Non-Local Instruction Scheduling with Limited Code Growth *

Keith D. Cooper and Philip J. Schielke

Department of Computer Science, Rice University, Houston Texas
keith@cs.rice.edu, phisch@cs.rice.edu

Abstract. Instruction scheduling is a necessary step in compiling for many modern microprocessors. Traditionally, global instruction scheduling techniques have outperformed local techniques. However many of the global scheduling techniques described in the literature have a side effect of increasing the size of compiled code. In an embedded system, the size of compiled code is often a critical issue. In such circumstances, the scheduler should use techniques that avoid increasing the size of the generated code. This paper explores two global scheduling techniques, extended basic block scheduling and dominator path scheduling, that do not increase the size of the object code, and in some cases may decrease it.

1 Introduction

The embedded systems environment presents unusual design challenges. These systems are constrained by size, power, and economics; these constraints introduce compilation issues not often considered for commodity microprocessors. One such problem is the size of compiled code. Many embedded systems have tight limits on the size of both RAM and ROM. To be successful, a compiler must generate code that runs well while operating within those limits.

The problem of code space reduction was studied in the 1970's and the early 1980's. In the last ten years, the issue has largely been ignored. During those ten years, the state of both processor architecture and compiler-based analysis and optimization have changed. To attack the size of compiled code for embedded systems, we must go back and re-examine current compiler-based techniques in light of their impact on code growth.

This paper examines the problem of scheduling instructions in a limited-memory environment. Instruction scheduling is one of the last phases performed by modern compilers. It is a code reordering transformation that attempts to hide the latencies inherent in modern day microprocessors. On processors that support instruction level parallelism, it may be possible to hide the latency of some high-latency operations by moving other operations into the "gaps" in the schedule.

* This work has been supported by DARPA and the USAF Research Laboratory through Award F30602-97-2-298.

Scheduling is an important problem for embedded systems, particularly those built around DSP-style processors. These microprocessors rely on compiler-based instruction scheduling to hide operation latencies and achieve reasonable performance. Unfortunately, many scheduling algorithms deliberately trade increased code size for improvements in running time. This paper looks at two techniques that avoid increasing code size and presents experimental data about their effectiveness relative to the classic technique—local list scheduling.

For some architectures, instruction scheduling is a necessary part of the process of ensuring correct execution. These machines rely on the compiler to insert NOPs to ensure that individual operations do not execute before their operands are ready. Most VLIW architectures have this property. On these machines, an improved schedule requires fewer NOPs; this can lead to a direct reduction in code space. If, on the other hand, the processor uses hardware interlocks to ensure that operands are available before their use, instruction scheduling becomes an optimization rather than a necessity. On these machines, NOP insertion is not an issue, so the scheduler is unlikely to make a significant reduction in code size.

In this paper, we focus on the VLIW-like machines without hardware interlocks. (Of course, good scheduling without code growth may be of interest on any machine.) For our discussion, we need to differentiate between *operations* and *instructions*. An operation is a single, indivisible command given to the hardware (*eg.* an add or load operation). An instruction is a set of operations that begin execution at the same time on different functional units.

Traditionally, compilers have scheduled each basic block in the program independently. The first step is to create a data precedence graph, or DPG, for the block. Nodes in this graph are operations in the block. An edge from node a to node b means that that operation b must complete its execution before operation a can begin. That is, operation a is data dependent on operation b. Once this graph is created it is scheduled using a list scheduler [17, 11].

Since basic blocks are usually rather short, the typical block contains a limited amount of instruction-level parallelism. To improve this situation, regional and global instruction scheduling methods have been developed. By looking at larger scopes, these methods often find more instruction-level parallelism to exploit. This paper examines two such techniques, *extended basic block scheduling* (EBBS) and *dominator path scheduling* (DPS). Both methods produce better results than scheduling a single basic block; this results in fewer wasted cycles and fewer inserted NOPs.

We selected these two techniques because neither increases code size. In the embedded systems environment, the compiler does not have the luxury of replicating code to improve running time. Instead, the compiler writer should pay close attention to the impact of each technique on code size. These scheduling techniques attempt to improve over local list scheduling by examining larger regions in the program; at the same time, they constrain the movement of instructions in a way that avoids replication. Thus, they represent a compromise between the desire for runtime speed and the real constraints of limited memory machines.

Section 2 provides a brief overview of prior work on global scheduling. In section 3 we explain in detail the two techniques used in our experiments: namely *extended basic block scheduling* (EBBS) and *dominator-path scheduling* (DPS). Section 4 describes our experiments and presents our experimental results.

2 Global Scheduling Techniques

Because basic blocks typically have a limited amount of parallelism [20], global scheduling methods have been developed in the hopes of improving program performance. All the global techniques we will be describing alter the *scope* of scheduling, and not the underlying scheduling algorithm. Each technique constructs some sequence of basic blocks and schedules the sequence as if it were a single basic block. Restrictions on moving operations between basic blocks are typically encoded in the DPG for the sequence.

The first automated global scheduling technique was trace scheduling, originally described by Fisher [8]. The technique has been used successfully in several research and industrial compilers [7, 18]. In trace scheduling, the most frequently executed acyclic path through the function is determined using profile information. This "trace" is treated like a large basic block. A DPG is created for the trace, and the trace is scheduled using a list scheduler. Restrictions on inter-block code motion are encoded in the DPG. After the first trace is scheduled, the next most frequently executed trace is scheduled, and so on. A certain amount of "bookkeeping" must be done when scheduling a trace. Any operation that moves above a join point in the trace must be copied into all other traces that enter the current trace at that point. Likewise, any operation that moves below a branch point must be copied into the other traces that exit the branch point, if the operation computes any values that are live in that trace.

One criticism of trace scheduling is its potential for code explosion due to the bookkeeping code. Fruedenberger, *et al.,* argue that this does not arise in practice [10]. They show an average code growth of six percent for the SPEC89 benchmark suite and detail ways to avoid bookkeeping (or compensation) code altogether. Restricting the trace scheduler to produce no compensation code only marginally degrades the performance of the scheduled code.

Hwu, *et. al.*, present another global scheduling technique called *superblock scheduling* [14]. It begins by constructing traces. All side entrances into the traces are removed by replicating blocks between the first side entrance and the end of the trace. This *tail duplication* process is repeated until all traces have a unique entry point. This method can lead to better runtime performance than trace scheduling, but the block duplication can increase code size. Several other techniques that benefit from code replication or growth have been used. These include Bernstein and Rodeh's "Global Instruction Scheduling" [4, 2], Ebcioglu and Nakatani's "Enhanced Percolation Scheduling" [6], and Gupta and Soffa's "Region Scheduling" [12].

3 The Two Techniques

In this section we look at two non-local scheduling techniques specifically designed to avoid increasing code size, namely dominator-path scheduling (DPS), and extended basic block scheduling (EBBS). We assume that, prior to scheduling, the program has been translated into an intermediate form consisting of basic blocks of operations. Control flow is indicated by edges between the basic blocks. We assume this control flow graph (CFG) has a unique entry block and a unique exit block.

3.1 Extended basic block scheduling

Little work has been published on scheduling over extended basic blocks. Freudenberger, *et. al.* show some results of scheduling over extended basic blocks, but only after doing some amount of loop unrolling [10]. Since we are striving for zero code growth, such loop unrolling is out of the question.

An extended basic block (or EBB) is a sequence of basic blocks, $B_1, \ldots, B_k$, such that, for $1 \leq i < k$, B_i is the only predecessor of B_{i+1} in the CFG, and B_1 may or may not have a unique predecessor [1]. For scheduling purposes, we view extended basic blocks as a partitioning of the CFG; a basic block is a member of only one EBB.

The first step in EBBS is to partition the CFG into extended basic blocks. We define the set of *header* blocks to be all those blocks that are the first block in some EBB. Initially, our set of headers consists of the start block and all blocks with more than one predecessor in the CFG. Once this initial set of headers is computed, we compute a weighted *size* for each basic block. The size for a header is set to zero. The size for all other blocks equals the total number of operations in the block weighted by their latencies plus the maximum size of all the block's successors in the CFG. To construct the EBB's, we maintain a worklist of header blocks. When a block B is pulled off the worklist, other blocks are added to its EBB based on sizes computed earlier. The successor of B in the CFG with the largest size is added to B's EBB. The other successors of B are added to to the worklist to become headers for some other EBB. This process continues for the new block, until no more eligible blocks are found for the current EBB. For each EBB, a DPG is constructed, and the EBB is scheduled with a list scheduler.

We must prohibit some operations from moving between the blocks of an EBB. Assume a block B_1 has successors $B_2, B_3, \ldots, B_n$ in the CFG. Further assume that B_2 is placed in the same EBB as B_1. We prohibit moving an operation from B_2 to B_1, and vice versa, if that operation defines a value that is live along some path from B_1 to B_i where $i \neq 2$. We call this set of values *path-live* with respect to B_2, or PL_{B_2}. The set is a portion of the set $liveout(B_1)$ as computed by the following equation:

$$PL_{B_2} = \bigcup_{B_i = B_3}^{B_n} livein(B_i)$$

Intuitively, we can't move the operation if any value it defines is used in some block other than B_1 or B_2 and that block is reachable from B_1 via some path not containing B_2. The operations that can be moved are called *partially dead* if they are in B_1 [16].

3.2 Dominator-path scheduling

DPS was originally described in Sweany's thesis [21]. Other work was done by Sweany and Beaty [22], and Huber [13].

We say a basic block B_1 *dominates* block B_2 if all paths from the start block of the CFG to B_2 must pass through B_1 [19]. If B_1 dominates B_2, and block B_2 executes on a given program run, then B_1 must also execute. We define the *immediate dominator* of a block B (or $idom(B)$) to be the dominator closest to B in the CFG. Each block must have a unique immediate dominator, except the start block which has no dominator. Let $G = (N, E)$ be a directed graph, where the set N is the set of basic blocks in the program, and define $E = \{(u, v) \mid u = idom(v)\}$. Since each block has a unique immediate dominator, this graph is a tree, called the *dominator-tree*. A dominator-path is any path between two nodes of the dominator-tree.

We now define two sets, $idef(B)$ and $iuse(B)$. For a basic block B, $idef(B)$ is the set of all values that may be defined on some path from $idom(B)$ to B (not including B or $idom(B)$.) Likewise $iuse(B)$, is the set of all values that may be used on some path from $idom(B)$ to B. The algorithm for efficiently computing these sets is given by Reif and Tarjan [23].

DPS schedules a dominator-path as if it were a single basic block. First, the blocks in the CFG must be partitioned into different dominator-paths. Huber describes several heuristics for doing path selection and reports on their relative success. We use a size heuristic similar to the one described above for EBBS. This is done via a bottom-up walk over the dominator-tree. The size of a leaf equals the latency-weighted number of operations in the block. For all other blocks, size equals the latency-weighted number of operations in the block plus the maximum size of all the block's children in the dominator-tree. When building the dominator-paths, we select the next block in the path by choosing the child in the dominator-tree with the largest size. All other children become the first block in some other dominator-path. Once the dominator-paths are selected, a DPG is created for each path, and the path is scheduled using a list scheduler. After each path is scheduled, liveness analysis and the $idef$ and $iuse$ sets must be recomputed to insure correctness.

When the compiler builds the DPG for the dominator-path, it adds edges to prevent motion of certain operations between basic blocks. Assume B_1 is the immediate dominator of B_2. Sweany's original formulation prohibited moving an operation from B_2 up into B_1 if that operation defined a value in $idef(B_2) \cup iuse(B_2)$, or if it referenced a value in $idef(B_2)$. That is, we don't want to move an operation that defines a value V above a use or definition of V in the CFG. Likewise, an operation that references V is not allowed to move above a definition of V. This strategy is safe when B_1 dominates B_2 *and* B_2 postdominates B_1.

However, Huber showed that in the general case this strategy is unsafe. Figure 1(a) illustrates the problem. In this simple CFG we show only the operations that use or define `r1`. Assume blocks B_1 and B_2 will be scheduled together. Clearly `r1` is not a member of $iuse(B_2)$ or $idef(B_2)$ since there are no blocks between B_1 and B_2. Thus, the definition of `r1` in B_2 could be unsafely hoisted into block B_1 causing the use of `r1` in B_4 to use the wrong value. Thus, Huber adds the restriction that any operation that defines a value in

$$idef(B_2) \cup iuse(B_2) \cup (liveout(B_1) - livein(B_2))$$

cannot be moved up from B_2 into B_1.

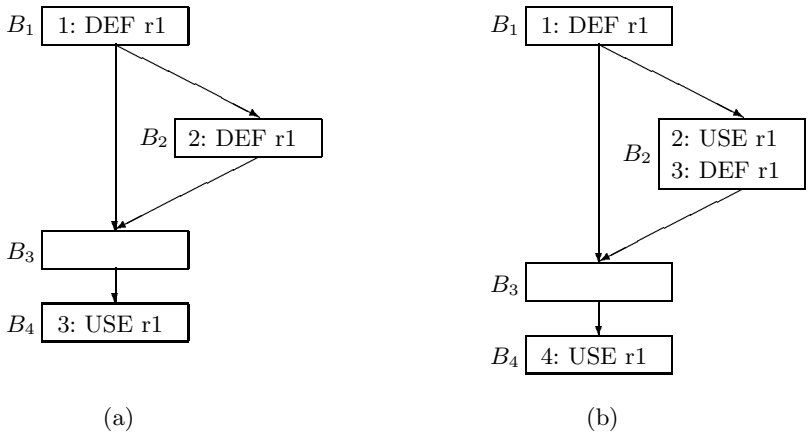

Fig. 1. DPS example

However, we have found that this formulation too is unsafe. Figure 1(b) demonstrates the problem. Again, assume that blocks B_1 and B_2 will be scheduled together. Note that $r1 \in liveout(B_1)$ and $r1 \in livein(B_2)$ since it is referenced before it is defined in B_2. Therefore, $r1$ is not in the set $liveout(B_1) - livein(B_2)$. Assuming operation 2 does not define anything that causes movement to be unsafe, we can move it up into block B_1. It would then be legal to move the operation 3 into B_1. Thus both operations in B_2 could be moved into block B_1, which would cause operation 4 in block B_4 to potentially get the wrong value. Once a dominator-path is selected for scheduling, no updating of the liveness information is done during the scheduling of that dominator-path. Some sort of incremental update would be one way to solve this problem, since moving operation 2 into B_1 would cause $r1$'s removal from the set $livein(B_2)$.

We use an approach that doesn't require incremental updates. What we really want to capture are those values that are live along paths other than paths from B_1 to B_2. This is fairly straightforward if B_1 is the only parent of B_2 in the CFG; we simply use the path-live notion discussed in the previous section. In

$dontdef = idef(B_2) \cup iuse(B_2)$
if B_2 does not post-dominate B_1
  then if B_1 is the predecessor of B_2 in CFG
    $dontdef = dontdef \cup PL_{B_2}$
  else
    $dontdef = dontdef \cup liveout(B_1)$
if B_2 and B_1 are in different loops
  then $dontdef = dontdef \cup liveout(B_1)$
    $dontdef = dontdef \cup memory\_values$

Fig. 2. Summary of prohibited moves between B_1 and B_2

other cases, we take the conservative approach and don't allow any operation that defines a value in $liveout(B_1)$ to move up.

Now we consider motion of an operation in the downward direction. Sweany does not allow an operation to move down the CFG, that is into block B_2 from its dominator B_1, but he does mention that this could be done if B_2 *post-dominates* B_1. A block B_2 post-dominates B_1 if every path from B_1 to the end of the CFG must pass through B_2 (simply the dominance relation in reverse). Huber allows operations to move down into the post-dominator if they don't define anything in $idef(B_2) \cup iuse(B_2)$ or use anything in $idef(B_2)$. No downward motion is allowed if B_2 does not post-dominate B_1. We take this one step further by allowing motion of an operation from B_1 into B_2 if B_1 is the predecessor of B_2 in the CFG, and the operation computes values that are only live along the edge (B_1, B_2). (This is the path-live notion from section 3.1.) In any other case where B_2 does not post-dominate B_1, we take the conservative approach and disallow any motion of operations that compute a value in $liveout(B_1)$.

Loops pose additional concerns. We must be careful not to allow any code that defines memory to move outside of its current loop or to a different loop nesting depth.[1] In addition to the restrictions described above, we disallow any operation that defines memory from moving between two blocks if they are in different loops or at different loop nesting levels. Finally, we don't allow an operation that defines registers in $liveout(B_1)$ to move between the two blocks.

To summarize, we disallow motion of an operation between block B_2 and its immediate dominator B_1 (forward or backward) if that operation defines a value in the set $dontdef$. This set is defined in figure 2. Additionally any operations that use a value in $idef(B_2)$ are not allowed to move.

4 Experimental Results

Our research compiler takes C or Fortran code and translates it into our assembly-like intermediate form, ILOC [5]. The ILOC code can then be passed to various op-

[1] Recall that scheduling follows optimization. The optimization should include some careful code motion [15].

timization passes. All the code for these experiments has been heavily optimized before being passed to the instruction scheduler. These optimizations include pointer analysis for the C codes, constant propagation, global value numbering, dead code elimination, operator strength reduction, lazy code motion, and register coalescing. No register allocation was performed before *or* after scheduling, as we wanted to completely isolate the effects of the scheduler. After optimization, the ILOC is translated into C, instrumented to report operation and instruction counts, and compiled. This code is then run.

A variety of C and Fortran benchmark codes were studied, including several from various versions of the SPEC benchmarks and the FMM test suite [9]. The C codes used are, `clean`, `compress`, `dfa`, `dhrystone`, `fft`, `go`, `jpeg`, `nsieve`, and `water`. All other benchmarks are Fortran codes. `clean` is an optimization pass from our compiler. `dfa` is a small program that implements the Knuth-Morris-Pratt string matching algorithm. `nsieve` computes prime numbers using the Sieve of Eratosthenes. `water` is from the SPLASH benchmark suite, and `fft` is a program that performs fast-fourier transforms.

4.1 A Generic VLIW Architecture

In the first set of experiments, we assume a VLIW-like architecture. This hypothetical architecture has two integer units, a floating point unit, a memory unit, and a branch unit. Up to four operations can be started in parallel. Each ILOC operation has a latency assigned to it. We assume that the latency of every operation is known at compile time. The architecture is completely pipelined, and NOPs must be inserted to ensure program correctness. We compare DPS and EBBS to scheduling over basic blocks. In each case the underlying scheduler is a list scheduler that assigns priorities to each operation based on the latency-weighted depth of the operation in the DPG. For both DPS and EBBS we select which blocks to schedule based on the size heuristic described above. In this experiment, we permit all blocks in a given EBB or dominator-path to be at any loop nesting level. Code is allowed to move between blocks as described above. One additional restriction on code movement is that we do not allow any operations that could cause an exception to be moved "up" in the CFG. We do not allow any divide operations, or loads from pointer memory (ILOC's `PLDor` operations), to move up.

Table 1 shows the dynamic instruction counts for our benchmark codes. This value can be thought of as the number of cycles required to execute the code. Both EBBS and DPS resulted in faster code than basic block scheduling. Slightly better than fifty per cent of the time DPS outperformed EBBS, and a few of these wins were substantial. On average EBBS produced a 6.5 per cent reduction in the number of dynamic instructions executed, and DPS produced a 7.5 per cent reduction.

Table 2 shows the static instruction counts for the same experiments. This corresponds to the "size" (number of instructions) of the object code. Note that all the object codes have the same number of operations; only the number of instructions changes. DPS did better by this metric in roughly the same number of

Table 1. Dynamic Instruction Counts for VLIW

	Basic Block	EBBS		DPS	
Benchmark	Dynamic Insts	Dynamic Insts	% decrease	Dynamic Insts	% decrease
clean	4515619	4113837	8.9	3969926	12.1
compress	10641037	9511683	10.6	9489915	10.8
dfa	696450	592836	14.9	625166	10.2
dhrystone	3660102	3340092	8.7	3220092	12.0
fft	22469970	22138422	1.5	22193147	1.2
go	589209782	527762311	10.4	521628685	11.5
jpeg	45900780	44107954	3.9	44040659	4.1
nsieve	2288889385	2254236158	1.5	2254236164	1.5
water	36111497	33544010	7.1	33253230	7.9
fmin	5370	4495	16.3	4100	23.6
rkf45	818884	731155	10.7	749565	8.5
seval	3340	3264	2.2	3261	2.4
solve	2813	2652	5.7	2627	6.6
svd	14649	13805	5.8	13921	5.0
urand	1117	1081	3.2	1093	2.1
zeroin	4603	4088	11.2	4035	12.3
applu	884028559	865609968	2.1	866257750	2.0
doduc	16953587	16122745	4.9	15248824	10.1
fpppp	95701038	90578189	5.4	89483748	6.5
matrix300	43073238	42802715	0.6	42803515	0.6
tomcatv	436717483	436706995	0.0	408090942	6.6

experiments. However, the static and dynamic improvements did not necessarily occur on the same codes. This demonstrates that smaller more compact code does not always results in enhanced runtime performance. On average EBBS reduced static code size by 10.9 per cent and DPS by 11.8 per cent.

When performing basic block scheduling, we found each block had an average of 6.8 operations (over all benchmarks). On average, an EBB consisted of 1.8 basic blocks and 12.4 operations. Dominator paths averaged 2.2 basic blocks and 15.1 operations, each.

We also measured the amount of time required to schedule. The scheduling times for each benchmark are shown in table 3. In two runs, the average scheduling time for all benchmarks was 88 seconds for basic block scheduling, 92 seconds for EBBS, and 2297 seconds for DPS. This comparison is a bit unfair. Several of our C codes have many functions in each ILOC module. Thus DPS is performing the dominator analysis for the whole file every time a dominator-path is scheduled. The go benchmark contributed 2109 seconds alone. We totaled times for the Fortran benchmarks (all ILOC files contain a single function), and a random sampling of the single function C codes (about 24 functions). The scheduling times were 56 seconds for basic block scheduling, 50 seconds for EBBS, and 105 seconds for DPS. If we eliminate fpppp, which actually scheduled faster with EBBS than basic block scheduling, we get times of 8 seconds, 10 seconds, and 49 seconds, respectively.

Table 2. Static Instruction Counts for VLIW

	Basic Block	EBBS		DPS	
Benchmark	Static Insts	Static Insts	% decrease	Static Insts	% decrease
clean	11479	10406	9.3	10439	9.1
compress	1601	1401	12.5	1403	12.4
dfa	1357	1040	23.4	1061	21.9
dhrystone	525	477	9.1	463	11.8
fft	2748	2554	7.1	2533	7.8
go	73528	62829	14.6	62059	15.6
jpeg	19825	18416	7.1	18486	6.8
nsieve	274	258	5.8	256	6.6
water	6485	6094	6.0	5962	8.1
fmin	712	503	29.4	447	37.2
rkf45	2389	2057	13.9	2032	14.9
seval	1057	995	5.9	1014	4.1
solve	1012	940	7.1	933	7.8
svd	2496	2245	10.1	2278	8.8
urand	192	172	10.4	168	12.5
zeroin	545	446	18.2	443	18.7
applu	13403	13008	2.9	12920	3.6
doduc	42135	38543	8.5	37401	11.2
fpppp	10525	9800	6.9	9666	8.2
matrix300	429	361	15.9	367	14.5
tomcatv	953	912	4.3	887	6.9

Table 3. Scheduling times in seconds

Benchmark	BB	EBBS	DPS
clean	5.10	6.49	44.54
compress	1.31	1.37	3.26
dfa	0.19	0.24	3.75
dhrystone	0.20	0.24	0.31
fft	0.37	0.47	5.15
go	12.07	20.86	2108.22
jpeg	12.83	14.85	41.29
nsieve	0.08	0.08	0.13
water	0.89	1.00	1.75
fmin	0.06	0.06	0.11
rkf45	0.18	0.22	0.57
seval	0.09	0.10	0.15
solve	0.12	0.12	0.31
svd	0.19	0.25	1.54
urand	0.03	0.03	0.04
zeroin	0.04	0.05	0.07
applu	1.84	2.34	7.03
doduc	4.01	5.20	22.24
fpppp	47.93	39.48	55.91
matrix300	0.08	0.08	0.14
tomcatv	0.13	0.15	0.26

4.2 The TI TMS320C62xx Architecture

The Texas Instruments TMS320C62xx chip (which we will refer to as TMS320) is one of the newest fixed point DSP processors [24]. From a scheduling perspective it has several interesting properties. The TMS320 is a VLIW that allows up to eight operations to be initiated in parallel. All eight functional units are pipelined, and most operations have no delay slots. The exceptions are multiplies (two cycles), branches (six cycles), and loads from memory (five cycles). NOPs are inserted into the schedule for cycles where no operations are scheduled to begin. The NOP operation takes one argument specifying the number of idle cycles.

This architecture has a unique way of "packing" operations into an instruction. Operations are always fetched eight at a time. This is called a *fetch packet*. Bit zero of each operation, called the p-bit, specifies the execution grouping of each operation. If the p-bit of an operation o is 1, then operation $o+1$ is executed in parallel with operation o. (*I. e.*, they are started in the same cycle). If the p-bit is 0, then operation $o+1$ begins the cycle after operation o. The operations that execute in parallel are called an *execute packet*. All operations in an execute packet must run on different functional units, and up to eight operations are allowed in a single execute packet. Each fetch packet starts a new execute packet, and execute packets cannot cross fetch packet boundaries. This scheme and the multiple-cycle NOP operation described above, allow the code for this VLIW to be very compact.

We have modified our scheduler to target an architecture that has the salient features of the TMS320. Of course, there is not a one-to-one mapping of ILOC operations to TMS320 operations, but we feel our model highlights most of the interesting features of this architecture from a scheduling perspective. Our model has eight fully pipelined functional units. The integer operations have latencies corresponding to the latencies of the TMS320. Since ILOC has floating point operations and the TMS320 does not, these operations are added to our model. Each floating point operation is executed on a functional unit that executes the corresponding integer operation. Latencies for floating point operations are double those for integer operations. All ILOC intrinsics (cosine, power, square root, *etc.*) have a latency of 20 cycles. The experiments in the last section assumed perfect branch prediction. However, the TMS320 has no mechanism for predicting branches. Thus, every control-flow operation (including an unconditional jump) incurs a five cycle delay to refill the pipeline. We simulate this by adding five cycles to the dynamic instruction count each time a branch, jump, subroutine call, or subroutine return is executed.

Our static instructions counts reflect the TMS320 fetch packet/execute packet scheme. We place as many execute packets as possible in each fetch packet. NOPs in consecutive cycles are treated as one operation, to be consistent with the multiple-cycle NOP on the TMS320. Each basic block begins a new fetch packet.

Table 4 shows the dynamic instruction counts for our TMS320-like architecture. Static instruction counts (*i.e.,* fetch packet counts) are reported in table 5. In dynamic instruction counts, we see improvements over basic block scheduling similar to those seen for the other architecture. On average, EBBS showed a

Table 4. Dynamic Instruction Counts for TMS320

	Basic Block	EBBS		DPS	
Benchmark	Dynamic Insts	Dynamic Insts	% decrease	Dynamic Insts	% decrease
clean	7120834	6809472	4.4	6759302	5.1
compress	6451078	5865494	9.1	5860732	9.2
dfa	1234284	1163901	5.7	1187176	3.8
dhrystone	6310416	6040418	4.3	5940418	5.9
fft	19134370	18881224	1.3	18896041	1.2
go	873164337	826369843	5.4	824064415	5.6
jpeg	59600121	58496774	1.9	58629470	1.6
nsieve	3810982127	3793649623	0.5	3793649623	0.5
water	37216606	34821429	6.4	34575067	7.1
fmin	4363	3852	11.7	3590	17.7
rkf45	621932	561135	9.8	564360	9.3
seval	2515	2443	2.9	2441	2.9
solve	2819	2692	4.5	2674	5.1
svd	12716	12023	5.4	12160	4.4
urand	1504	1487	1.1	1494	0.7
zeroin	3710	3391	8.6	3292	11.3
applu	676079366	658466445	2.6	658542326	2.6
doduc	13510434	12693031	6.1	12111705	10.4
fpppp	57174493	49772531	12.9	49523545	13.4
matrix300	33652694	33362954	0.9	33363754	0.9
tomcatv	291504852	291494375	0.0	265479402	8.9

5.0 per cent improvement over basic block scheduling, and DPS an 6.1 per cent improvement. Since the same number of control-flow operations are executed regardless of the scheduling technique used, these percentages would improve if the architecture could take advantage of branch prediction information.

However, static code sizes increased slightly over all benchmarks by as much as nine per cent, with DPS producing smaller codes than EBBS in 13 out of 21 cases. This degradation is due to the code compaction method described above. Consider a basic block that has eight operations all packed into one instruction. If six of these operations are moved into another block, and the number of instructions in that block is increased by one, the overall length of the code will increase by one instruction. While we have not added any operations to the compiled code, the number of instructions has increased due to the code motion. This shows how effective the TMS320 design is at keeping object code compact. It also highlights the need for improved scheduling techniques to keep the static code size for these architectures small, while still improving runtime performance.

5 Conclusions and Observations

This paper has examined the problem of scheduling instructions without increasing code size. We looked at two techniques that consider regions larger than a

Table 5. Static Instruction Counts for TMS320

Benchmark	Basic Block	EBBS		DPS	
Benchmark	Static Insts	Static Insts	% increase	Static Insts	% increase
clean	1960	2011	2.6	2001	2.1
compress	306	315	2.9	310	1.3
dfa	345	357	3.5	353	2.3
dhrystone	109	110	0.9	113	3.7
fft	398	411	3.2	406	2.0
go	12780	13328	4.3	13322	4.2
jpeg	3051	3083	1.0	3092	1.3
nsieve	52	54	3.8	54	3.8
water	749	759	1.3	755	0.8
fmin	61	65	6.6	64	4.9
rkf45	187	190	1.6	188	0.5
seval	82	86	4.9	85	3.7
solve	122	126	3.3	125	2.5
svd	258	262	1.6	259	0.4
urand	23	25	8.7	23	0.0
zeroin	40	42	5.0	43	7.5
applu	1943	1961	0.9	1966	1.2
doduc	3768	4015	6.6	3898	3.5
fpppp	1560	1573	0.8	1576	1.0
matrix300	77	81	5.2	81	5.2
tomcatv	127	134	5.5	134	5.5

single basic block, but do not replicate code. We compared the performance of these two methods against that of list scheduling over single basic blocks. We reformulated the safety conditions for DPS to avoid problems that arose in our implementation of the algorithm.

1. Both techniques improved on the single block list scheduling by about seven percent. DPS produced better results, on whole, than EBBS. This may be due to the fact that DPS generated larger regions for scheduling.
2. Both EBBS and DPS required more compile time than list scheduling. EBBS was reasonably competitive with list scheduling, taking up to thirty percent longer. DPS required much more time–the worst case slowdown was two orders of magnitude. This suggests that better implementation techniques are needed for DPS.
3. On machines that require the compiler to insert NOPs for correctness, the improvement in running time may lead to a decrease in code size. Our measurements showed that this averaged roughly eleven percent for the codes used in our study. The experiments with the TMS320 showed negative results for code size; that machine's hardware strategy for achieving compact instructions makes the arithmetic of compiler-based code compaction very complex.

Taken together, these findings suggest that, even in a memory constrained environment, non-local scheduling methods can achieve significant speedups compared to a purely local approach. For machines that require NOPs, the accompanying reduction in code size may be important.

This study suggests two directions for future study.

- Techniques that quickly generate larger acyclic regions may lead to further reductions in running time (and code space), even when code growth is prohibited. These warrant investigation.
- A more efficient implementation of DPS is needed. This may be a matter of engineering; on the other hand, it may require some significant re-thinking of the underlying algorithms.

If code size is an issue, these techniques deserve consideration. In fact, the compiler writer should consider using EBB as the baseline scheduling technique, and using a best-of-several approach for final, production compiles. This approach has proved profitable on other problems [3]; Huber has recommended it for finding the best dominator path.

6 Acknowledgements

The scheduler was implemented inside the experimental compiler built by the Massively Scalar Compiler Group at Rice; the many people who have contributed to that effort deserve our heartfelt thanks. Also, thanks to Phil Sweany for his pointer to Brett Huber's work, and to the referees for their helpful comments.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compiler: Principles, Techniques, and Tools.* Addison Wesley, 1986.
2. David Bernstein, Doron Cohen, and Hugo Krawczyk. Code duplication: An assist for global instruction scheduling. *SIGMICRO Newsletter*, 22(12):103–113, December 1991. *Proceedings of the 24th Annual International Symposium on Microarchitecture.*
3. David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*
4. David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. *SIGPLAN Notices*, 26(6):241–255, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*
5. Preston Briggs. The massively scalar compiler project. Technical report, Rice University, July 1994.
6. Kemal Ebcioglu and Toshio Nakatani. A new compilation technique for parallelizing regions with unpredictable branches on a VLIW architecture. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, August 1989.

7. John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, 1986.
8. Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.
9. G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.
10. Stefan Freudenberger, Thomas R. Gross, and P. Geoffrey Lowney. Avoidance and supression of compensation code in a trace scheduling compiler. *ACM Transactions on Programming Languages and Systems*, 16(4):1156–1214, July 1994.
11. Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Notices*, 21(7):11–16, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*.
12. Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.
13. Brett L. Huber. Path-selection heuristics for dominator-path scheduling. Master's thesis, Computer Science Department, Michigan Technological University, Houghton, Michigan, 1995.
14. Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscaler compilation. *Journal of Supercomputing – Special Issue*, 7:229–248, July 1993.
15. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *SIGPLAN Notices*, 27(7):224–234, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.
16. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *SIGPLAN Notices*, 29(6):147–158, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.
17. David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, pages 261–294, September 1980.
18. P. Geoffrey Lowney, Stephen M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, Robert P. Nix, J. S. O'Donnell, and J. C. Ruttenburg. The Multiflow trace scheduling compiler. *Journal of Supercomputing – Special Issue*, 7:51–142, July 1993.
19. R.T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pages 133–138. Spartan Books, NY, USA, December 1959.
20. B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *Journal of Supercomputing – Special Issue*, 7:9–50, July 1993.
21. Philip H. Sweany. *Inter-Block Code Motion without Copies*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1992.
22. Philip H. Sweany and Steven J. Beaty. Dominator-path scheduling – A global scheduling method. *SIGMICRO Newsletter*, 23(12):260–263, December 1992. *Proceedings of the 25th Annual International Symposium on Microarchitecture*.
23. Robert E. Tarjan and John H. Reif. Symbolic program analysis in almost-linear time. *SIAM Journal on Computing*, 11(1):81–93, February 1982.
24. Texas Instruments. *TMS320C62xx CPU and Instruction Set Reference Guide*, January 1997. Literature Number: SPRU189A.