Extending List Scheduling to Consider Execution Frequency*

Michael J. Bourke III
Superior Software
318 N. First Street
Brighton, MI 48116

Philip H. Sweany
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295

Steven J. Beaty
3612 Chipperfield Court
Fort Collins, Colorado, 80525

Abstract

Frequency-Based List Scheduling (FBLS) extends standard List Scheduling by considering execution frequencies within a schedule. This is useful for global instruction scheduling methods that schedule groups of basic blocks, called meta-blocks, as though they were a single block.

Traditional local schedulers operate on the premise that each instruction is executed the same number of times as every other instruction in the "block", an unwarrented assumption for meta-blocks. This assumption can lead meta-blocks schedulers to produce inefficient code. FBLS provides an answer to this problem by considering the differing execution frequencies within meta-blocks when scheduling operations.

To evaluate our contention that FBLS is a useful extension to standard list scheduling, we implemented FBLS and compared it to standard list scheduling within the context of dominator-path scheduling [1], a meta-block global scheduling algorithm. Experimental results show overall run-time improvement of 10.9% for livermore loops.

1 Introduction

Architectures exhibiting instruction-level parallelism (ILP), such as superscalar, VLIW, and superpipelined machines, have become increasingly popular. [2] To best exploit instruction-level parallelism in these machines, an instruction scheduling phase is required during compilation. Instruction scheduling is typically classified as *local* if it considers code only within a basic block and *global* if it schedules multiple basic blocks. Local scheduling methods are well known (Beaty provides a summary [3].) Local instruction scheduling's largest impediment is its inability to consider context from surrounding blocks. While local scheduling can find parallelism within a basic block, it can do nothing to exploit parallelism between basic blocks.

*This research is partially supported by NSF Grant CCR-9308348. Email addresses: *mjbourke@cs.mtu.edu*, *sweany@cs.mtu.edu*., and *beaty@lance.colostate.edu*

Among global instruction scheduling methods, several use a local scheduler to place operations into instructions. These include *trace scheduling* [4, 5, 6] and *dominator-path scheduling* [7, 1] which schedule multiple basic blocks as though they were a single basic block. We call these combined blocks which are treated as a single basic block *meta-blocks.* Using a local scheduler for meta-blocks adds complexity to the scheduling algorithm that has not been properly addressed. Specifically, local schedulers operate on the premise that each instruction is executed exactly the same number of times as every other instruction in the "block." As local scheduling is extended to handle the meta-blocks selected for optimization, however, it is no longer true that each instruction will be executed with the same frequency. Therefore, such meta-blocks violate one of the main assumptions of local scheduling, namely that every instruction in the block will be executed the same number of times. This motivates the search for a scheduler which places operations based in part on the expected execution frequencies of different portions of the meta-block being scheduled. *Frequency-Based List Scheduling* (FBLS) is such a scheduler.

Although the instruction scheduling methods discussed here could be incorporated into any ILP compiler, some discussion of the implementation context is appropriate. The algorithms described here have been implemented in the Rocket compiler [8, 3, 7], an instruction-scheduling C compiler retargetable for a broad class of ILP architectures. Current targets include two commercial superscalar architectures (an IBM RS6000 computer and a computer based on an Intel i860 chip) as well as several hypothetical LIW processors. Rocket's global optimization includes common subexpression elimination, copy propagation, constant folding, constant propagation, algebraic simplification, induction variable simplification, and reduction in strength. Aho, et al. [9]. describe these "traditional" compiler optimizations. Rocket includes local instruction scheduling, a form of trace scheduling [10] and dominator-path scheduling [7]. Register assignment via graph coloring can either immediately precede or follow instruction scheduling. The issues involved in choosing early or late register assignment

are discussed in [8]. Thorough discussions of both the local scheduler [3, 11] and the Rocket's global scheduling algorithms [7, 1] are available elsewhere. We shall refer to such details only when relevant to the discussion at hand.

In the remainder of this paper, Section 2 motivates the need for FBLS by discussing the use of meta-blocks in trace scheduling and dominator-path scheduling; Section 3 describes *list scheduling*, a popular technique used for local scheduling; Section 4 introduces Frequency-Based List Scheduling; Section 5 describes an experimental comparison of FBLS and list scheduling on a small test suite of C programs compiled for an architecture with limited ILP; and Section 6 presents our conclusions.

2 Global Scheduling Using Meta-Blocks

A considerable body of research has shown that to exploit the considerable amount of ILP that exists within most programs, a global scheduling technique must be used [12, 13, 14, 15, 16]. In this paper, we concentrate on a global scheduler that uses a local scheduling algorithm to schedule meta-blocks, namely dominator-path scheduling (DPS). We shall see how DPS exploits parallelism by scheduling meta-blocks and thus motivate why DPS benefits from FBLS.

2.1 Dominator-Path Scheduling

Dominator-Path Scheduling (DPS) [7, 1] is a global scheduling method in which parallelism in multiple basic blocks can be exploited without the cost of trace scheduling's bookkeeping operations. This makes DPS an attractive global scheduling technique for architectures with limited ILP. Like trace scheduling, DPS selects groups of blocks to schedule as a single meta-block. Unlike trace scheduling which chooses adjacent control-flow graph (CFG) nodes, DPS builds meta-blocks from paths in the dominator tree for a function. The *dominator tree* for a function is a representation of the *dominator relation* among the blocks of the function's CFG. A basic block, D, dominates another block B, if all paths from the root of the CFG to B must pass through D.

DPS uses an extension to Reif and Tarjan's fast symbolic covers algorithm [17] to build a dominator tree where tree nodes (basic blocks) are decorated with dataflow information. Sweany describes this extension in [7]. After building the dominator tree a path in the dominator tree is selected for scheduling and combined into a single meta-block. This resulting meta-block is scheduled using the local scheduler. During scheduling, operations are free to move (where not confined by data dependence) across block boundaries to exploit parallelism that exists on a dominator path. Due to the conservative nature of the data flow analysis used, operations cross block boundaries only when no compensation code is required to ensure correct program semantics.

2.2 List Scheduling Limitations

DPS suffers from local instruction scheduling's inability to consider differing execution frequencies within meta-blocks as operations are scheduled throughout the combined meta-block. A traditional local scheduler moves operations without regard for the potential execution frequency of the locale in which an operation is to be finally placed. Thus, the scheduler may move an operation from a block that has a relatively low execution frequency to one that has a higher frequency. This could result in a schedule that takes more instruction cycles to execute than that generated by an instruction scheduler which does not allow for inter-block code movement. Thus, DPS will benefit from an instruction scheduler like FBLS which considers execution frequency when placing operations.

DPS meta-blocks are not restricted to a single loop. In fact, one of the largest benefits of DPS is the ability to move operations across loop edges. It is expected that the estimated execution frequencies may vary greatly within a meta-block using DPS, and the difference between the best and worst block could be significant. This difference should allow for considerable improvement by FBLS when instructions are scheduled in the best location within the meta-block.

3 List Scheduling

The pursuit of improved scheduling techniques has been driven in part by the popularity of ILP architectures. In the past, several techniques for local scheduling have been studied including Linear Analysis, Critical Path, Branch and Bound, and List Scheduling [18]. This paper will only be concerned with List Scheduling, a derivative of Branch and Bound which attempts to obtain a good schedule in reasonable time (in contrast to Branch and Bound's guarantee of an optimal schedule in exponential time.)

List scheduling is commonly used in practice and serves both as the foundation for FBLS and as the yardstick by which we shall measure the efficacy of FBLS. List scheduling takes as input some representation of the operations to be scheduled. This input is usually a *Data Dependence Dag* (DDD), in which DDD nodes represent operations to schedule and arcs correspond to data dependences between DDD nodes. The DDD's directed edges indicate that a node **x** preceding a node **y** constrains **x** to occur no later than **y**. More sophisticated systems [19, 20] label edges from **x** to **y** with a pair of non-negative integers (min,max) indicating

that **y** can execute no sooner than min cycles after **x** and no later than max cycles after **x**. For example, if **x** placed a value on a bus that **y** read, an edge from **x** to **y** would establish a "data dependence" with timing (0,0) indicating that the *read* must follow the *write* in the same instruction. In contrast, if **x** assigned a value to a register subsequently read by **y** and the target machine did not permit reading a register after it is written in the same instruction, the edge from **x** to **y** would include timing $(1,\infty)$, specifying that **y** must follow **x** by at least one instruction, but can be placed any number of instructions after **x**. Throughout the remainder of this paper, we shall assume the use of DDD edges with min and max times.

List scheduling is based upon the maintenance of a *Data Ready Set* (DRS). The DRS contains all DDD nodes that are currently ready to be scheduled (i.e. have no unscheduled predecessors in the DDD.) A DDD node becomes data ready when all the DDD nodes which produce data for it have been scheduled. The scheduler selects the best node from the DRS to schedule based on a heuristic choice. The selected DDD node is inserted in the schedule at the point nearest its last dependence (or at the start of the schedule if there is no dependence) such that no machine resource conflict exists. Note that placing the node as close as possible to the last dependence is a heuristic and might not always produce the best code. After each DDD node is scheduled, its children (nodes dependent on it) are evaluated for data readiness. When a child becomes data ready, it is added to the data ready set. This scheduling process is repeated until all the nodes have been scheduled.

Traditionally, list scheduling is performed an instruction at a time — by which we mean that instruction n is completely scheduled before moving to instruction $n+1$. While this works, it is unnecessarily restrictive. There is no reason why scheduling can't be filling any number of instructions at the same time. In order to allow this out-of-order filling of instructions, we need an additional method to specify not only that all predecessors of a node have been scheduled, but also the range of instructions into which a DDD node can be placed. In our model of list scheduling, each DDD node, N, includes data fields which specify the earliest and latest possible instructions during which N might be scheduled. These *absolute times* for a node are determined by combination of the absolute times of a node's predecessors and the timings on the arcs from those (scheduled) predecessors, and will change as a node's predecessors are scheduled. An efficient algorithm to update nodes' absolute times during scheduling is given in: [3, 11]. While not essential to perform list scheduling, the absolute node times provide flexibility in scheduling which proves useful for FBLS. Using the absolute instruction times we can schedule nodes whenever they become data ready, rather than requiring that we fill an instruction at a time before moving on. This ability to fill instructions out-of-order based on the absolute times of the DDD nodes is a feature of the Rocket local scheduler and plays a significant part in FBLS' implementation.

There are generally multiple data-ready nodes at any point during scheduling, as should be expected with any program that exhibits ILP. Thus, the effect of heuristics on the efficiency of generated schedules (and even the ability to find a schedule) is an important concern. Beaty [3] discusses the relative importance of different oft-used heuristics and gives an extensive list of heuristics used in practice.

4 Frequency-Based List Scheduling

The ability to consider varying instruction frequencies when scheduling meta-blocks can be a substantial benefit to dominator-path scheduling (DPS). To date, however, such benefit has not been available because of limitations imposed by local scheduling. To best use DPS we need an instruction scheduler which will account for the differing cost of the instructions of a meta-block. If a single meta-block includes multiple nesting levels, we want the scheduler to recognize that instructions added to blocks with higher nesting levels are more costly than those at lower nesting levels. Even within a loop, there exists the potential for considerable variation in the execution frequencies of different blocks in the meta-block due to control flow. Of course variable execution frequency is not an issue in traditional local scheduling because within the context of a single basic block, each DDD node is executed the same number of times — namely once for each time execution enters the block.

Frequency-Based List Scheduling (FBLS) is an extension of list scheduling which provides an answer to this difficulty by considering that execution frequencies differ within sections of the meta-blocks. FBLS uses a greedy method to place DDD nodes in the lowest-cost instruction possible.

FBLS amends the basic list scheduling algorithm by revising only the DDD node placement policy in an attempt to reduce the run-time cycles required to execute a meta-block. To do this, FBLS modifies list scheduling so that once a DDD node has been chosen for scheduling, a two phase approach for operator placement is used. The first phase attempts to schedule a DDD node only within an already existing instruction with which the node can execute in parallel. If the scheduler cannot locate such an existing instruction, a second phase is used. The second phase creates a new instruction in the "best" portion of the meta-block where all the dependences of the selected node

```
Algorithm Frequency-Based List Scheduling
Input:
        Data Dependence Dag (DDD) representing operations to schedule
        Data Ready Set (DRS) of DDD nodes ready to schedule
        For each node, N, of DDD, the earliest instruction, EI in which
                N may be scheduled, based upon scheduled predecessors
        For each node, N, of DDD, the latest instruction, LI in which
                N may be scheduled, based upon scheduled predecessors
#       A sorted list, LF, of sections within the DDD ordered by
#               the frequency of the DDD segments

Output:
        Schedule of Instructions for the DDD, representing a meta-block

Algorithm:
        While DRS not empty
                Heuristically choose a node, N, from the DRS to schedule next
                SchedulePtr = N.EI
                Scheduled = FALSE
                        While ((SchedulePtr ≤ N.LI) And (Not Scheduled))
#                               If N can be placed in Schedule[SchedulePtr]
#                               AND Schedule[SchedulePtr] is non-empty
                                        Add N to Schedule[SchedulePtr]
                                        Scheduled = True
                                        For each successor, S, of N
                                                Remove arc N → S
                                                Add S to DRS, if it has no predecessors
                                else
                                        Increment SchedulePtr
                        End While

#                       If not Scheduled (Pass 1 failed, try Pass 2)
#                               Let B be that portion of the DDD with
#                               lowest execution frequency which overlaps
#                               the range (N.EI,N.FE)

#                               SchedulePtr = max(N.EI, B.start)
#                               Repeat the while loop above WITHOUT
#                               the restriction that N must be placed in
#                               an existing instruction.
        End While DRS not empty
        If one or more DDD nodes not scheduled
                Scheduling has failed!
```

Figure 1: Frequency-Base List Scheduling Algorithm

are met. The "best" portion is defined as the area in the meta-block that has the lowest execution frequency within the possible range of the DDD node to be placed. This scheme presupposes a mechanism to partition the DDD for a meta-block into sections of differing execution frequencies and the further ability to place an instruction in any partition. Techniques to accomplish this will be discussed shortly when we shift attention to implementation detail. In essence, FBLS first attempts to schedule a node "for free" by overlapping it with an existing instruction, and only when such an attempt fails, does FBLS create a new instruction. Whenever a new instruction is created it will be in the lowest-cost possible portion of the meta-block. Figure 1 provides an algorithmic description of FBLS. To facilitate comparison between the FBLS algorithm and that for simple list scheduling, any line which is added (or changed) for the FBLS algorithm is marked with a #.

4.1 Frequency-Based List Scheduling Implementation

The foundation of FBLS is to schedule a DDD for a meta-block such that instructions are placed in lower-frequency portions of the meta-block whenever possible. This requires a mechanism to associate an execution frequency with each instruction generated. There may be several ways to accomplish this feat. The one discussed here (and implemented in Rocket) is to add special nodes to the DDD for a meta-block and to use those nodes both to represent the execution frequencies within the meta-block and to drive the scheduling process by affecting each DDD node's absolute instruction times.

Since each basic block included in a meta-block has its own execution frequency, we designate for each block in the meta-block a single DDD node as a BlockStart node. This BlockStart node represents the start of the instructions that will be executed in that basic block. We use the BlockStart nodes to anchor a block and restrict motion of nodes into and out of basic blocks when such motion would be illegal in whatever global scheduling strategy is employed[1]. DDD dependence arcs are added between BlockStart nodes to ensure the correct relative placement of the basic blocks within a meta-block. As an example of how the BlockStart nodes are used to ensure only legal inter-block motion, consider an arbitrary meta-block containing five basic blocks, which we shall name B1-B5. For simplicity assume that block B_i precedes block B_j in the meta-block iff $i \leq j$. Then any node, O, which originally resides in block B_i and which cannot (due to restrictions of the global scheduling algorithm used) move to block B_j would include a dependence arc to block B_j's BlockStart node. This would require that, in the final schedule, O must be scheduled in an instruction preceding the one in which B_j's BlockStart node is scheduled (but after B_i's BlockStart node, of course.) We use this same principle to determine the execution frequency of scheduled instructions. Any instruction which resides (in the final schedule) between B_i's BlockStart node and B_{i+1}'s BlockStart node will be executed with the frequency of block B_i.

Our task then, is to place the BlockStart "anchors" in the list of scheduled instructions such that sufficient instructions are available to schedule other DDD nodes between the BlockStart nodes. We do this by placing all BlockStart nodes at appropriate intervals in the scheduled instruction array *before* any other nodes are placed, even though the BlockStart nodes may not fit the data-ready criterion. The capability for such out-of-order placement of nodes does add implementation complexity to the local instruction scheduler but the flexibility it offers is quite valuable. In fact, this is where the use of absolute instructions times within the node structure becomes important. While placing the BlockStart nodes the absolute times for other nodes dependent upon the BlockStart nodes automatically are updated. Returning briefly to our generic five-block example meta-block, we might decide a priori that no more than 300 instructions should ever be required to schedule all DDD nodes of any block. Then we initially place B_i's BlockStart node at instruction $300 * (i-1)$ for $1 \leq i \leq 5$. This leaves ample instructions to fill in DDD nodes between the (already scheduled) BlockStart nodes and also provides a mechanism to know the execution frequency of each instruction. Any node scheduled in instruction X will be executed with the same frequency as instruction I_k where $k = Xdiv300 + 1$. Once the BlockStart nodes are scheduled, their fixed absolute times drive the absolute times of the other DDD nodes and scheduling proceeds normally.

Given the above method for placing BlockStart nodes and thereby restricting other DDD nodes to instruction ranges over legal basic block boundaries, it is a relatively easy task to determine which basic block range within a DDD node's possible absolute times corresponds to the lowest execution frequency. FBLS merely looks at each BlockStart anchor within the range of each node's absolute min and max times.

It should be noted that one consequence of the chosen method for considering execution frequency when placing DDD nodes is some loss of scheduling flexibility *within* the meta-block itself. The main rationale for both trace scheduling and dominator-path scheduling is that a larger scheduling context will lead to better schedules. By partitioning the meta-block, we lose some (but not all) of that

[1] In Rocket, both trace scheduling and dominator-path scheduling use BlockStart nodes to enforce algorithmic restrictions on movement between blocks. See [7] for details.

flexibility. Specifically, if we place *every* BlockStart node in the schedule, this denotes that each block has a different execution frequency than its neighbors, and can limit our flexibility somewhat more than is necessary. For example, we might well have a meta-block in which contiguous blocks have the same execution frequency[2]. In such a case FBLS would sacrifice scheduling flexibility needlessly if it "anchored" every BlockStart node. Instead, if multiple consecutive blocks within a meta-block have the same execution frequency, FBLS will only anchor the first block's BlockStart node. Returning once again to our generic five-block meta-block, let us assume that we are using 300 as the maximum number of instructions for any block. Given the following execution frequencies,

B1 = 15
B2 = 15
B3 = 150
B4 = 150
B5 = 15

FBLS would anchor the BlockStart nodes for B1 (at instruction 0), B3 (at instruction 600) and B5 (at 1200), while the BlockStart nodes for B2 and B4 would be allowed to float within the confines of the BlockStart nodes for the previous and next blocks. This provides more flexibility for DDD nodes which could be scheduled in either B1 or B2 (and similarly for nodes which can be placed in either B3 or B4.)

4.2 FBLS Example

To illustrate how FBLS works and how it can lead to improved execution frequency, let us consider a small example. As our example code, consider the C code in Figure 2. This code fragment should be thought of as part of a larger program which will lead to a meta-block we wish to schedule.

We shall assume that we wish to compile this code fragment for an LIW architecture with the following characteristics:

- Any instruction can include one integer operation and one floating point operation. Thus, for our example, we will include at most one operation per instruction, as the example includes no floating point computation.
- Integer multiply is pipelined and requires 4 cycles to complete.
- Integer modulus is pipelined and requires 10 cycles to complete.

[2]While this would be unusual in trace scheduling where the blocks within a meta-block are lexically adjacent, it is quite the normal condition in dominator-path scheduling where a post-dominator block often follows its post-dominatee.

```
int A[SIZE];
    .
    .
    .

max = 0;
for( i = 0; i < SIZE; i++ )
{
    A[i] = (i * 32767) % 1000;
    if( A[i] > max )
    max = A[i];
}
mid = SIZE / 2;
low = 0;
high = SIZE;
sum = 0;
prod = 1;
maxsum = max * SIZE;
    .
    .
    .
```

Figure 2: Example C Code

- Loads and stores are pipelined and require two cycles to complete.
- All other instructions are completed in a single cycle.

Admittedly this is an extremely limited ILP architecture, but it will allow the example to demonstrate the potential benefit of FBLS scheduling without a large amount of superfluous detail. A control flow graph for the example code is shown in Figure 3. Notice that the basic blocks are decorated with a representation of the Rocket intermediate code which would be generated for each block. This example includes the effects of standard optimizations such as common subexpression elimination, copy propagation, dead code removal, and induction variable simplification.

To perform dominator-path scheduling for the chosen example, Rocket would build the dominator tree for the control-flow graph of Figure 3. Let us assume that the meta-block to be scheduled would include basic blocks B3, B4, B7, and B8. Consider what local scheduling might construct as a schedule for each block, shown in Figure 4. The assembly language chosen is meant to be an intuitive one rather than actually representative of any real assembly language. Notice that blocks B4 and B7 constitute a loop and if SIZE is 1000, then we would expect each of B4 and B7 to execute 1000 times while blocks B3 and B8 would execute once each. Since the schedules for B4 and B7

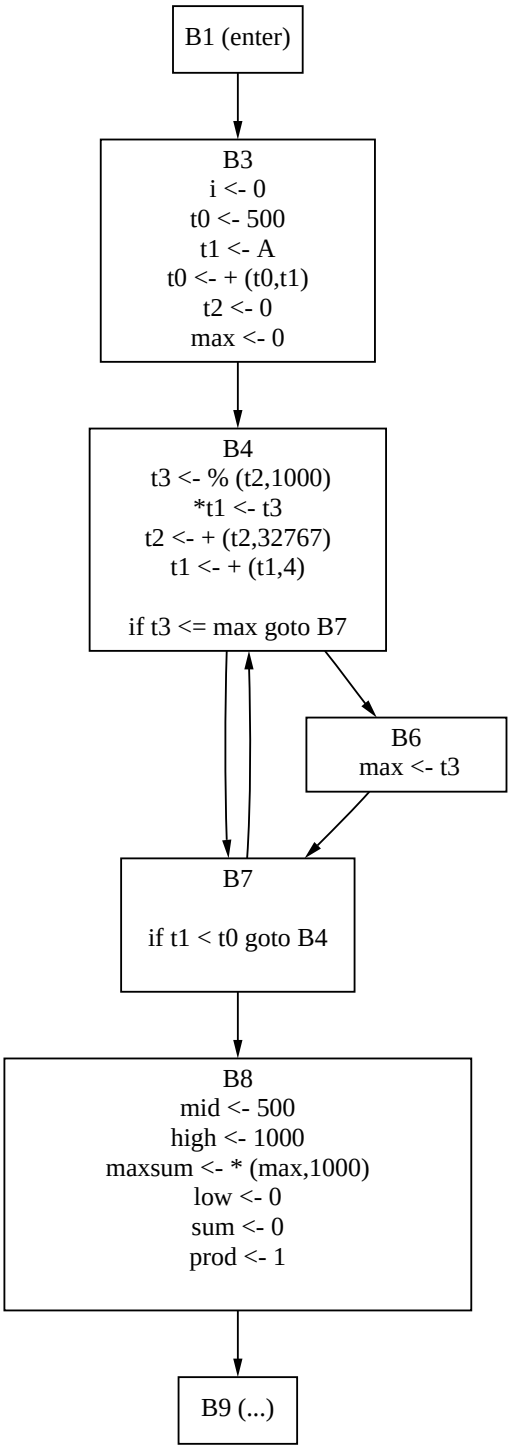

Figure 3: Control Flow Graph for Example C Code

```
B3:
        i = 0
        t0 = 500
        t1 = &A
        t0 = t0 + t1
        t2 = 0
        max = 0
B4:
        t3 = t2 % 1000
        t2 = t2 + 32767
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        0(t1) = t3
        t1 = t1 + 4
        ble t3,max,B7

B7:
        blt t1,t0,B4

B8:
        maxsum = max * 1000
        mid = 500
        high = 1000
        low = 0
        sum = 0
        prod = 1
```

Figure 4: Local Scheduling of Blocks B3, B4, B7, B8

combined require 14 instructions while the schedules for B3 and B8 require 12 instructions, we would expect 14012 cycles would be required to execute these blocks if local scheduling were used. We're ignoring block B6 because it is not in the chosen meta-block.

Now consider the schedule we might expect from dominator-path scheduling, with blocks B3, B4, B7, and B8 considered a single block for scheduling purposes. Using a traditional local scheduler Rocket would attempt to minimize the instructions required to schedule the code. Figure 5 shows a likely schedule. Note that the total number of instructions required for the meta-block is reduced from 26 to 22. However, this is due to moving instructions out of block B8 and into the loop (B4, B7). While this

```
B3:
        i = 0
        t0 = 500
        t1 = &A
        t0 = t0 + t1
        t2 = 0
        max = 0
B4:
        t3 = t2 % 1000
        t2 = t2 + 32767
        mid = 500
        high = 1000
        low = 0
        sum = 0
        prod = 1
        NOP
        NOP
        NOP
        0(t1) = t3
        ble t3,max,B7

B7:
        maxsum = max * 1000
        t1 = t1 + 4
        NOP
        blt t1,t0,B4

B8:
```

Figure 5: Dominator-Path Scheduling — Traditional List Scheduler

gives us fewer instructions overall, it increases the number of instructions in the loop.[3] So, for DPS with a traditional local scheduler, the cost of the meta-block would be 16006, even higher than local scheduling. To remedy the problem, we need FBLS to consider expected execution frequencies. Figure 6 shows the code we could expect from DPS with an FBLS local scheduler. Now we require 24 instructions, with 14 still in the loop blocks, B4 and B7. The total cycles required for the DPS-FBLS scheduled loop is 14010, a modest (2 cycle) improvement over local scheduling for this example, but a substantial improvement over DPS with a traditional scheduler.

[3] It might appear that the computation of maxsum could be moved to block B4, but it cannot since the value of max *could* be redefined in block B6.

```
B3:
        i = 0
        t0 = 500
        t1 = &A
        t0 = t0 + t1
        t2 = 0
        max = 0
B4:
        t3 = t2 % 1000
        t2 = t2 + 32767
        mid = 500
        high = 1000
        low = 0
        sum = 0
        prod = 1
        NOP
        NOP
        NOP
        0(t1) = t3
        ble t3,max,B7

B7:
        t1 = t1 + 4
        blt t1,t0,B4

B8:
        maxsum = max * 1000
        NOP
        NOP
        NOP
```

Figure 6: Dominator-Path Scheduling — FBLS List Scheduler

5 Experimental Evaluation

To evaluate our contention that FBLS is a useful extension to standard list scheduling when scheduling meta-blocks, we compared FBLS to standard list scheduling within the context of dominator-path scheduling for the well-known Livermore loops [21] benchmark program. To perform a comparison of the two scheduling methods, we needed an architecture for which to compile code. We chose Poplawski's URM machine ([22]) as it provided us with a simulator and analyzer for a "vanilla" ILP architecture. The compiler used profile information to provide the necessary "estimates" of each basic block's execution frequency.

While the parameterized URM machine supports a wide range of architectural features and allows the user to specify different levels of ILP in a superscalar format, we chose as a base machine a load-store architecture which allows up to 2 integer and 1 floating point instructions to be issued each cycle. Integer loads and stores "used" an integer instruction while loads/stores of floating point values were considered floating point instructions. At most one memory operation could be issued in a cycle. Instruction issue was restricted to be in-order of the assembly code presented to the URM simulator.

To compare standard list scheduling with FBLS list scheduling Livermore loops was compiled using dominator-path scheduling and the total cycles required to execute the programs were recorded. Rocket chose 31 dominator paths to schedule as meta-blocks and the loops compiled with FBLS required 8.6% fewer cycles to complete than the same meta-blocks with standard list scheduling. This strongly suggests that FBLS does indeed provide better scheduling for meta-blocks. To evaluate how FBLS would perform with slightly more available hardware parallelism, we repeated the experiment with all URM parameters the same except that we allowed up to 4 integer and 2 floating point instructions to be issued in any cycle. On this more parallel machine, FBLS out-performed standard list scheduling by 10.9%. Given the example of Section 4 one might ask how DPS without FBLS fared against local scheduling alone. They produced quite similar results for Livermore, with DPS and traditional scheduling winning by an insignificant .7%. Therefore using FBLS allowed DPS to improve upon local scheduling while using a traditional scheduler did not.

The Livermore run-time results suggest that FBLS can out-perform standard list scheduling for some significant code. However, as suggested in Section 4, the smaller context for list scheduling certainly allows the possibility that FBLS could "lose" compared to standard scheduling for some meta-blocks. To obtain more detailed results than possible from run-time comparison, we performed a static analysis of the cycles required for each of the 31 dominator paths scheduled in the Livermore loops experiment. First, a word about how Rocket models superscalar architectures when scheduling. Essentially Rocket treats superscalars as LIW architectures, inserting NOPs into the schedule when it determines that the superscalar architecture will be unable to issue an instruction for any particular cycle. While these NOPs are commented out in the assembly code, and thus ignored by the superscalar URM simulator, they allow Rocket to accurately predict the number of cycles required to execute a basic block. This number of cycles, NC, is merely the number of instructions Rocket determines would be needed by an LIW with the same parallelism capabilities as the superscalar being modeled. Having computed NC for each block B_i in a scheduled meta-block, statically computing the cost for a meta-block merely requires summing (for each block) the number of times the block is executed multiplied by NC. Thus, if F_i is the frequency that basic block, B_i is executed and NC_i is the number of cycles required for block, B_i, then the meta-block cost, MBC is: $MBC = \sum_{i=1}^{n} F_i * NC_i$ Using this static measurement, we looked at each of the 31 meta-blocks in Livermore loops and found that for two of the meta-blocks, standard list scheduling produced better results (fewer cycles) than FBLS. For five of the meta-blocks, FBLS and standard list scheduling produced the same results and for 24 of the meta-blocks FBLS out-performed standard scheduling. This supports our contention that, while FBLS might lead to less efficient schedules for some meta-blocks, schedules for the majority of meta-blocks which include basic blocks from different nesting levels would benefit from FBLS.

6 Conclusions

Current list scheduling techniques are not adequate to schedule meta-blocks that include variable execution frequency among instructions. The advent of global scheduling techniques such as dominator-path scheduling which perform local scheduling on meta-blocks requires more sophisticated methods. Frequency-Based List Scheduling is an instruction scheduling technique based upon list scheduling that considers execution frequency of instructions when placing operations. As such, FBLS promises to find better schedules for meta-blocks. In a small experimental sample FBLS showed improvement of roughly 10% over a traditional list scheduler when using dominator-path scheduling to compile Livermore loops for an architecture with limited instruction-level parallelism. We conclude that when used for scheduling meta-blocks, local scheduling techniques need to consider variable execution frequency within the meta-block. FBLS provides an effective way to make use of such meta-block frequency information.

References

[1] P. Sweany and S. Beaty, "Dominator-path scheduling — a global scheduling method," in *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)*, (Portland, OR), pp. 260–263, December 1992.

[2] B. R. Rau and J. A. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *The Journal of Supercomputing*, vol. 7, pp. 9–50, 1993.

[3] S. Beaty, *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Mechanical Engineering Department, Colorado State University, Fort Collins, Colorado, 1991.

[4] J. Fisher, *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling*. PhD thesis, Courant Institute of Mathematical Sciences, New York University, New York, NY, October 1979.

[5] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, 1985. PhD thesis, Yale, 1984.

[6] S. A. Mahlke, W. Y. Chen, W. mei W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for VLIW and superscalar processors," in *asplos5*, vol. 27, (Boston, MA), pp. 238–247, oct 1992.

[7] P. Sweany, *Inter-Block Code Motion without Copies*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1992.

[8] P. Sweany and S. Beaty, "Post-compaction register assignment in a retargetable compiler," in *Proceedings of the 23th Microprogramming Workshop (MICRO-23)*, (Orlando, FL), pp. 107–116, November 1990.

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

[10] M. Howland, R. Mueller, and P. Sweany, "Trace scheduling optimization in a retargetable microcode compiler," in *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, (Colorado Springs, CO), December 1987.

[11] S. Beaty, "Lookahead scheduling," in *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)*, (Portland, OR), pp. 256–259, December 1992.

[12] G. Tjaden and M. Flynn, "Detection and parallel execution of independent instructions," *IEEE Transactions on Computers*, vol. C-19, no. 10, pp. 889–895, Oct 1970.

[13] A. Nicolau and J. Fisher, "Measuring the parallelism available for very long instruction word architectures," *IEEE Transactions on Computers*, vol. 33, no. 11, pp. 968–976, Nov 1984.

[14] D. W. Wall, "Limits of instruction-level parallelism," in *asplos4*, (Santa Clara, CA), pp. 176–188, April 1991.

[15] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream prallelism is greater than two," in *19th Annual International Symposium on Computer Architecture*, pp. 276–286, ACM, 1992.

[16] A. K. Uht, "Extraction of massive instruction level parallelism," *Computer Architecture News*, vol. 21, pp. 5–12, June 1993.

[17] J. Reif and R. Tarjan, "Symbolic program analysis in almost-linear time," *SIAM Journal of Computing*, vol. 11, no. 1, pp. 81–93, February 1981.

[18] D. Landskov, S. Davidson, B. Shriver, and P. Mallett, "Local microcode compaction techniques," *ACM Computing Surveys*, vol. 12, no. 3, pp. 261–294, September 1980.

[19] S. Vegdahl, *Local Code Generation and Compaction in Optimizing Microcode Compilers*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1982.

[20] V. Allan, *A Critical Analysis of the Global Optimization Problem for Horizontal Microcode*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1986.

[21] F. McMahon, "The Livermore FORTRAN kernels: A computer test of numerical performance range," tech. rep., Lawrence Livermore National Laboratory, December 1986.

[22] D. A. Poplawski, "The unlimited resource machine (URM)," Tech. Rep. CS-95-01, Department of Computer Science, Michigan Technological University, Houghton, January 1995.