Boosting Beyond Static Scheduling in a Superscalar Processor

Michael D. Smith, Monica S. Lam, and Mark A. Horowitz
Computer Systems Laboratory
Stanford University, Stanford CA 94305-4070

May 1990

1 Introduction

Superscalar processors are uniprocessor organizations capable of increasing machine performance by executing multiple scalar instructions in parallel. Since the amount of instruction-level parallelism within a basic block is small, superscalar processors must look across basic block boundaries to increase performance. In numerical code, the do-loop branches, which are a large percentage of the total branches executed, can be resolved early to expose the parallelism between many iterations. Unfortunately, many of the branches in non-numerical code are data dependent and cannot be resolved early. Thus, speculative execution, the execution of operations before previous branches, is an important source of parallelism in this type of code.

Instruction-level parallelism can be extracted statically (at compile-time) or dynamically (at run-time). Statically-scheduled superscalar processors and Very Long Instruction Word (VLIW) machines exploit instruction-level parallelism with a modest amount of hardware by exposing the machine's parallel architecture in the instruction set. For numerical applications, where branches can be determined early, compilers harness the parallelism across basic blocks by techniques such as software pipelining [11] or trace scheduling [7]. However, the overhead and complexity of speculative computation in compilers has prevented efficient parallelization of non-numerical code.

Dynamically-scheduled superscalar processors, on the other hand, effectively support speculative execution in hardware. By using simple buffers, these processors can efficiently commit or discard the side effects of speculative computations. Unfortunately, the additional hardware necessary to look far ahead in the dynamic instruction stream, find independent operations, and schedule these independent operations out of order is costly and complex.

We are interested in using superscalar techniques to increase the performance of non-numerical code at a reasonable cost. To accomplish this goal, we propose a superscalar architecture, which we call *TORCH*, that combines the strengths of static and dynamic instruction scheduling. The strength of static scheduling is the compiler's ability to efficiently schedule operations across many basic blocks; consequently, TORCH performs all instruction scheduling in the compiler. The strength of dynamic scheduling is in its ability to efficiently support speculative execution; consequently, TORCH provides hardware that allows the compiler to schedule any instruction before preceding branches, an operation we term *boosting*. Boosted instructions are conditionally committed upon the result of later branch instructions. Boosting, therefore, removes the scheduling constraints that result from the dependences caused by conditional branches and makes aggressive instruction scheduling in the compiler simple.

To make the conditional evaluation of boosted instructions efficient at run-time, the TORCH hardware includes two shadow structures: the *shadow register file* and *shadow store buffer*. These structures buffer the side effects of a boosted instruction until its dependent branch conditions are determined. On a correctly predicted branch, the hardware commits the appropriate values in the shadow structures. On a mispredicted branch, the hardware guarantees correct program operation by squashing all shadow values.

In this paper, we overview the TORCH architecture, describe the TORCH hardware to support boosting, and present the results of a simple static scheduler which performed limited instruction boosting and no load/store reorganization. The simple scheduler and our evaluation system allow us to quickly assess the viability of

boosting on non-numerical programs. The results of this evaluation are supportive and indicate that TORCH, with only a subset of its full functionality, can approach the performance level of a dynamically-scheduled superscalar processor. As a result, we are proceeding to implement a compiler and a full simulator for TORCH.

The next section covers in more detail how different schedulers handle the issues involved in scheduling across conditional branches. Section 3 overviews the organization and operation of TORCH. Section 4 presents the specifics of the evaluation system, the limitations on the instruction schedulers, and the details of the dynamically-scheduled superscalar machine used in our evaluations. Section 5 reports on the results of the experiments, and the final section presents the conclusions of the study.

2 Scheduling Across Branches

Instruction-level parallelism within a basic block or extended basic block can be easily exploited by providing an adequate number of functional units and by limiting the effects of storage conflicts [1] in the code. However, the amount of instruction-level parallelism within a basic block is limited. An effective superscalar machine must move instructions across basic block boundaries to find larger amounts of concurrency.

2.1 Dynamic Scheduling

Dynamically-scheduled superscalar processors support code motion across basic block boundaries by looking ahead in the instruction stream, by buffering the internal state, and by executing instructions conditionally and out of order. These techniques were first utilized in the IBM Stretch [2] and IBM 360/91 [22]. These machines attempted to increase processor performance by hiding memory latency and decoupling instruction fetch from instruction execution. They provided internal buffering for issued but unexecuted instructions and extra state to keep track of the out-of-order results. Though these machines only fetched and decoded a single instruction per cycle, later machines incorporated multiple-instruction fetch and decode to further enhance machine performance (Johnson provides a good overview of these machines [9]). These investigations culminate in today's dynamically-scheduled superscalar designs which try to isolate instruction fetch/decode from instruction issue/execute to allow each to run at its own pace [8, 14, 16, 20].

Multiple instruction execution occurs when the hardware issues independent instructions from a window of dynamic instructions. To maintain scalar code compatibility, all instruction scheduling is done from this window by the hardware. Out-of-order execution and branch prediction provide these machines with the ability to simultaneously execute instructions from multiple basic blocks. Buffering within the processor supports the conditional evaluation of instructions that are executed before previously fetched branches, increasing the opportunities for the hardware to find instructions to issue in parallel.

Unfortunately, dynamic scheduling has many shortcomings. All of these shortcomings result from the detection of parallelism and the scheduling of independent operations at run time. First of all, the detection and scheduling of independent operations by the hardware increases its complexity, possibly lengthening the cycle time of the machine and reducing the actual speedup over the scalar processor. Another effect of using hardware to detect instruction-level parallelism is that the hardware can only analyze a small window of dynamic instructions during each cycle, thus limiting the possible candidates for parallel issue. Finally, *instruction fetch efficiency*, defined in Smith et al. [21] as the average number of useful instructions fetched per cycle, is reduced when executing from scalar object code. As a result of the large number of branches during the execution of non-numerical code, dynamic schedulers suffer a significant performance penalty due to branch point misalignment in a fetch block.

2.2 Static Scheduling

Statically-scheduled machines overcome the run-time scheduling problems of dynamically-scheduled machines by making the parallelism explicit in the instruction set architecture and depending upon the compiler to identify and schedule all instruction-level parallelism in a program. The compiler explicitly specifies which instructions

[1] Storage conflicts are anti- and output data dependences caused by the reuse of registers or memory locations.

are issued and executed in parallel. There is no overhead during run-time to schedule instructions, and the hardware is simple. The compiler essentially has a infinite instruction window and uses global program knowledge, program semantics (dependences), and resource constraints in constructing each instruction schedule. Finally, instruction alignment is not an issue in static scheduling since the compiler schedules instructions in fetch blocks.

The basis for statically-scheduled superscalar processors comes from the field of horizontally-microcoded machines and Very Long Instruction Word (VLIW) architectures. Early on, VLIW machines were similar to horizontally-coded microcode in that each instruction word had a field for each independent functional unit [3]. More recent VLIW machines [5] remove this restriction by providing dynamic NOPs for idle functional units. In fact, the distinction between statically-scheduled superscalar processors and VLIW machines is blurry. Both machines rely on the compiler to explicitly specify the instruction-level parallelism and manage the hardware resources. The difference between statically-scheduled superscalar processors and VLIW machines is basically one of terminology. VLIW machines refer to operations within a singly-fetched instruction, while statically-scheduled superscalar processors refer to instructions within a single fetch block. A VLIW operation is equivalent to a superscalar instruction since both control a single functional unit. Many statically-scheduled machines have been announced either as superscalar processors [1, 17] or as VLIW processors [4].

Unfortunately, all these machines encounter difficulties when scheduling across conditional branches even though software branch prediction can be as accurate as hardware branch prediction [13]. Delay-branch schedulers are able to perform some limited movement of instructions across basic block boundaries. The compiler either moves instructions down from within a basic block to fill the branch delay slot; the branch must not depend upon these instructions so that it does not matter whether the machine executes them before or after the branch. The other way to fill branch delay slots is to lift an instruction up from the fall-through or target basic block. These instructions must not have any side effects so that their execution does not affect the machine state if the branch goes the other direction. Though these schemes are able to effectively fill a single branch slot, the effectiveness of these motions drops off significantly as the number of branch slots increases. For instance, a compiler can fill approximately 70% of single branch slots, but only 25% of double branch slots [13]. As we can see from the percentages of multiple branch slots filled, these reorganizing schemes are very limited in their ability to move instructions across conditional branches.

Trace scheduling [7] is a compiler technique that was designed to increase the compiler's ability to move instructions across basic block boundaries. It utilizes branch predictions to create a single long block of code out of individual basic blocks without hardware support. A greater level of concurrency is achieved by scheduling this long block instead of the individual basic blocks, but at the expense of fix-up code at the entry and exit points of the trace. The difficulty and overhead of saving and restoring state for instructions with side effects and of finding a few major traces in non-numerical code with its large number of run-time branches makes trace scheduling an unattractive choice.

Other statically-scheduled machines reduce the effects of conditional branches by scheduling both paths of a conditional branch in parallel [18]. This scheme requires hardware support to NOP the instructions along the non-taken branch path. Though this technique allows for some overlap between the schedules of the basic blocks before and after the conditional statement with the combined branches of the conditional statement, the compiler's ability to move instructions with side-effects is not improved.

2.3 Scheduling in TORCH

Boosting combines into one architecture the best aspects of static and dynamic scheduling to overcome the difficulties of scheduling instructions across conditional branches. Boosting relies on the compiler's knowledge of the program semantics and ability to explore many schedules to find the best instruction schedule. Boosting simplifies scheduling by assigning the hardware the responsibility of handling side effects. To the scheduler, all boosted instructions appear free from side effects.

The hardware efficiently handles boosting by providing extra buffering in the register file and store buffer. The shadow structures hold the results of boosted instructions until they are committed or squashed. Efficiency is guaranteed by performing the commit and squash operations without any performance penalty. For further efficiency, the hardware postpones all exception processing on boosted instructions until the hardware tries to commit the boosted instructions. Exceptions are precise and easy to identify.

3 TORCH

Before expanding on the implementation and operation of boosting in TORCH, we briefly overview the relevant aspects of the TORCH architecture. First of all, our current implementation of TORCH is based upon the MIPS R2000 RISC architecture[2]. We use the R2000 processor because the implementation of superscalar techniques is simpler on a load/store machine with fixed length instructions than on a memory-to-memory machine with variable length instructions.

Like the R2000, TORCH uses delayed branches to eliminate the delay between executing the branch and fetching the first instruction of the target branch. Furthermore, like other scalar processors that contain squashing branches, TORCH encodes static branch prediction information into each condition branch instruction. The branch prediction used in TORCH is based on branch profile statistics, since the accuracy of the branch prediction directly impacts the performance gain. Profiling indicates the most-likely branch path, and TORCH boosts instructions from this path to overlap instructions from different basic blocks.

Storage conflicts and memory disambiguation also constrain instruction scheduling and boosting. TORCH minimizes the effects of storage conflicts by considering register allocation during instruction scheduling. Run-time register renaming is not needed since TORCH issues instructions in order.

Successful memory disambiguation removes constraints on instruction scheduling by permitting the reordering of load and store operations. Compilers can disambiguate some memory references, but not to the extent of the hardware during run-time. Though not discussed further in this paper, the boosting mechanism in TORCH provides a framework for the TORCH compiler to perform speculative reordering of load and store instructions that cannot be completely disambiguated. This reordering by the compiler is supported by similar buffering and checking hardware as is found in dynamically-scheduled superscalar processors.

3.1 An ISA for Boosting

TORCH instructions are basically identical in function to the scalar instructions. The TORCH instructions are encoded differently, however, to include bits that specify boosting information. With a single level of boosting, one bit in a TORCH instruction encodes whether or not the instruction is boosted. If the bit is set, the instruction depends upon the outcome of the next conditional branch. The prediction is encoded in the branch instruction itself. With multiple levels of boosting, multiple bits are needed to indicate the level of boosting, i.e. the number of later branches upon which this instruction depends. The following discussion only describes the architecture and hardware necessary to support boosting through a single conditional branch since this produces the largest incremental gain in performance.

TORCH machine instructions are able to address all the locations in both the sequential and shadow register files. Each sequential register location has a dual location in the shadow register file. A single shadow register file is sufficient to buffer all side effects, since we only boost through one conditional branch. For instance, if we boost an instruction that writes into register `r3`, the boosted instruction would specify shadow register `r3` as its destination. If the conditional branch that this boosted instruction depends upon is correctly predicted, the value in shadow register `r3` is written into sequential register `r3`, maintaining sequential semantics. An incorrect prediction would cause shadow register `r3` to be invalidated.

As mentioned above, TORCH supports precise exceptions in a clean and efficient manner. Exceptions on sequential instructions are handled immediately, while exceptions on boosted instructions are postponed until the boosted instruction attempts to commit. In this way, exceptions on boosted instructions that never commit do not alter the semantics of the program or degrade machine performance. When TORCH executes a conditional branch that tries to commit an outstanding exception on a boosted instruction, it invalidates the shadow structures and reexecutes all the boosted instructions that depend upon this branch. Since the branch that these "boosted" instructions depend upon is now resolved, the instructions are now sequential, and a sequential interrupt occurs at the precise time.

[2] R2000 is a Trademark of MIPS Computer Systems, Inc.

3.2 Small Example

Figures 1, 2, and 3 show a small piece of C code with its MIPS R2000[3] and TORCH machine code equivalents. Both the R2000 and TORCH have a single load delay slot, and this particular implementation of TORCH issues a maximum of two instructions per cycle. Boosted instructions are indicated by a `.b` suffix on the opcode, and accesses to registers in shadow register file are indicated by a `.s` suffix on the particular register specifier. (We use symbolic names instead of register identifiers simply for clarity.) A predicted-taken branch is indicated by a `.t` suffix and a not-taken prediction is indicated by a `.n` suffix.

```
register int cnt = 0;
while (ptr) {
    if (ptr->data > 1000) cnt++;
    ptr = ptr->next;
}
```

Figure 1: Example in C

```
(1)           beq    ptr,0,lab1
(2)           move   cnt,0
(3)    lab3:  lw     data,0(ptr)
(4)           nop
(5)           slti   temp1,data,1001
(6)           bne    temp1,0,lab2
(7)           nop
(8)           addiu  cnt,cnt,1
(9)    lab2:  lw     ptr,4(ptr)
(10)          nop
(11)          bne    ptr,0,lab3
(12)          nop
(13)   lab1:
```

Figure 2: MIPS R2000 Machine Code for C Example

```
(1)           beq.n  ptr,0,lab1          ; lw.b    data.s,0(ptr)
(2)           move   cnt,0               ; nop
(3)           slti   temp1,data,1001     ; nop
(4)    lab3': bne.n  temp1,0,lab2'       ; addiu.b cnt.s,cnt,1
(5)           lw     ptr,4(ptr)          ; nop
(6)    lab2': nop                        ; lw.b    data.s,0(ptr)
(7)           bne.t  ptr,0,lab3'         ; nop
(8)           slti.b temp1.s,data.s,1001 ; nop
(9)    lab1:
```

Figure 3: TORCH Machine code for C Example

The example code simply walks a linked list, updating a count variable depending upon the value of the data at each element. The code contains four basic blocks with nearly no intra-basic-block parallelism. If we assume that the conditional branch at line 6 of Figure 2 is taken 50% of the time and we ignore the while loop startup, the MIPS R2000 code requires 9.5 cycles on average to execute the loop. With boosting, we can completely

[3]This code was generated with the maximum optimization level.

hide the update of variable `cnt` and overlap the critical paths of the two inner loop basic blocks. Thus, the while loop only takes an average of 5 cycles to execute on TORCH, just over half the time of the MIPS R2000.

3.3 Hardware Organization

Figure 4 illustrates the overall structure of TORCH. TORCH consists of two operational units, one for integer or logical operations and one for floating-point operations. Within each of these operational units, there exists a variety of independent functional units, register files, and other support hardware. The degree of parallelism in the machine dictates the number of component interconnections. For instance, the TORCH implementation in the previous example supports a degree of parallelism of two, and thus, two buses would connect the functional units with the register file for the transport of results.

The TORCH hardware differentiates itself from other statically-scheduled superscalar processors by including two shadow structures: the shadow register file and the shadow store buffer. The register file contains both a sequential register file and a shadow register file; similarly, the store buffer contains both a sequential store buffer and a shadow store buffer. There is one sequential/shadow register file for the integer unit and one sequential/shadow register file for the floating-point unit. There is only a single sequential/shadow store buffer for the entire processor. The sequential structures contain the sequential state of the machine; they do not contain the results of any instructions executed speculatively. These results are kept in the shadow structures.

The purpose of the shadow structures then is to hold the results of boosted instructions until the conditional branch upon which these boosted instructions depend are resolved. If a conditional branch that a set of boosted results depend upon is executed and found to be predicted correctly, this set of results is copied in mass from the shadow structures to the sequential structures. When a conditional branch executes differently from its prediction, all values in the shadow structures are thrown away since they were boosted assuming the branch would go the way of its prediction.

Similarly, boosted instructions that are in the pipeline when the condition of a branch is determined are also updated. On a correct prediction, the destinations of the boosted instructions in the pipeline are changed from the shadow structures to the sequential structures. On an incorrect prediction, all boosted instructions in the pipeline are squashed.

4 Evaluation Methodology

To measure the effectiveness of boosting and ultimately the viability of the TORCH architecture, we compare its performance against that of an aggressive dynamic scheduler on non-numerical programs.

4.1 MATCH

This section briefly overviews the organization and operation of a dynamically-scheduled superscalar processor that we used for our comparisons. We obtained the dynamically-scheduled model directly from a study by Johnson [9]. Though the simulator for his study allows for many different dynamically-scheduled superscalar organizations, we limit ourselves to using the basic processor model. This model, which we call MATCH[4], is ambitious in its attempts to exploit instruction-level parallelism at the expense of complex and costly hardware. MATCH is able to execute scalar object code and thus makes no changes to the instruction set architecture.

Figure 5 illustrates the overall structure of MATCH. At first glance, the structure is very similar to TORCH; MATCH still contains two operational units, each with independent functional units. MATCH does not contain shadow structures, but as explained below, its reorder buffer performs a similar function. Since MATCH performs all instruction scheduling in hardware, most of the complexity of the hardware is in the control logic and is not shown in the figure.

In front of each functional unit is a reservation station [22]. The reservation stations are instruction buffers that disassociate the actual instruction fetch rate from the instruction execution rate. With this buffering, MATCH

[4]Though Johnson did not name his models, we call one of his models MATCH because TORCH attempts to "match" its performance and because the ideas for TORCH grew out of our work on MATCH.

only needs enough hardware in the decoder, register file ports, and buses to support the average instruction-execution rate. These reservation stations also allow the processor to execute instructions out of order even though the instructions are fetched and decoded in program order. Therefore, it is these reservations stations and their control logic that perform the dynamic scheduling of instructions in MATCH.

To provide more opportunities for parallel issue, MATCH allows concurrent issue of a load and a store instruction. The load and store functional units are simply buffers for the memory instructions. The buffers also contain logic to perform memory disambiguation at run-time, thereby allowing loads and stores to bypass each other when advantageous.

Storage conflicts in the original code are eliminated at run-time by performing register renaming in the hardware [10]. Register renaming is implemented by using a reorder buffer [19] associated with each register file. The reorder buffer provides the additional storage necessary to implement register renaming. For example, when an instruction is decoded, MATCH dynamically allocates a location in the reorder buffer for this instruction's result and the instruction's destination-register number is associated with this new location. The next instruction that tries to fetch this register number as an operand will receive the value in the reorder buffer since this location contains the latest value. In effect, the register is renamed.

Furthermore, the reorder buffer allows MATCH to execute instructions across outstanding conditional branches by providing storage for the uncommitted results. MATCH allocates and deallocates locations in the reorder buffer as a FIFO queue. When the instruction at the head of the queue completes and writes its result into the reorder buffer, MATCH proceeds to write that value back to the register file. Since MATCH decodes instructions and allocates locations in the reorder buffer in program order, MATCH maintains updates to the register file in program order. Thus, when a conditional branch instruction reaches the head of the reorder buffer queue, the only values committed to the register file are those that had no dependence upon the conditional branch since they were fetched and decoded before the branch. If a branch is mispredicted, MATCH simply invalidates all locations in the reorder buffer from the point of the branch backwards, and then starts executing instructions from the correct branch target. As a result, the reorder buffer provides an easy mechanism to eliminate storage conflicts and bypass conditional branches.

To provide conditional branch prediction, MATCH relies on a Branch Target Buffer (BTB) [12] which is incorporated into the instruction cache design. This structure improves the instruction fetch efficiency by removing the need to wait for a branch condition determination. Unfortunately, MATCH cannot eliminate the penalty to fetch efficiency due to instruction misalignment. Even the addition of alignment hardware in the fetch does little to reduce this penalty since the fetcher cannot stay far enough ahead of the decoder [9].

4.2 Evaluation Tools

Using the *pixie* [15] trace facility on the MIPS machines to generate a dynamic instruction trace for a particular program, Johnson's simulator for MATCH analyzes each dynamic instruction in turn. It keeps track of the functional requirements and result latencies of each dynamic instruction to determine the cycle count of this particular program on MATCH. A more indepth description of Johnson's simulator can be found in [9].

To quickly evaluate the viability of boosting, we built a simple scheduler that implements only a small subset of the functionality of the TORCH compiler. Furthermore, this simple scheduler works on optimized object code and therefore does not have access to higher-level program information. Since the disambiguation of memory instructions is extremely difficult at the object code level, the simple scheduler maintains the order of stores with respect to other memory operations. A full-blown implementation of a statically-scheduling superscalar compiler could disambiguate some of these memory references.

It is interesting to note that MATCH uses the buffering in its reservation stations to effectively move instructions both up and down in the execution sequence to try and smooth out the resource utilization. For instance, if an instruction in a basic block does not affect the condition of the branch instruction at the end of the basic block, this instruction can be executed after the branch instruction. The dynamic scheduler effectively moves this instruction down. A TORCH compiler could perform this same movement, but it is not done in the current simple scheduler. The simple scheduler only boosts instructions up through a single conditional branch and *reorganizes* instructions up through a single unconditional branch.

The simple scheduler compacts static basic blocks only individually or in pairs. Each basic block is list

scheduled [6] given the current hardware constraints (the number of functional units and the number of instructions in a fetch block, for instance) and the program semantics. The scheduling between basic block pairs is accomplished by first list scheduling the dynamically-preceding basic block and then list scheduling the following basic block into the first schedule as if the two basic blocks were a single large basic block. The combination of the lengths of the individual basic block schedules minus the length of this new schedule is the number of cycles saved by boosting or reorganization. Reorganization always reduces the execution by these cycles; boosting reduces the execution only when the condition branch goes the way of the boost.

The evaluation process begins by running the limited scheduler over the object code for a program. The scheduler calculates the new basic block lengths of a program for a TORCH machine with a specified hardware configuration, a single boosting level, and no memory disambiguation. This data is combined with the basic block execution counts and the profile data for conditional branches to determine the final program cycle count. This calculation is correct since the simple scheduler does not change the number of basic blocks or reorder the conditional branches.

Both evaluation systems use the same machine configuration file. This file contains information on the number of functional units and the issue/result latencies of each functional unit. The evaluation systems, in addition to determining the performance of the superscalar processor, also determine the performance of a scalar processor that has functional units with the same latencies. Thus, we calculate the speedup of each superscalar processor over the scalar processor, and we ensure proper comparison between superscalar processors by requiring that both scalar simulations execute the same number of instructions in the same number of clock cycles.

4.3 Evaluation Specifics

To simplify the experiments, we restricted the scope of our evaluation systems to deal with only those effects we thought would have the greatest impact on the relative performances. We ran both the TORCH and MATCH evaluations using ideal caches, perfect register renaming, and a small number of functional units. We assumed ideal caches to simplify and speedup our runs. Though real caches will have a definite effect on the relative performances, we believe that caches should affect both machines in similar ways.

We also assumed perfect register renaming to reduce the complexity of the scheduling algorithms. With perfect register renaming, the instruction scheduler only considers true dependences when calculating data dependence constraints. We believe that this is a valid assumption given enough reorder buffer space in MATCH and enough general-purpose registers in TORCH.

Finally, we assumed a small number of functional units since the fetch efficiency and not the number of functional units is the limiting factor when exploiting instruction-level parallelism in non-numerical applications [21]. As a result, we maximize the functional unit cost/performance tradeoff by making the load/store pipe, our most expensive functional unit to duplicate, the most frequently used resource. With one of each functional unit, the integer ALU is the most frequently used functional unit. By adding an extra ALU, the load/store unit becomes the most frequently used functional unit, as desired. Thus, our superscalar machines have two integer ALUs and one of every other type of functional unit.

The functional unit characteristics are listed in Table 1, and they are quite similar to those found in the R2000 processor and its associated floating-point coprocessor. All floating-point result latencies are specified for double-precision operands. The FP Convert unit converts floating-point numbers to integer format and vice-versa. The buffering of each store allows the store issue latency in Table 1 to be a single cycle.

We configured the dynamically-scheduled simulation with enough hardware and buffering in MATCH to guarantee that the dynamic scheduler could effectively use all the functional units. In other words, the effectiveness of the scheduler, the size of the instruction fetch, and the available instruction-level parallelism in the program are the only limitations on the performance of the dynamic scheduler. These three parameters are all basic limitations in the statically-scheduled model too. Thus, we are comparing the ability of the TORCH scheduler against the ability of a dynamic scheduler to execute instructions speculatively before branches.

Table 2 lists our benchmark suite of programs, all of which are written in C. Each of these programs ran to completion on a fairly large input data set, as indicated by the size of the dynamic instruction count in Table 2. None of these programs are floating-point intensive, and each exhibits a high frequency of branches during

Functional Unit	Issue Latency (cycles)	Result Latency (cycles)
Integer ALU	1	1
Barrel Shifter	1	1
Load Pipe	1	2
Store Pipe	1	-
Branch Unit	1	2
FP Adder	1	2
FP Multiplier	1	5
FP Divider	1	19
FP Convert	1	2

Table 1: Functional Units with Issue and Result Latencies

execution. Finally, all of these programs are highly-optimized since we are interested in the true advantage of a superscalar architecture and not in the advantages of executing redundant code in parallel.

Program Name	Static Instrs	Dynamic Instrs	Description
awk	26.9K	60.7M	pattern scanning/processing
ccom	58.1K	19.4M	front-end of a C compiler
espresso	43.6K	340.5M	logic minization
irsim	89.4K	54.3M	simulator for VLSI layouts
latex	53.2K	100.6M	document preparation system

Table 2: Program Descriptions

5 Results

All results in this section report speedups over the base scalar processor. The speedups are reported for each benchmark program along with the harmonic mean of the five benchmark speedups. Of course, we are mostly interested in the relative performance of the TORCH and MATCH processors, but absolute speedups of each superscalar processor over the scalar processor are also interesting. To have a reference point for the absolute speedups, we first present some numbers indicating the amount of exploitable instruction-level parallelism in our benchmark programs given only a few physical constraints.

Performance is always achieved at some cost. For our investigations, we believe that data memory interface is our most expensive resource, and as such we limit our machines to a single load/store pipe. Table 3 presents the maximum theoretical speedups for our benchmark programs if this single load/store pipe is the only constraint on instruction scheduling. These speedups are simply the ratio of the total serial clock cycles divided by the number of dynamic load and store instructions in the program. In other words, executing one load or store operation every cycle increases performance by a factor two to four over the scalar processor.

awk	ccom	espresso	irsim	latex	hm
3.91	3.03	4.19	2.84	2.88	3.28

Table 3: Maximum Theoretical Speedups Based on Load/Store Frequency

Since the scalar processor executes approximately 0.9 instructions per cycle on average due to pipeline stalls, a speedup of 4 corresponds to executing less than 4 instructions per cycle in the superscalar machine. A balanced machine design requires an average fetch rate of four instructions per cycle to keep this execution unit busy in steady state. Of course, this execution rate is a theoretical maximum and the real execution rates will be less; consequently, we limit the TORCH and MATCH superscalar machines to fetching either two or four instructions per cycle.

5.1 Justifying Boosting

Boosting in a statically-scheduled superscalar processor allows the compiler to schedule instructions up through previous conditional branches. Boosting is not necessary if the compiler can find adequate instruction-level parallelism within the basic block. To determine the speedup from scheduling only within a basic block, we ran four different types of experiments on our TORCH machine, all of which did not use TORCH's ability to boost instructions. Table 4 presents these results.

	awk	cccom	espresso	irsim	latex	hm
Fetch 2	1.17	1.11	1.22	1.11	1.19	1.16
Fetch 4	1.17	1.11	1.22	1.12	1.21	1.16
one IPC	1.19	1.23	1.08	1.27	1.13	1.18
Infinite	1.24	1.32	1.35	1.24	1.41	1.31

Table 4: Speedups on TORCH When Scheduling Only Within a Basic Block

The first two experiments limited TORCH to fetching two and four instructions per cycle respectively. These speedups indicate that our non-numerical programs do not contain a large degree of concurrency within the basic block. Since the amount of instruction-level parallelism in a basic block is so low, fetching and executing two instructions concurrently finds the vast majority of parallelism, and increasing the fetch size to four instructions is not advantageous. In fact, the third line of Table 4, which reports the speedups of a machine executing one instruction per cycle, shows that intra-basic-block scheduling corresponds to executing approximately one instruction per cycle.

The final line of Table 4 shows the maximum possible speedups for an infinite, statically-scheduled superscalar machine. This machine has an infinite number of functional units and an infinitely large instruction fetch; its performance is only limited by true data dependences and basic block boundaries. This small amount of instruction-level parallelism, even in an ideal machine, makes a strong argument for boosting and other methods for exploiting the parallelism between basic blocks.

5.2 TORCH Results

In our next experiment, we compared the performance of static scheduling with boosting in TORCH against the performance of dynamic scheduling in MATCH. To isolate the scheduling effects, we made the superscalar models equal in their instruction-fetch size, their handling of branch prediction, and their ability to reorganize load and store operations. For Table 5, both TORCH and MATCH fetch either two or four instructions per cycle, and for this table, both processors predict that all branches are taken. TORCH, in our simple scheduler, is not able to disambiguate memory addresses since we have not implemented a full compiler system. Consequently, for the results in Table 5, we disabled the hardware in MATCH which supports movement of load and store instructions past each other.

When fetching two instructions per cycle in Table 5, static scheduling with boosting outperforms dynamic scheduling. This advantage comes from the compiler's ability to align instructions in memory to eliminate the instruction misalignment effects that are especially detrimental to MATCH with such a small fetch block. The small fetch block also limits the size of the window of instructions from which the dynamic scheduler tries to find concurrent instructions. Though a four-instruction fetch reduces both of these effects as shown by the smaller performance difference in the bottom half of Table 5, TORCH is still faster.

	awk	cccom	espresso	irsim	latex	hm
Fetch 2						
TORCH	1.42	1.37	1.61	1.45	1.44	1.45
MATCH	1.34	1.28	1.45	1.31	1.34	1.34
Fetch 4						
TORCH	1.45	1.41	1.69	1.53	1.49	1.51
MATCH	1.40	1.37	1.58	1.43	1.40	1.43

Table 5: Speedups When Predicting Branches Are Taken and No Load/Store Reorganization Is Done

The TORCH results are a lower bound on the performance of this type of scheduling since the current TORCH scheduler is limited to boosting only through a single branch and reorganizing between dynamically-adjacent basic blocks. As we improve the compiler's ability to boost and reorganize, TORCH could better utilize the two and four instruction fetch, at least up to the limits of the parallelism in the machine or program. For these experiments though, static scheduling with boosting can already outperform dynamic scheduling because static scheduling is able to align instructions in fetch blocks and analyze larger amounts of code.

If we then allow each scheduler to use its best branch prediction technique, the TORCH scheduler outperforms the dynamic scheduler by an even larger amount as seen in Table 6. Table 7 helps explain this result. The experiment in Table 5 assumed that the superscalar machine predicted all branches are taken. Table 7 shows that this results in a 73.2% accuracy rate for all branches, conditional and unconditional. Using branch profiling statistics to statically predict branches in TORCH though, we can increase the accuracy rate to 90.6% for our benchmark programs; and by using a 4-way set associative, 2048-entry branch target buffer in MATCH, we can increase its accuracy rate to 83.4%. Since the accuracy rate in TORCH is higher than the accuracy rate in MATCH, the TORCH scheduler performs even better than the MATCH scheduler.

	awk	cccom	espresso	irsim	latex	hm
Fetch 2						
TORCH	1.49	1.52	1.70	1.55	1.55	1.56
MATCH	1.41	1.41	1.51	1.40	1.42	1.43
Fetch 4						
TORCH	1.52	1.57	1.79	1.64	1.63	1.63
MATCH	1.48	1.50	1.66	1.53	1.49	1.53

Table 6: Speedups When Using Improved Branch Prediction Techniques and No Load/Store Reorganization

	awk	cccom	espresso	irsim	latex	hm
Predict Take	79.3	68.5	77.6	75.6	66.8	73.2
BTB	87.1	80.8	80.0	88.1	81.5	83.4
Profiling	91.0	92.6	86.9	92.1	90.6	90.6

Table 7: Branch Prediction Accuracies (as percentages)

5.3 Future Expectations

To see how reorganizing load and store instructions with respect to each other affects performance, we reran the MATCH simulations from Tables 5 and 6 with the memory disambiguation hardware in MATCH enabled. These results are reported in Tables 8 and 9 respectively. The TORCH numbers in the new tables are identical to the numbers in the original tables since the simple scheduler for TORCH cannot perform memory disambiguation.

	awk	cccom	espresso	irsim	latex	hm
Fetch 2						
TORCH	1.42	1.37	1.61	1.45	1.44	1.45
MATCH	1.54	1.46	1.58	1.51	1.48	1.51
Fetch 4						
TORCH	1.45	1.41	1.69	1.53	1.49	1.51
MATCH	1.73	1.71	1.88	1.73	1.70	1.75

Table 8: Speedups When Predicting Branches Are Taken and Allowing Load/Store Reorganization in MATCH Only

	awk	cccom	espresso	irsim	latex	hm
Fetch 2						
TORCH	1.49	1.52	1.70	1.55	1.55	1.56
MATCH	1.63	1.59	1.62	1.63	1.61	1.62
Fetch 4						
TORCH	1.52	1.57	1.79	1.64	1.63	1.63
MATCH	1.86	1.97	1.97	1.94	1.95	1.94

Table 9: Speedups When Using Improved Branch Prediction Techniques and Allowing Load/Store Reorganization in MATCH Only

With memory disambiguation, MATCH is able to outperform our limited TORCH scheduler, especially when fetching four instructions. Memory disambiguation, then, is important since it provides more opportunities for parallel issue. By providing static memory disambiguation in the compiler and hardware to support the speculative reorganization of the other memory operations, we expect to see similar increases in the performance of TORCH. Till then, we feel that the 1.6-times speedup for TORCH is surprisingly good considering our simple TORCH scheduler, and is impressive given TORCH's relatively inexpensive hardware design.

6 Conclusions

This paper introduces a cost-effective way of boosting the performance of statically-scheduled superscalar processors on non-numerical programs. We change the instruction set architecture of a statically-scheduled machine to include boosted instructions, a mechanism that provides the compiler with an efficient method of expressing speculative evaluation.

Boosting removes the dependences caused by conditional branches and makes the reorganization of side-effect instructions as simple as those without side effects. To support boosting in the hardware, we describe relatively inexpensive shadow structures, the shadow register file and shadow store buffer, that hold the side effects of boosted instructions until the conditional branch that the boosted instructions depend upon is executed. When the conditional branch is executed, its prediction is compared to the actual decision and the boosted instructions are either committed or squashed without run-time overhead.

Our experiments on non-numerical code show that adding a single level of boosting to a statically-scheduled superscalar processor yields a 1.6-times speedup over scalar code. This performance is comparable to the performance of an aggressive, dynamically-scheduled superscalar processor. Consequently, we are working on a compiler and a simulator for the full functionality of TORCH. This system will experiment with the boosting of instructions above multiple conditional branches, the reorganizing of instructions both up and down across multiple basic block boundaries, and the advantages of memory disambiguation.

7 Acknowledgements

Mike Smith was supported in part through Digital Equipment Corporation's contributions to the Center for Integrated Systems at Stanford University. This research was supported in part by DARPA, under contract number N00014-87-K-0828.

We are indebted to Mike Johnson of Advanced Micro Devices, Inc. for the use of his simulator, and to Neil Wilhelm of DEC for his help in the design of the TORCH architecture.

References

[1] Apollo Computer Inc. Marketing Brochure, *The Series 10000 Personal Supercomputer*. Chelmsford, MA, 1988.

[2] W. Buchholz(Editor), *Planning a Computer System: Project Stretch*. McGraw-Hill, 1962.

[3] A.E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family". *Computer* (September 1981), pp. 18-57.

[4] R. Cohn, T. Gross, M. Lam, and P.S. Tseng, "Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor." *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1989), pp. 2-14.

[5] R.P. Colwell et al., "A VLIW Architecture for a Trace Scheduling Compiler." *IEEE Transactions on Computers* (August 1988), pp. 967-979.

[6] S. Davidson, D. Landskov, B. Shriver, and P.W. Mallett, "Some Experiments in Local Microcode Compaction for Horizontal Machines". *IEEE Transactions on Computers* Vol. C-30 (July 1981), pp. 460-477.

[7] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction". *IEEE Transactions on Computers* Vol. C-30 (July 1981), pp. 478-490.

[8] R.D. Groves and R. Oehler, "An IBM Second Generation RISC Processor Architecture." *Proceedings 1989 IEEE International Conference on Computer Design: VLSI in Computers and Processors* (October 1989), pp. 134-137.

[9] M. Johnson, "Super-Scalar Processor Design." Technical Report No. CSL-TR-89-383, Stanford University (June 1989).

[10] R.M. Keller, "Look-Ahead Processors." *Computing Surveys* (December 1975), pp. 177-195.

[11] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines." *Proceedings of ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (June 1988), pp. 318-328.

[12] J.K.F. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design." *IEEE Computer* (January 1984), pp. 6-22.

[13] S. McFarling and J. Hennessy, "Reducing the Cost of Branches". *Proc. 13th Annual Symposium on Computer Architecture* (June 1986), pp. 396-404.

[14] "Metaflow Targets SPARC and 386 with Parallel Architecture." *Microprocessor Report* (December 1988), pp. 6-9.

[15] MIPS Computer Systems, Inc., *MIPS Language Programmer's Guide* (1986).

[16] K. Murakami, N. Irie, M. Kuga, and S. Tomita, "SIMP (Single Instruction stream / Multiple instruction Pipelining): A novel High-Speed Single-Processor Architecture." *Proceedings of the 16th Annual International Symposium on Computer Architecture* (May 1989), pp. 78-85.

[17] T.S. Perry, “Intel’s Secret Is Out.” *IEEE Spectrum* (April 1989), pp. 22-28.

[18] B.R. Rau, D.W.L. Yen, W. Yen, and R.A. Towle, “The Cydra 5 Departmental Supercomputer.” *IEEE Computer* (January 1989), pp. 12-35.

[19] J.E. Smith and A.R. Pleszkun, “Implementation of Precise Interrupts in Pipelined Processors.” *Proceedings of the 12th Annual International Symposium on Computer Architecture* (June 1985), pp. 36-44.

[20] J.E. Smith, “Dynamic Instruction Scheduling and the Astronautics ZS-1.” *IEEE Computer* (July 1989), pp. 21-35.

[21] M.D. Smith, M. Johnson, M.A. Horowitz, “Limits on Multiple Instruction Issue.” *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1989), pp. 290-302.

[22] R.M. Tomasulo, “An Efficient Hardware Algorithm for Exploiting Multiple Arithmetic Units.” *IBM Journal* (January 1967), pp. 25-33.

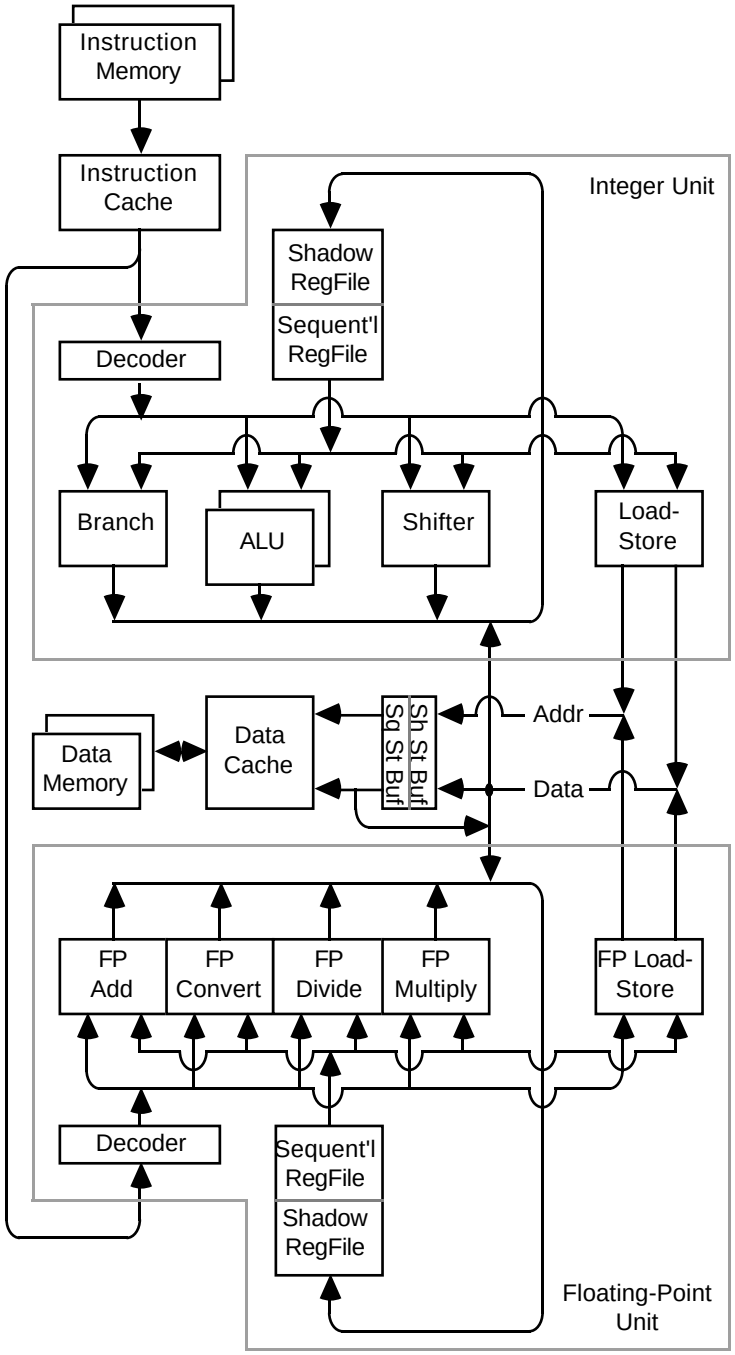

Figure 4: Block Diagram of TORCH

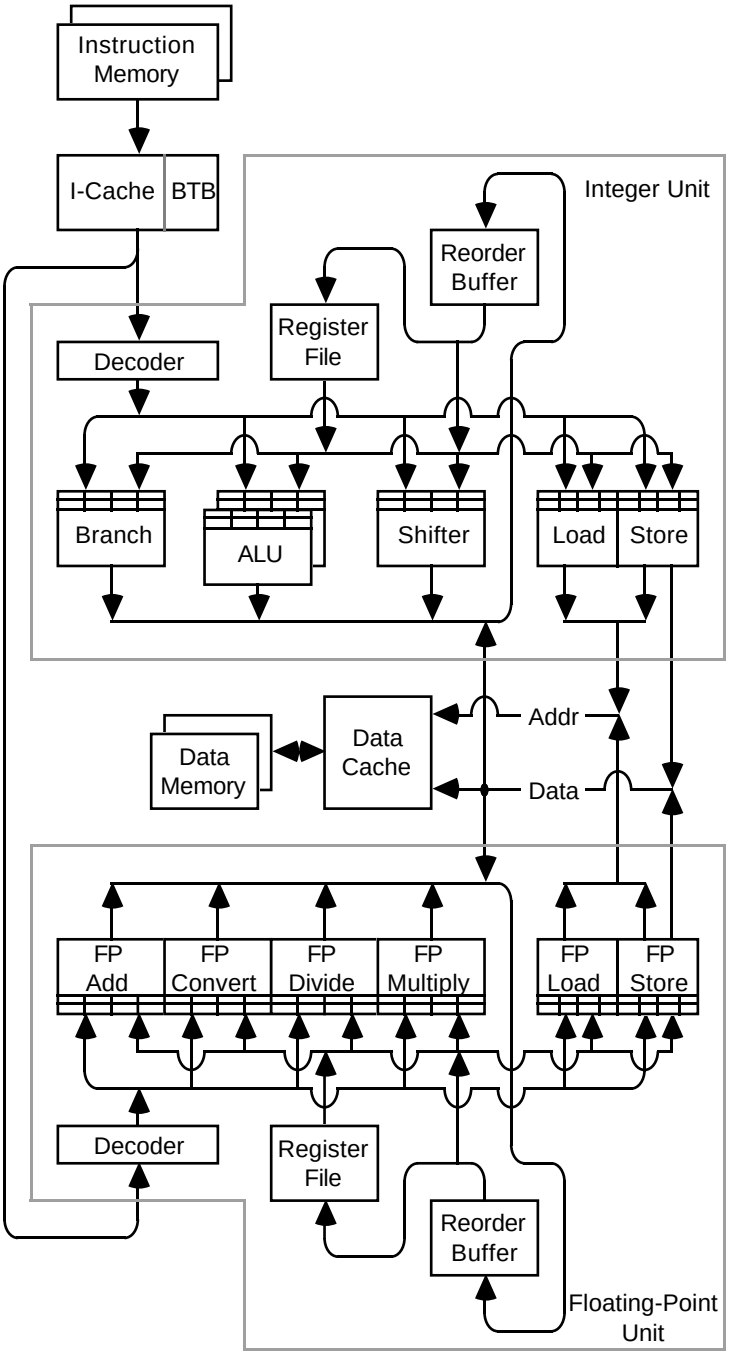

Figure 5: Block Diagram of MATCH