use these resources as they execute. This information is typically specified in a machine description as a list of processor resources and a set of reservation tables describing how operations use these resources over time. For example, a scalar processor might be described as shown in Figure 6.10. In this figure, a list of the processor resources and the reservation tables for a two-cycle shift operation (RT_Shft) and a store operation (RT_St) are shown.

Each reservation table is specified as a list of resource usages, where each usage specifies a processor resource and the time that the resource is used relative to when the operation begins execution. A time of zero is usually defined to be just before an operation begins execution (since scheduling usually focuses on the execution stage). So the reservation table RT_Shft for a two cycle shift operation specifies that the decoder is used at time -2, both register read ports (Rp1 and Rp2) are used at time -1, the Shift resource is used for two cycles (time 0 and 1), and the register write port (Wp1) is used at time 1. The reservation table RT_St is for a store operation that reads its address operands a cycle early (time -2) and then reads the data to be stored a cycle late (time 0). The resources that need to be modeled depends heavily on the the resource constraints of the processor. If there are plenty of register ports, it may not be necessary to model them. If result buses are a bottleneck, then they need to be modeled. For the two reservation tables shown, the shift unit and the read register ports are the source of resource conflicts. The two cycle use of the shift resource prevents a shift from being initiated every cycle. The unusual read port usage by stores prevents shifts from initiating the cycle before or after a store and prevents a store from initiating two cycles after another store. The scheduler's job is to rearrange operations in a code segment to minimize the amount of conflict that occurs due to these resource constraints.

The information in these reservation tables can be then used by a resource management algorithm, shown in Figure 6.11, to answer the scheduler's questions about whether or not there are enough resources to schedule an operation. The ResoucesAvailable function checks each resource usage (ru) in the operation's reservation table to see if every resource is available at the time required, relative to when the operation will execute (cycle). If all the resources are available at the required times, the function returns TRUE to the scheduler, otherwise it returns FALSE. If the resources are available to schedule the operation, the list scheduler algorithm described earlier then calls ScheduleOper to schedule the operation in that cycle. Inside the call to ScheduleOper routine (not shown), the UseResources function is called so that the resource availability information is kept up to date. The UseResources function

```
Resources {
    Decoder Rp1 Rp2 Wp1 Agen Shift Mem
}

Reservation_Tables {
    // Define time 0 as just before an operation begins execution
    RT_Shft  ((Decoder -2)(Rp1 -1)(Rp2 -1)(Shift 0)(Shift 1)(Wp1 1));
    RT_St    ((Decoder -2)(Rp1 -2)(Rp2 -2)(Agen -1)(Rp1 0)(Mem 0));
}
```

Figure 6.10 Example specification of a processor's resources and how operation use them as they execute.

goes through each resource usage in the operation's reservation table again, this time marking each resource as being used. These functions assume that behind the scenes there is a resource map table that is used by the ResourceAvailable and UseResource functions to keep track of when processor resources have been used.

For simplicity, the resource manager algorithms presented assume only one reservation table per operation, but more flexibility is usually needed when describing superscalar processors. There are usually multiple function units, decoders, etc. that can be used as the operation executes and all of these scheduling alternatives need to modeled. A common solution is to specify a reservation table for each scheduling alternative, and during scheduling try each of these alternatives until one is found where all the required resources are available. This approach works well as long as the number of alternatives remains small. There are representation enhancements that can be used to reduce the number of alternatives, such as counter or graph-based approaches, but these enhancements are beyond the scope of this chapter.

6.2.6 Relation to register allocation

Code scheduling can be done before, after, or in conjunction with register allocation. When performed before register allocation, it is referred to as prepass code scheduling or prescheduling. Since no register allocation has been done to the code, there is an infinite number of virtual registers and little register reuse has been imposed. Therefore, the scheduler tends to have the most freedom to reorder instructions [8]. The issue is, however, to control the register usage during scheduling to avoid excessive register spilling during

```
// Returns 1 if resources available for the operation
ResourcesAvailable(resource_map,oper,cycle)
{
    // Check each resource/time pair to see if all resources are available
    for (ru = oper->ru_list; ru != NULL; ru = ru->next) {
        if (!ResourceAvailable(resource_map,ru->resource,cycle+ru->time))
            return(FALSE);
    }
    return (TRUE);
}

// Mark the resources used by an scheduled operation
UseResources(resource_map,oper,cycle)
{
    // For each resource/time pair, mark the resource used at that time
    for (ru = oper->ru_list; ru != NULL; ru = ru->next)
        UseResource(resource_map,ru->resource,cycle+ru->time);
}
```

Figure 6.11 A generic resource manager algorithm

register allocation.

When code scheduling is performed after register allocation, it is referred to as postpass code scheduling or postscheduling. The register allocator introduces extra dependences whenever it reuses registers. These extra dependences restrict the ability of the code scheduler to move instructions to their desired positions. Therefore, postschedulers tend to have limited effectiveness. The advantage is that they do not cause additional register spills.

There has also been work done in the area of combined register allocation and code scheduling [14] [21]. In this case, the compiler typically switches between scheduling strategies according to the register usage. When register shortage occurs, the scheduler would switch to a strategy to minimize register usage rather than to maximize parallel execution.

6.2.7 Dependence removal optimizations

Code scheduling operates under dependence constraints. In many situations, the dependence constraints in the input code may be too restrictive for the scheduler to achieve good schedule. Dependence removal optimizations can be applied in these situations to reduce the constraints to facilitate code schedul-

ing. Flow dependences can often be removed by using techniques that rewrite how expressions are evaluated. One of these techniques, operation combining [26], can be used to eliminate flow dependences between pairs of instructions if each has a compile-time constant source operand. Opportunities for using operation combining often arise between address calculation instructions and memory access instructions along with loop variable increments and loop exit branches. To illustrate the application of operation combining to calculation of memory addresses and branch conditions, consider the following instructions sequence:

```
I1: r1 = r2 + 4
I2: r3 = MEM(r1+8)          ⇒   r3 = MEM(r2+12)
I3: beq r1, 100, exit       ⇒   beq r2, 96, exit
```

The goal of the operation combining algorithm is to rewrite $I2$ and $I3$ so that they use $r2$ instead of $r1$ with their constants adjusted appropriately. This removes the flow dependence from $I1$ to $I2$ and $I3$, allowing them to now all to be scheduled in the same cycle. The algorithm for performing operation combining on expressions involving addition and subtraction[1] is shown in Figure 6.12.

The algorithm scans the instruction sequence looking for an addition, subtraction, memory access, or conditional branch operation that has a constant source operand (to simplify matters, all constant operands are assumed to be placed in src2). If such an operation is found, the algorithm then determines if the operation's register source operand (assumed to be in src1) is defined within the basic block and that the defining operation is an addition or subtraction operation that also has a constant source operand. If all these criteria are satisfied, then this operation is a candidate for operation combining. The algorithm then checks the validity of the transformation by determining if the register operand of the defining operation is still available for use (has not been redefined). If the register is not available, the algorithm then tries to make the value available by reordering operations or performing some other transformation to make the register available for use. If the register operand is available, or can be made available, the operation combining transformation is performed.

The first step in performing the operation combining transformation is to calculate the value to add to adjust the operation's constant operand, taking

[1] Multiple and divide expressions can be transformed with a similar algorithm.

```
// Perform operation combining on applicable operation pairs in BB
ClearRegisterDefinitions();
for (oper = FirstOp(bb); oper != NULL; oper = oper->next)
{
    // To apply, oper must be an add/sub/mem/cbr with const src2
    // and src1 must be defined by an add/sub oper with a const src2
    def_op = DefiningOp(OperSrc1(oper));
    if ( (IsAdd(oper) || IsSub(oper) || IsMem(oper) || IsCbr(oper)) &&
        IsConst(OperSrc2(oper)) && (def_op != NULL) &&
        (IsAdd(def_op) || IsSub(def_op)) && IsConst(OperSrc2(def_op)))
    {
        // To apply, defining op's src1 value must be available to oper
        if ( ValueAvailable(bb, def_op, OperSrc1(def_op), oper) ||
            MakeAvailable(bb, def_op, OperSrc1(def_op), oper))
        {
            // Determine value and the proper sign for const adjustment
            adjustment = OperSrc2(def_op);
            if (IsSub(def_op))
                adjustment = -adjustment;
            if (IsSub(oper) || IsCbr(oper))
                adjustment = -adjustment;

            // Replace oper's src1 with def_op's src1 and adjust const in src2
            SetOperSrc1(oper, OperSrc1(def_op));
            SetOperSrc2(oper, OperSrc2(oper) + adjustment);
        }
    }
    DefineRegister(oper,dest);
}
```

Figure 6.12 An algorithm for operation combining.

into account any implicit negation of the constant operand in either of the operations. Initially, the adjustment is set to the defining operation's constant. If the defining operation is a subtraction, the adjustment is negated (the constant is assumed to be the value being subtracted). If the operation being transformed is a subtraction or a conditional branch, the adjustment is also negated. Then the actual transformation is performed by replacing the operation's register operand (src1) with the defining operation's register operand and adding the adjustment to the operation's constant operand. Now the transformed operation is no longer dependent on the defining operation

and can potentially be executed earlier.

Operation combining and other transformations for rewriting how expressions are evaluated are often called height reduction transformations because they aim to reduce the dependence height for a code segment. Although there is not enough space to go into their details here, it should be noted that, in general, height reduction transformations are not restricted to operating only on expressions with constant operands or expressions involving just two operations. In addition, sometimes these transformation require the addition of new operations into the basic block which may hurt performance if the additional resources are not available. In these cases, being able to apply a transformation does not necessarily imply that the transformation should be performed.

All of the code scheduling and optimization techniques introduced in this chapter so far are designed to operate within basic blocks. Although these techniques must deal with a substantial amount of complexity to properly address machine constraints and achieve speedup, they do not need to deal with the complexities involved in scheduling and optimizing across multiple basic blocks. The rest of this chapter will be dedicated to extending code scheduling and optimizations so that they can be applied across basic blocks.

6.3 BEYOND BASIC BLOCKS - SUPERBLOCKS

The need for code scheduling and optimization across basic blocks is real. There are many programs where only a small number (3-5 [23]) of instructions typically exist within basic blocks. Furthermore, these instructions in the same basic block tend to have dependences among them. In order to derive compact instruction schedules, the compiler must be able to remove dependences and reorder instructions across basic block boundaries. A popular structure to allow the compiler to go across basic block boundaries is the control flow graph.

6.3.1 Control flow graph

A control flow graph is a graph where nodes represent basic blocks and arcs control transfers between basic blocks. In a typical compiler where functions are compiled separately from each other, each function has its own control flow graph. In compilers that perform interprocedural analysis, two approaches exit. One is to connect all functions with a call graph while keeping the control flow graph on a per function basis. The other is to construct a single control flow graph for the entire program. The techniqiues discussed in this chapter are independent of the choice between these approaches.

Figure 6.13 shows a generic algorithm for constructing control flow graph from a list of basic blocks generated by the basic block recognition algorithm

```
// Construct Control Flow Graph
for   ( bb = bb_list; bb != NULL; bb = bb->next ) {
      last_op = LastOp(bb);
      if ( IsCondBranch(last_op) ) {
            AddArc(bb,BranchTarget(last_op));
            AddArc(bb,BranchFallThru(last_op));
      }
      else if ( IsUnCondBranch(last_op) )
            AddArc(bb,BranchTarget(last_op);
      else
            AddArc(bb,NextBlock(bb));
}
```

Figure 6.13 A generic control flow graph construction algorithm.

presented in Figure 6.4. The algorithm adds an arc between two basic blocks A and B if there is a possible control transfer from A to B. These transfers can be explicit: when there is a conditional or unconditional branch from A to B. The transition can also be implicit: when B is the fall-through alternative of the conditional branch of A when the condition is not met. The control flow graph of the assembly code segment in Figure 6.3 is shown in Figure 6.14.

In order to perform aggressive optimization and code reordering across basic blocks, the compiler often relies on frequency information to give preference to some control transfers than others. For example, in Figure 6.14, BB2 can either go to BB3 or BB4. The frequency information indicates that out of 100 instances of BB2 execution, the execution transfers to BB3 10 times and BB4 90 times. Such frequency information can be derived with execution profiling or estimates based on program analysis. When execution profiling is used, the compiler compiles the program twice. During the first or preliminary compilation, the compiler inserts extra instructions into the program to record the frequency of each branch taking one direction versas the other(s). The user runs the program on sample input files to collect frequency information. The information is then used by the compiler during the second or final compilation where the frequency information is used during code scheduling and optimization.

Techniques have also been developed to estimate frequency information using program analysis. Hank presented a set of heuristics [16] to derive frequencies by taking code characteristics into account. For example, assume that

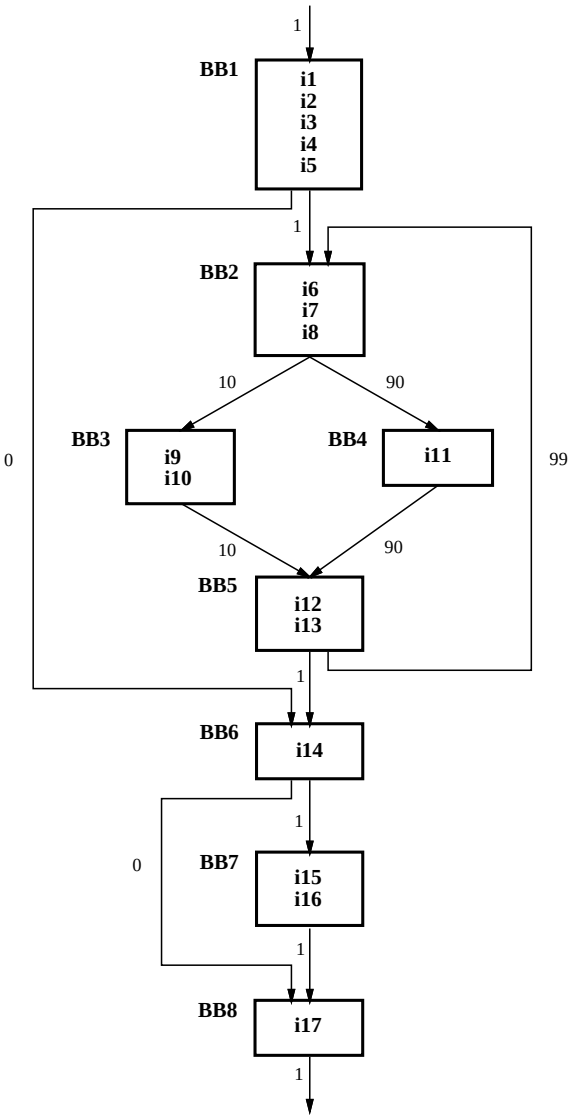

Figure 6.14 Weighted control flow graph.

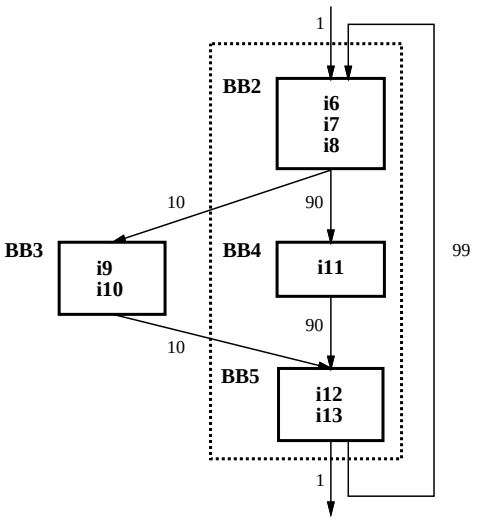

Figure 6.15 Loop portion of control flow graph after trace selection.

a basic block A can go to two basic blocks B and C. If B contains a call to buffer filling I/O functions, which tend to occur infrequently, the compiler can give a much higher estimated frequency to the transfer from A to C than from A to B.

A large number of frameworks and techniques have been proposed in the literature to facilitate scheduling across basic blocks. It is impossible to address all the techniques in one book chapter. The objective of this chapter is to introduce the important issues involved. We will use a framework based on the superblock structure, a compiler internal data structure widely used in industry, to illustrate how these issues can be addressed.

6.3.2 The superblock structure

The superblock is an extension to trace [11] which reduces some of the bookkeeping complexity. The first step of the superblock scheduling algorithm is to use trace selection to form traces from the most frequently executed paths of the program [11]. Figure 6.15 shows the portion of the control flow graph corresponding to the `while` loop after trace selection. The dashed box outlines the most frequently executed path of the loop.

In addition to a top entry and a bottom exit point, traces can have multiple side entry and exit points. A side entry point is a branch into the middle of a trace and a side exit is a branch out of the middle of a trace. For example,

the arc from **BB2** to **BB3** in Figure 6.15 is a side exit and the arc from **BB3** to **BB5** is a side entrance.

To move code across a side entrance, complex bookkeeping is required to ensure correct program execution [11][18]. For example, to schedule the code within the trace efficiently, it may be desirable to move instruction **i12** from **BB5** to **BB4**. To ensure correct execution when control flows through **BB3**, **i12** must also be copied into **BB3** and the branch instruction **i10** must be modified to point to instruction **i13**. If there was another path out of **BB3** then a new basic block would need to be created between **BB3** and **BB5** to hold instruction **i12** and a branch to **BB5**. In this case, the branch instruction **i10** would branch to the new basic block.

The second step of the superblock scheduling algorithm is to form superblocks. Superblocks avoid the complex repairs associated with moving code across side entrances by removing all side entrances from a trace. Side entrances to a trace can be removed using a technique called *tail duplication* [9]. A copy of the tail portion of the trace from the side entrance to the end of the trace is appended to the end of the function. All side entrances into the trace are then moved to the corresponding duplicate basic blocks. The remaining trace with only a single entrance is a superblock. Figure 6.16 shows the loop portion of the control flow graph after superblock formation and branch expansion.[2] During tail duplication, BB5 is copied to form superblock 2, (**SB2**). Since **BB3** only branches to **BB5**, the branch instruction **i10** can be removed and the two basic blocks merged to form **BB3'**. Note that superblock 1, **SB1**, no longer has a side entrance.

Loop-based transformations such as loop peeling and loop unrolling [4] can be used to enlarge superblock loops, a superblock which ends with a control flow arc to itself. For superblock loops that usually iterate only a small number of times, a few iterations can be peeled off and combined with the outer loop code to form larger superblocks. A loop is generated on the side to catch situations where the loop iterates more than the peeled iterations. For most cases, the peeled iterations will suffice and the body of the loop will not need to be executed. For superblock loops that iterate a large number of times, the superblock loop is unrolled several times. These transformations rely on frequency information to make their

After superblock formation many classic code optimizations are performed to take advantage of the profile information encoded in the superblock structure and to clean up the code after the above transformations. These opti-

[2] Note that the profile information is scaled during tail duplication. This reduces the accuracy of the profile information.

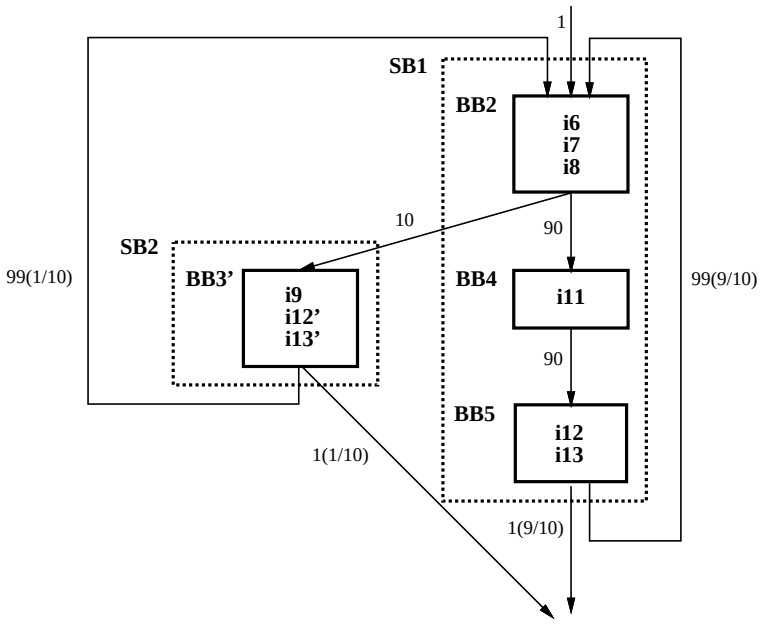

Figure 6.16 Loop portion of control flow graph after superblock formation and branch expansion.

mizations include the local and global versions of: constant propagation, copy propagation, common subexpression elimination, redundant load and store elimination, dead code removal, branch expansion and constant folding [1][9]. Local strength reduction, local constant combining and global loop invariant code removal, loop induction strength reduction, and loop induction elimination are also performed. To improve the amount of parallelism in superblocks, register renaming, loop induction variable expansion, accumulator expansion, and tree height reduction are applied to each superblock [18].

Dependence The third step in the superblock scheduling algorithm is to build a dependence graph. The dependence graph represents the data and control dependences between instructions. Data dependences have been introduced in Section 6.2.2. Data dependence arcs are formed in superblocks using the same algorithm as in basic blocks, as shown in Figure 6.6.

Control dependences represent the ordering constraints between a branch instruction and the instructions above and below the branch. In the case of basic blocks, branch instructions can only occur at the end. Therefore, in Section 6.2.2, control dependence arcs are added from all instructions of a

```
// Determine def/use sets for each basic block
for   ( bb = bb_list; bb != NULL; bb = bb->next ) {
      def = ∅;
      use = ∅;
      for   ( oper = FirstOp(bb); oper != NULL; oper = oper->next ) {
            if ( !SetIn(def,OperSrc1(oper)) )
                  SetAdd(use,OperSrc1(oper));
            if ( !SetIn(def,OperSrc2(oper)) )
                  SetAdd(use,OperSrc2(oper));
            if ( !SetIn(use,OperDest(oper)) )
                  SetAdd(def,OperDest(oper));
      }
      DefineDefSet(bb,def);
      DefineUseSet(bb,use);
}
```

Figure 6.17 Computing def/use sets for each basic block

basic block to the ending branch. However, since superblocks are formed from several basic blocks, they may contain multiple branches. One should allow instructions to move across branches unless such movement can cause incorrect execution.

If an instruction is moved below a branch instruction and the destination variable of that instruction is used before redefined along the taken path of the branch, incorrect execution will result. The converse is also true. Moving an instruction above a branch whose destination variable is used before redefined along the taken path of the branch will result in incorrect execution when the branch is taken. In order to prevent this, we define for each conditional branch instruction **I**, *live_out*(**I**) as the set of variables that may be used before they are defined when **I** is taken. A control-dependence arc is added from an instruction to a subsequent conditional branch **I** if the instruction writes a variable that is in *live_out*(**I**) when branch **I** is taken. A control-dependence arc is added from a conditional branch **I** to an instruction below it in the superblock if the destination variable of the instruction is in *live_out*(**I**) or if the instruction may cause an exception.

The purpose of live-variable analysis is to determine for a variable v and a point p whether the value of v at p could be used sometime later in the control

```
// Determine in/out sets for each basic block
for ( bb = bb_list; bb != NULL; bb = bb->next )
    DefineLiveInSet(bb,∅);

    change = 1;
    while ( change ) {
    change = 0;
    for ( bb = bb_list; bb != NULL; bb = bb->next ) {
        out = ∅;
        for ( s = FirstSuccessor(bb); s != NULL; s = NextSuccessor(bb) )
            out = out ∪ InSet(s);
        tmp = SetSubtract(out,DefSet(bb));
        in = SetUnion(UseSet(bb),tmp);

        // Determine if bb in set has changed
        tmp = SetSubtract(in,InSet(bb));
        if ( !SetEmpty(tmp) ) {
            change = 1;
            DefineLiveInSet(bb,in);
            DefineLiveOutSet(bb,out);
        }
    }
}
```

Figure 6.18 Computing live_in/live_out sets for each basic block

flow graph from point p. If so, then v is live at p; otherwise v is dead. For example, in Figure 6.19 register r3 is live at the entry to basic block BB1 since it is later used in both basic blocks BB2 and BB3.

Define live_in(B) to be the set of variables live at the entry point of basic block B. Define live_out(B) to be the set of variables live at the exit point of basic block B. Furthermore, let def(B) be the set of variables defined in B prior to being used and let use(B) be the set of variables used in B prior to being defined. Figure 6.17 contains the algorithm for computing the def and use sets for a basic block. The algorithm examines each operation of the basic block and if a source operand of an operation has not been defined by a previous

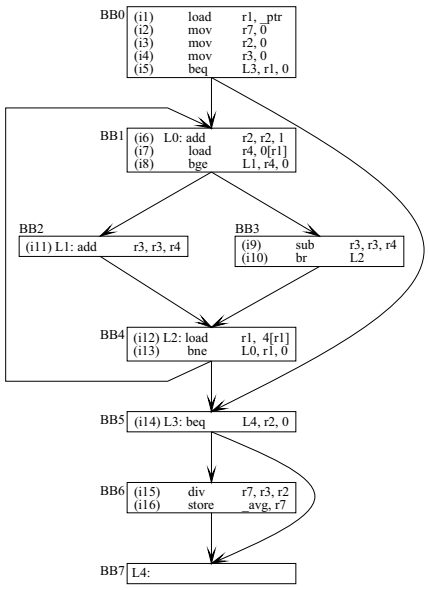

Figure 6.19 New control flow graph for first example

operation that operand is added to the use set of the basic block. The same is true for definitions. If the destination operand has not yet been used by a previous operation it is added to the def set of the basic block. As an example, consider BB1 in Figure 6.19. The set use(BB1) = r1, r2 , both of which are used before defined. Even though r4 is referenced by operation (i8), it is not placed in the use set because it is defined by the previous operation (i7). The set def(BB1) = r4 since it is the only register in BB1 that is defined before being used. Figure 6.20 shows the use and def sets for each basic block in Figure 6.19.

Once the def and use sets for a block are calculated, the live_in() and live_out() sets for each basic block can be computed using the iterative algorithm in Figure 6.18. The algorithm begins by initializing the live_in set every basic block to $\emptyset$. Then for each basic block, the live_in set is determined using the equation:

live_in(B) = use(B) $\cup$ (live_out(B) - def(B)), where

live_out(B) = $\cup$ live_in(S) for all successors of B

In order to determine the live_in/live_out sets for a function these two equations must be solved for each basic block. If there are n basic blocks in the function then $2n$ equations must be solved. Also, the new live_out set of a basic block results in new live_in sets for its successors. Therefore, the

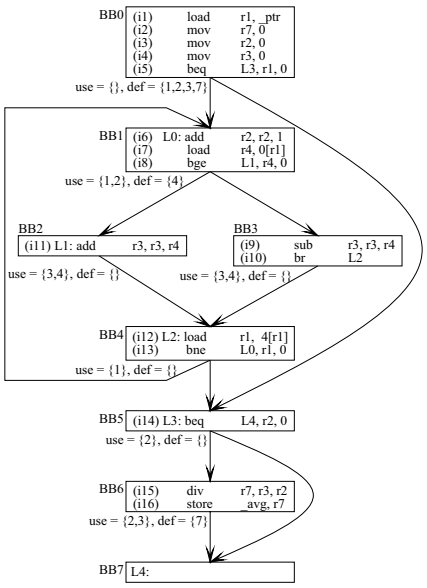

Figure 6.20 Example of def and use sets.

algorithm iterates until there is no more change in the live_in sets of all basic blocks. In the worst case, the algorithm in Figure 6.18 requires n iterations to converge to solution. For the control flow graph in Figure 6.20, this algorithm requires only two iterations. Figure 6.21 contains the live_in and live_out sets for each basic block.

Superblock scheduling Once the control and data dependence arcs are in place, the basic block code scheduling algorithm in Figure 6.9 applies directly to superblock. Instructions without control dependences move freely across branch instructions during code scheduling. This is one of the advantages of the superblock structure. Many local optimization and scheduling algorithms are directly applicable to superblocks. In order to achieve high performance, however, the scheduling heuristics must be designed to avoid excessive delays to branches when moving instructions above branches [6].

6.4 ADVANCED TOPICS

This chapter is designed to cover the very basics of compiling for superscalar processors. A state-of-the-art compiler for superscalar processors today typically employs more advanced techniques. This section gives a incomplete set of further references for readers who want to master the art of superscalar

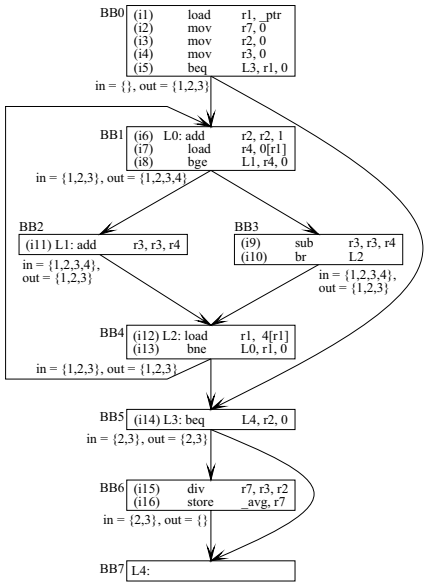

Figure 6.21 Example of live_in and live_out sets.

compilation.

In the area of program analysis, memory disambiguation techniques exist to facilitate aggressive reordering of memory loads and stores [21] [12]. Safe analysis techniques allow compilers to detect instructions that do not cause additional exceptions when moved above branches and therefore eliminate unnecessary control dependence arcs [6].

In the area of scheduling technqiues, there exist a large number of techniques that take advantage of loop structures to perform better code scheduling [28] [25] [30]. Also, there exist numerous techniques to perform code scheduling on a wide variety of code structures [2] [5] [15].

In the area of dependence removal transformations, techniques have been developed to perform register renaming, induction variable expansion, accumulator/search variable expansion, and height reduction [19] [31] [32] [20].

In the area of using special architectural features, compiler techniques have been designed to use control speculation [10] [22], data speculation [13], predication [3] [27] [24] [17].

Last but not least, interested readers are referred to an excellent article by Rau and Fisher on the history, overview, and the perspective of instruction-level parallel processing [29] where a comprehensive reading list is also included.

6.5 REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14:584–594, May 1988.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[4] K. Anantha and F. Long. Code compaction for parallel architectures. *Software Practice and Experience*, 20:537–554, June 1990.

[5] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[6] R. A. Bringmann. *Compiler-Controlled Speculation*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[7] P. P. Chang, D. M. Lavery, and W. W. Hwu. The importance of prepass code scheduling for superscalar and superpipelined processors. Technical Report CRHC-91-18, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1991.

[8] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. The importance of prepass code scheduling for superscalar and superpipelined processors. *IEEE Transactions on Computers*, 44(3):353–370, March 1995.

[9] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, December 1991.

[10] P. P. Chang, N.J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Transactions on Computers*, 44(4):481–494, April 1995.

[11] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30:478–490, July 1981.

[12] D. M. Gallagher. *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[13] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 183–193, October 1994.

[14] J. R. Goodman and W. C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 442–452, July 1988.

[15] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16:421–431, April 1990.

[16] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu. Superblock formation using static program analysis. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.

[17] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler technology for future microprocessors. *To appear IEEE Proceedings*, December 1995.

[18] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[19] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective structure for VLIW and superscalar compilation. Technical report, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.

[20] D. M. Lavery and W. W. Hwu. Unrolling-based optimizations for modulo scheduling. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 327–337, November 1995.

[21] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace scheduling compiler. *The Journal of Supercomputing*, 7(1):51–142, January 1993.

[22] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for superscalar and VLIW processors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247, October 1992.

[23] S. A. Mahlke, R. E. Hank, J.E. McCormick, D. I. August, and W. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 138–150, June 1995.

[24] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[25] S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and vliw processors. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 55–71, December 1992.

[26] T. Nakatani and K. Ebcioglu. Combining as a compilation technique for VLIW architectures. In *Proceedings of the 22nd International Workshop on Microprogramming and Microarchitecture*, pages 43–55, September 1989.

[27] J. C. Park and M. S. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.

[28] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 63–74, December 1994.

[29] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1):9–50, January 1993.

[30] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 158–169, December 1992.

[31] M. Schlansker and V. Kathail. Acceleration of first and higher order recurrences on processors with instruction level parallelism. In *Proceedings of Languages and Compilers for Parallel Computing, 6th International Workskop*, August 1993.

[32] M. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 40–51, December 1994.