Computer Systems Laboratory
Cornell University, Ithaca, NY 14853

The Branch Processor Architecture

Rajit Manohar and Mark Heinrich

Technical Report No. CSL-TR-1999-1000

November 1999

The Branch Processor Architecture

Rajit Manohar and Mark Heinrich

Computer Systems Laboratory, Cornell University, Ithaca, NY 14853.

Abstract

We present a novel decoupled *branch processor architecture* that separates control-flow information from computational instructions. The control-flow is encoded in an independent instruction stream, potentially providing benefits such as instruction cache prefetching, loop unrolling, code compression, reduced call/return overhead, and improved throughput. We discuss features and tradeoffs in the design and implementation of a branch processor architecture. We present a preliminary instruction set, and describe compilation strategies for the new architecture. We evaluate the architecture by studying metrics that impact its performance using the SPEC95 benchmark suite, and comment on the potential for using branch processor architectures to study control-flow issues in modern processors.

1. Introduction

Branches have long been a source of headaches for designers of pipelined processors. By the time the processor decodes a branch instruction, determines the branch target, and resolves the branch, it has already speculatively fetched several instructions that follow the branch. The deeper the pipeline and the later in the pipeline that branches are resolved, the greater the number of speculative instructions fetched. These additional instructions are either scheduled in a number of branch delay slots or speculatively fetched and end up being killed if the branch is mispredicted. Either method results in significant misprediction penalties and reduced performance.

Modern processors [5,11,28] employ several architectural techniques and devote significant hardware resources to reduce the number of branch-related stalls in the pipeline, including dynamic branch prediction, branch-target buffers, and return-address prediction [16,22,30]. Still, these methods rely on prediction heuristics, cache locality, and guesses to construct the proper program counter (PC) control sequence for the executed program. The dependence of the next PC on data values being produced by previous instructions creates a feedback loop in the processor that begins with instruction fetch and extends through the the pipeline stage where the branch resolves and selects the PC for the next instruction fetch. This fetch loop can negatively impact the throughput of the processor, and in fact was the critical path in the asynchronous MIPS processor from Caltech [15]. In synchronous designs with high clock rates this feedback loop may also affect the clock cycle time—a much more serious threat to performance than even large branch misprediction penalties.

We propose a new *branch processor architecture* that addresses these issues by removing the branches from the processor code stream and breaking the feedback loop between the ALU and instruction fetch. The branch processor architecture embeds a special-purpose *branch processor* in the microprocessor and decouples control-flow information from the stream of "useful" instructions. The branch processor has a small, independent instruction set with its own instruction cache, and is responsible for supplying the *data processor* with a stream of PC values corresponding to computational instructions. The data processor has the ability to communicate data results to the branch processor via a control queue. Applications are compiled into two separate code streams, one for the branch processor that specifies all program control-flow information, and one for the data processor that is void of all control information and contains only computational instructions.

Because of its decoupled nature and its ability to specify entire blocks of PCs with a single instruction, the branch processor has the ability to "run ahead" of the data processor (especially in asynchronous designs). The goal of the branch processor is to keep the queue between it and the data processor full of PCs. Because these are not predicted PC values, the data processor will not fetch any useless instructions. In fact, as we will see in Section 2, the data processor may even fetch and execute fewer instructions once the branch processor removes branches, procedure calls and returns, and loop control information from the data processor instruction stream.

The branch processor architecture has many potential advantages over traditional architectures including automatic loop unrolling, code compression, instruction cache prefetching, code inlining, and reduced call/return overhead. In addition, the branch processor architecture is orthogonal to traditional branch prediction techniques and may be used in combination with these techniques if warranted. These advantages combined with the potential throughput improvement cited above make the branch processor architecture an intriguing new technique in the war on branches for both asynchronous and synchronous processor designs.

We present the branch processor architecture in Section 2, along with a discussion of its potential advantages and disadvantages. We then propose an instruction set architecture (ISA) and outline compilation strategies in Section 3. Section 4 shows the results of a preliminary evaluation of the architecture using both analysis and simulation. We discuss related work in Section 5 and conclude with a summary and suggestions for future research in Section 6.

2. The Branch Processor Architecture

The vast majority of instructions in a traditional processor can be divided into two categories: computational, and control-flow. Branches and jumps are categorized as control-flow instructions since they can modify the program counter. Arithmetic instructions such as `add`, `sub`, and `mult` operate on registers and are thought of as computational instructions.

However, on closer inspection, one is forced to conclude that *every instruction* in a traditional processor is a control-flow instruction. An `add` instruction that adds two registers and stores the result in a third register *implicitly* encodes the fact that the next program counter to be executed is the program counter of the instruction plus the size of the instruction. The left half of Figure 1 shows the overall architecture of an asynchronous MIPS processor [15]. The **fetch** computes the next program counter and sends it to the **icache**, which fetches the next instruction and sends it to the **decode**. At this point the decode determines whether the instruction is a branch or not, and sends this information back to the **fetch**. This cyclic data-dependency of the **fetch** on the instruction that has just been fetched is a direct consequence of the fact that each instruction affects the control-flow. The latency of this fetch loop adversely affects branch performance, and is commonly combated by introducing a predicting branch-target buffer [12,16]. The MIPS R12000 uses such a structure to avoid the full cycle stall on a correctly predicted taken branch that was present in the MIPS R10000 [27]. The fetch loop latency goes up dramatically on an instruction cache miss, since the next level of the memory hierarchy is typically several cycles away.

The branch processor architecture eliminates this deficiency in modern instruction sets by encoding control-flow information in an independent instruction stream executed by a branch processor that computes the sequence of program counter values for the original data processor. A program is therefore compiled into two instruction streams: one that determines the computation to be performed, and one that determines the control-flow. The branch processor is connected to the data processor by two queues. The *instruction queue* contains the sequence of instructions computed by the branch processor, and the *control queue* contains feedback from the data processor required for data-dependent branches, as shown in Figure 1.

The architectural features and instruction set of the branch processor are tailored to support efficient execution of control-flow instructions. Since the primary purpose of the branch processor is to fetch instructions from the **icache**, the basic operation performed by the branch processor is a *block fetch*. This operation fetches a sequential block of instructions starting at a specified address, and inserts it into the instruction queue. The instruction queue and PC queue shown in Figure 1 decouple the execution of the branch processor from **icache** access. The branch processor can continue processing additional control-flow instructions while the fetched instructions are executed by the data processor. This decoupled nature of the branch and data processor makes it possible for the branch processor to "run ahead" of the data processor.

The branch processor can determine control-flow behavior for loops and function calls and returns without information from the data processor. The branch processor includes a hardware stack that keeps track of structured control flow information such as loop nests and return addresses for nested function calls. For example, consider the following loop:

```
      for(i=0;i<10;i++) {
```

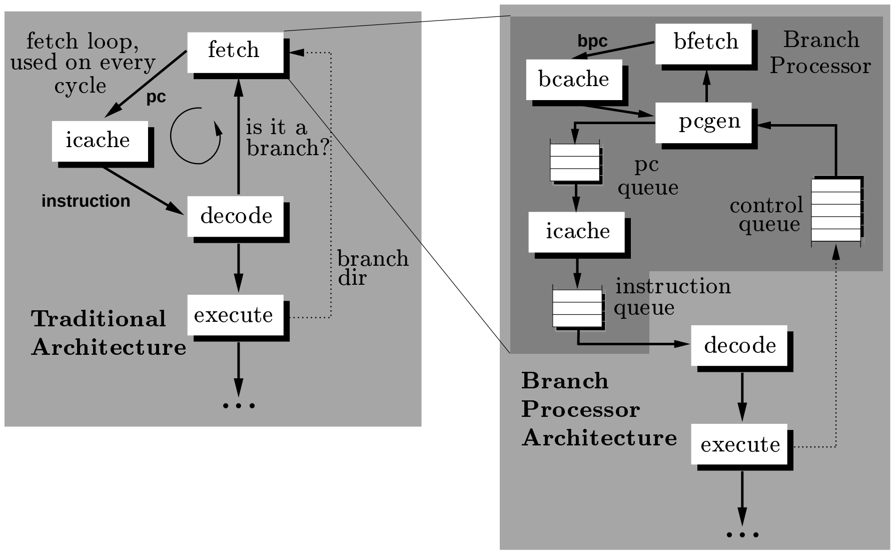

Figure 1: The Branch Processor Architecture.

```
        a[i]=b[i]+c[i];
    }
```
The branch processor can fetch the instructions after the loop before the loop execution completes, since the control-flow of the program is not data-dependent on the computation performed in the loop body.

When a program encounters a truly data-dependent branch, the branch processor cannot determine the next program counter value without feedback from the data processor. The data processor sends any information necessary via the control queue, and the branch processor reads this feedback information before proceeding. Note that while the branch processor can stall while performing this operation, the data processor might still be executing instructions since the branch processor has the capability to fetch blocks of instructions.

2.1. Branch Processor Features and Trade-Offs

We now discuss some of the issues and trade-offs in the design of a branch processor architecture.

Latency Tolerance. The branch processor is designed to "run ahead" of the data processor. If the instruction queue connecting the branch processor to the decode can be kept relatively full, then the architecture can partially hide the latency of an instruction cache miss. The branch processor architecture naturally prefetches instructions from the code stream, since it is decoupled from the data processor.

Code Compaction. Since data processor instructions no longer encode control-flow information, we can use the same sequence of instructions in different contexts by replicating branch processor code rather than data processor code. One of the transformations applied by modern compilers is code inlining. This typically increases the instruction cache footprint of a program because the same code stream is replicated at each call site [9]. If the scheduled code stream contains blocks of identical instructions, we can simply replicate the branch processor instructions and share the data processor instructions among call sites. Since the branch processor instruction stream is compact compared to the data processor instruction stream, this reduces the memory overhead of inlining, as discussed in Section 4. Since there are no branches in the data processor, we do not need branch delay slots for it; this eliminates the additional `nop` instructions that are inserted into unscheduled branch delay slots, further reducing code size.

Loop Unrolling. Special loop instructions in the branch processor make it possible to execute simple loops without feedback from the data processor. Knowledge of program loops enables the branch processor to continue execution beyond the loop while the data processor is executing the loop body. This also increases the tolerance to instruction cache misses, because the data processor could be busy while the cache miss is being serviced. Loops have long been a focus of compile-time analysis, especially in the context of parallelizing compilers for scientific applications [26]. Compiler-generated loop information can be passed down to the branch processor hardware so as to take advantage of control-flow information known at compile-time.

Function Call/Return. The stack architecture of the branch processor makes it possible to precisely determine the sequence of instructions for structured call/return sequences. Therefore, the return address of a function need not be predicted; it is known by the branch processor. The branch processor can fetch the instruction stream after the return of a function call while the epilogue of the function is executing using the information stored in its hardware stack. This technique, combined with loop unrolling, makes the branch processor architecture an attractive solution for scientific applications.

Improved Throughput. The branch processor is a much simpler processor than the data processor. In an asynchronous implementation, this implies that we can design the branch processor to have a higher average throughput compared to the data processor. In addition, we have removed some instructions from the data processor, making its decode and dispatch logic simpler. This combined effect can result in increased overall throughput for the data and branch processor, improving the performance of all instructions. In a clocked system, the branch processor architecture could potentially result in higher clock rates for the same reason. Another potential source of throughput improvement is the removal of the fetch loop, as discussed above. Section 4 shows how this results in improved overall throughput for an existing asynchronous processor.

Area Trade-offs. While the branch processor has a small instruction set, it does impact the overall area of the architecture. The largest area increase is due to the introduction of the branch processor cache that holds branch processor instructions. However, this increase could be offset by the fact that we have split the original code stream into two instruction streams—one for the data processor and one for the branch processor. In effect, the increased area due to the branch processor cache could be thought of as a larger effective instruction cache size for the processor. Using two, smaller instruction caches with a combined size comparable to the original cache size could also improve the throughput of the fetch loop, improving the performance of the entire processor.

Some processors store predecoded instructions in the instruction cache to improve decode performance. Any branch information that is part of the predecoded instruction can be removed from the cache, thereby reducing the number of bits stored per instruction for such architectures.

Memory Bandwidth. The branch processor architecture requires a split control/data code stream. Since the two code streams are decoupled, the branch processor cache has to share the access port to the processor's second-level cache and memory interface. This new requirement could destructively interfere with second-level cache performance. However, the branch processor architecture could have a reduced instruction cache miss rate, thereby alleviating this problem. In addition, the latency tolerance properties of the branch processor could more than offset any overhead introduced by destructive interference in the second-level cache.

One of the important features of the branch processor architecture is that it is a technique that can be readily combined with existing approaches to solving the problem of control dependences. Branch prediction can be incorporated into the branch processor itself, as can a branch-target buffer or predicated execution. We discuss some of the effects of combining a branch processor architecture with branch prediction and predication.

Differentiated Branch History. Several researchers have studied the effect of branch classification, and its impact on branch predictor performance [6]. The branch processor architecture differentiates loop branches, jumps for function calls, and jumps for function return from all other control-flow operations. Any branch predictor can use this information to selectively incorporate branch information into its history table. In addition, the predictor may incorporate additional high-level information—such as whether a program is executing the first or last iteration of a loop—that can be directly determined by inspecting the state of the branch processor stack.

Predication. Predicated execution is a technique for transforming control dependences into data dependences. Predication trades off executing branch instructions versus executing conditional instructions that may not contribute toward the computation. Predicated execution in the data processor would benefit the branch processor architecture because short sequences of conditional instructions could be transformed into predicated code thereby increasing the number of instructions fetched by a single block fetch instruction.

3. Initial ISA and Implementation Issues

In this section we present an initial instruction set architecture for the branch processor and compilation strategies for the proposed instruction set. The branch processor includes a hardware stack to store structured control-flow information, and we describe the effect each instruction has on the state of the stack.

There are times when control flow information is only available at run time. To execute such programs, we introduce a class of *forwarding instructions* in the main data processor. These instructions send a data value from the data processor to the branch processor via the control queue in addition to storing the result in the register file. These must be matched by a branch processor instruction that reads the data from the control queue. Instructions that receive values from this control queue have a "`?`" appended to them to make this explicit.

3.1. Instruction Set Summary

In what follows, *addr* refers to the address of instructions to be executed on the data processor, and *baddr* refers to addresses for branch processor instructions. `bpc` refers to the address of the current branch processor instruction, while `nextbpc` refers to the address of the next branch processor instruction.

Block Fetch. Block fetch instructions are introduced to compress control flow information within basic blocks. The instruction `fetch` *addr*, *N* fetches and executes *N* instructions that begin at address *addr*. This instruction can also be used to implement straight-line microcode. A sequential stream of instructions that implements a complex task can be invoked without increasing code size significantly by using a single `fetch` instruction. Using this instruction can result in a smaller instruction cache footprint when common code can be shared among different parts of the program, as shown in Section 4.

Loops. To permit simple loop constructs to be implemented without significant overhead, we introduce the following two instructions: `loop` *N* and `dec`. The `loop` instruction stores the pair $\langle \texttt{nextbpc}, N \rangle$ on the hardware stack. Branch processor execution continues with the next instruction. `dec` examines the pair $\langle baddr, N \rangle$ stored on the top of the stack, and decrements *N*. If the result is zero (or negative), the stack is popped; otherwise, the branch processor jumps to address *baddr*. For example, the code corresponding to a loop that executes a sequence of 15 instructions 10 times would be:

```
loop 10
fetch addr, 15
dec
```

The number of iterations in a loop is not always known at compile time. To permit the execution of loops with iteration counts determined at run time, we introduce the `loopN?` instruction. This instruction receives the next data value from the control queue and uses it as the loop count N (as in the normal `loop` instruction); other than that it behaves like a `loop` instruction. When breaking out of a loop, the hardware stack still contains state information that must be destroyed. The `pop` instruction accomplishes this by explicitly popping the top of the hardware stack.

Function Calls. Function calls are implemented with the `call` instruction. `call` *baddr* pushes $\langle \texttt{nextbpc}, 1\rangle$ onto the branch processor stack and transfers control to *baddr*. Returning from a function is implemented by a `ret` instruction, that jumps to the address on the top of the stack and pops the stack. The hardware stack is backed by main memory to handle programs with deeply-nested calls.

To execute a function call to an address determined at run time (this occurs when executing a function determined by looking at a function pointer stored in a table, or in the case of dynamic dispatch of methods in object-oriented languages), we introduce the `call?` instruction. This instruction reads the target address from the control queue, and otherwise behaves like a `call`.

Data-dependent Control Flow. The `loop` and `pop` instructions can be used to implement control flow in loops. To handle arbitrary branches, we introduce goto instructions of two flavors: `goto` *baddr* and `goto?`. The first instruction unconditionally changes the branch processor PC to *baddr*; the second instruction reads the target address from the control queue.

When control flow depends on computation performed by the data processor, the control queue is used to determine the direction of the branch. We use single-operand branch instructions `bgez?`, `bltz?`, `bgtz?`, `blez?` and two-operand branch instructions `beq??` and `bne??` for this purpose. The branch instruction `b`*cc* *baddr* reads a value (or two values) from the control queue and continues execution at address *baddr* if the value received is compatible with the condition code. Otherwise, execution continues with the next branch processor instruction.

Block Predication. We provide a simple mechanism for predicating a block of instructions. The instruction `fetch?` *addr*,*N* is used for this purpose. If the value received from the data processor is non-negative, then the block of N instructions stored at address *addr* are executed; otherwise, the instruction behaves like a `nop`.

Combined Loop/Branch and Fetch. The only branch processor instructions that actually compute data processor program counter values are `fetch` and `fetch?`. We propose to make the branch and loop instructions also support this functionality. Effectively, we collapse a branch/loop instruction followed by a `fetch` instruction into a single instruction. Instruction `floop` *cnt*,*addr*,*N* behaves like `loop` *cnt* followed by `fetch` *addr*,*N*. Instruction `floopN?` *addr*,*N* behaves like `loopN?`

Instruction	Purpose
`fetch` *addr*`,`*N*	fetch and execute block of instructions
`floop` *cnt*`,`*addr*`,`*N*	push loop counter and block fetch
`dec`	decrement/pop loop counter
`floopN?` *addr*`,`*N*	push loop counter, value unknown at compile time
`pop`	break out of loop
`call` *baddr*	function call
`call?`	function call, target unknown at compile time
`ret`	return from function call
`goto` *baddr*	arbitrary control flow
`goto?`	goto with target unknown at compile time
`fb`*cc* *baddr*`,`*addr*`,`*N*	conditional branches
`fetch?` *addr*`,`*N*	block predication

Table 1: Instruction set summary.

followed by `fetch` *addr*`,`*N*. Branch instructions of the type `fb`*cc* *baddr*`,`*addr*`,`*N* behave like normal `b`*cc* *baddr* instructions followed by `fetch` *addr*`,`*N*. Note that these instructions can implement the original loop and branch instructions. Executing the `floop` *cnt*`,`*addr*`,`*N* instruction pushes the value $\langle \texttt{bpc}, cnt \rangle$ with a bit set indicating that the "loop" part of the instruction has already been executed. Table 1 summarizes the basic ISA for the branch processor.

3.2. Deadlock, Exceptions, and Context Switching

The state in the branch processor architecture is distributed, since we have two streams of instructions that are being executed concurrently. In addition, we have instructions that synchronize the branch processor and data processor. In this section we examine some of the consequences of such an architecture, presenting solutions to the new issues they raise.

Deadlock Detection. Since our architecture includes explicit instructions that synchronize the data processor and branch processor, incorrect code could deadlock the hardware. To avoid this problem, we must be able to detect the occurrence of deadlock and correct the problem. Deadlock can occur in two different cases in the branch processor architecture:

- a receive on the control queue is blocked because there is no matching forwarding instruction and the queue is empty
- a send on the control queue is blocked because the queue is full and there is no matching branch processor instruction that receives a value from the queue

Every forwarding instruction must be `fetch`ed before the corresponding receive is executed in the branch processor. Therefore, the first case can only be caused by an incorrect program. This possibility can be prevented by using a correct compiler. The second case could occur if multiple forwarding instructions have been dispatched in advance, causing the control queue to become full before any receives could be executed. This case could also be prevented by a compiler that keeps track of the queue space at the end of each basic block, ensuring that the number of pending send operations does not exceed the hardware limit. Although both cases of deadlock can be prevented using appropriate compilation techniques, we might want to execute arbitrary programs on the hardware without causing the processor to deadlock. We discuss some deadlock-detection techniques below.

Deadlock can be detected by using a timing assumption or by running a deadlock detection algorithm. Simple timing assumptions include assuming that the processor has deadlocked if instructions have not been decoded for a long interval. A more direct solution is to execute a simple termination detection algorithm to detect deadlock [7]. In the latter case, we only have to involve the two ends of the control queue in the termination detection algorithm along with counters to detect that there are no data values in transit from the branch processor to the data processor.

Deadlock Recovery. If a receive action is blocked forever, the code being executed on the branch processor is erroneous. In this case, we must be able to begin execution of the exception handler. If a forwarding action is blocked forever, this could imply that the code generated by the compiler is erroneous; however, the processor might deadlock because the control queue is full. In this case, we should gracefully recover by permitting the program to continue execution while draining the values stored in the control queue.

To permit program execution in the presence of blocked forwarding instructions, we must be able to save (and restore) the values stored in the control queue to memory. We can do so by introducing instructions that save and restore the queue state. In particular, we can treat the control queue as a memory-mapped structure so that we can dynamically increase the queue size. With such an implementation, a blocked forwarding action will cause execution to fail only when a process exceeds its resource limits.

Exceptions and Context Switching. The processor architecture just proposed has state stored in both the data processor and the branch processor. To support context switching, the processor must have the capability to store its entire state to memory. The state of the data processor can be saved to and restored from memory in the same way as in traditional processors. The state of the branch processor is stored in the contents of the branch processor stack and the contents of the control queue between the data processor and branch processor. We add branch processor instructions to save and restore the hardware stack to memory to support this functionality.

Exceptions can be handled using a mechanism based on the one used by modern processors [15,28]. The forwarding instructions must be treated as instructions that modify the state of the control queue. In addition, if an exception is encountered in the middle of a block fetch instruction, we must be able to restore execution from the middle of the block. This implies that the branch processor must keep track of pending block fetch instructions so that they can be restarted after an exception is handled.

Exceptions that occur in the branch processor itself (such as address translation errors or stack underflows) can be handled by sending them to the data processor with a special flag set indicating a branch processor exception. The instruction will be executed as a `nop` in the data processor, and raise an exception in the usual way. Making the graduation unit in the data processor handle branch processor exceptions ensures that exceptions are handled in program order.

3.3. Compilation

Existing compilation techniques can be used to generate code for the branch processor architecture. The branch processor instruction stream is an encoding of the control-flow graph, while the data processor instruction stream corresponds to the sequence of instructions in a basic block.

Loop Detection. Both fixed length and variable length loops can be detected by modern compilation systems. Most programming languages have constructs for simple iterated loops, simplifying the problem of loop detection. Therefore, a compiler can generate branch processor instructions for loops. In addition, function calls and returns are explicit in the language. Therefore, these instructions can be easily generated by standard compilation systems. Indeed, the branch processor instruction set is easier to map to because the call and return semantics are directly implemented by the hardware.

Peephole Optimization. Peephole optimization can be used to schedule forwarding instructions away from the end of the block they are executed in. This provides early branch information via the control queue, improving the performance of the branch processor architecture. These techniques are similar to the ones used to schedule uses away from loads [9].

Code Sharing. Loop unrolling and loop peeling are transformations used to improve the performance of programs. Both transformations replicate the body of the loop to statically determine the direction of some of the branches in the loop body. Observe that such program transformations replicate code just in the branch processor; streams of instructions in the data processor can be re-used because they no longer encode any control flow information. This implies that we may not have as severe an impact on instruction cache performance by applying such transformations.

4. Architectural Evaluation

In this section we present a preliminary evaluation of the branch processor architecture, qualitatively and quantitatively evaluating some of the trade-offs discussed in Section 2.

4.1. Analysis

In the best case, the branch processor keeps the PC queue and instruction queue relatively full thereby keeping the data processor continuously busy. Instructions that cause the branch processor to wait are those with "`?`" in their name, since they receive a value from the control queue before continuing execution. For these instructions, we examine the conditions under which the branch processor is forced to wait.

Branches. Instructions of the form `fbgtz?` *baddr*`,`*addr*`,`*N* conditionally modify the branch processor PC. If the branch processor is waiting at this instruction, it can fetch the branch processor instruction at *baddr* instead of waiting for the control queue. Once the control queue contains the branch information, we can simply select between *addr*, and the address specified by the fetch instruction at the target of the branch. Assuming that the PC queue and instruction queue were empty (the worst-case scenario), the data processor would have two `nop` cycles before executing the next instruction. In a traditional architecture, this case corresponds to the one where the delay slot is filled with a `nop` and therefore there is no additional penalty since the traditional processor executes `bgtz` followed by the `nop`.

The worst-case occurs when the branch processor executes the `fbgtz?` and reads the control queue without stalling, but where the PC and instruction queues are empty. This case occurs when the branch and data processors are synchronized and when the last `fetch` operation executed two data processor instructions. In this case, the branch processor cannot fetch the branch processor instruction at *baddr* until a cycle later, introducing one additional `nop`. The likelihood of this case is examined later in this section.

Calls. The `call?` and `goto?` instructions correspond to jump register instructions that are not function returns. In these cases, the branch processor must wait until the target can be determined by the data processor. If the PC queue, instruction queue, and control queue are all empty when the instruction executes, then the data processor executes one additional `nop` compared with a traditional architecture.

Block Predication. The `fetch?` instruction introduces an additional `nop` when the block of instructions is killed by the value received from the control queue. There is no additional `nop` cycle when the instructions are executed.

Loops. An examination of the definition of the `floopN?` instruction shows that this instruction does not have to wait for the control queue before the branch processor continues execution. The value received by `floopN?` is only examined when the `dec` instruction is reached, during which the branch processor can continue execution.

4.2. Simulation

In this section we simulate the complete integer and floating-point SPEC95 benchmark suite [24] to determine several metrics that affect branch processor performance. The code was compiled using `gcc` version 2.7.2.3 [23] for a standard MIPS processor and run on an execution-driven processor emulator. The details of the simulator system are not important since we only report simple statistics like branch frequencies and dynamic basic block counts.

Branch Processor Code Size. The number of branch processor instructions required for a naive translation of standard assembly code into branch processor code can be estimated by examining the control-flow graph of the original program. Each basic block requires a `fetch` instruction, and the conditional branch at the end requires a `b`*cc* instruction. (A `b`*cc* followed by a `fetch` corresponds to the combined `fb`*cc* instruction.) Table 2 and Table 3 show the percentage of basic blocks (after removing branches) that have identical code sequences in the SPEC95 benchmark suite (% shared bblocks). These statistics were taken from code compiled by an unmodified version of `gcc`. Ignoring outliers, between 18% and 25% of the basic blocks in most SPEC95 applications are identical to other basic blocks in the code stream. The final change in static code size (in terms of number of instructions) for a naive translation of the SPEC95 benchmark suite is also shown in Table 2 and Table 3 (% change in code). Note that the actual code size may be larger, because branch processor instructions might need more bits for encoding the fetch address than data processor instructions. However, our preliminary results indicate that the combined static code size for the branch processor and data processor is no larger than the code stream for a traditional processor.

Another interesting statistic is the number of distinct instructions present in the data processor

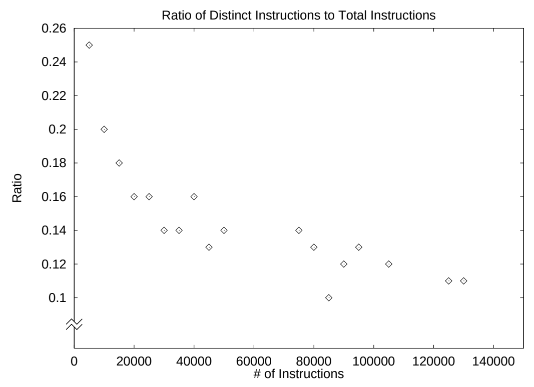

Figure 2: Fraction of unique instructions v/s total instruction count.

code stream. This corresponds to a compilation strategy where we attempt to share as many data processor instructions as possible at the expense of branch processor code size. Table 2 and Table 3 show the percentage of unique instructions in the code streams for the different SPEC95 benchmarks when branches and jumps are deleted (% unique insns). While the percentage of unique instructions varies from 8% for `gcc` to 28.4% for `compress`, this wide range is a result of differing total code size. A general trend that can be observed is that as the code size increases, the fraction of unique instructions decreases. This trend is demonstrated by Figure 2, which shows the fraction of unique instructions plotted against total code size for 267 standard system binaries for NetBSD compiled using `gcc`.

Another source of code compression that we have not measured or estimated (because we do not have a compiler for the branch processor at the time of writing) is the savings that result due to the special purpose loop and function call/return instructions. The loop instructions potentially save a comparison and subtraction per loop iteration because of the special-purpose branch processor hardware support. The savings of this form are hard to quantify without a realistic compiler, because optimization techniques like induction variable elimination can be used to share the loop index calculations with other induction variables required by the body of the loop. Function call and return instructions use the hardware stack to save and restore the return address, thereby eliminating instructions from the prologue and epilogue of non-leaf procedure calls that save and restore the return address of a procedure.

Branch Distance. For the branch processor to achieve maximum benefit, instructions that produce the source registers used in a later branch would ideally be scheduled as far in advance of the branch as possible. We refer to the distance between the branch use and the instruction that defines the register as the *branch distance*. The worst case for the branch processor occurs when the branch distance is one and the basic block size is small (3 or fewer instructions), since it restricts the ability of the branch processor to "run ahead" of the data processor. However, not all of these distance one branches in small basic blocks cause problems. For example, a two instruction basic block that branches on the return value of a procedure will not trigger worst-case behavior in the branch processor.

An exact analysis of the effect of branch distance on branch processor performance is difficult,

	go	m88ksim	gcc	compress	li	ijpeg	perl	vortex	avg.
Branch freq.	8.6	14.1	12.9	7.2	11.0	22.2	20.1	15.1	13.9
Call/Ret freq.	1.6	2.3	2.4	1.2	4.7	11.2	9.4	6.8	5.0
Avg. branch dist.	2.7	1.4	2.7	1.4	3.0	2.2	2.4	2.6	2.3
% bad blocks	8.3	4.5	11.0	6.0	7.6	14.3	14.0	11.2	9.6
% unique insns	13.4	21.1	8.0	28.4	22.0	19.9	15.3	11.0	17.4
% shared bblocks	18.1	21.1	26.6	18.3	20.9	22.5	24.6	33.8	23.2
% change in code	-7.0	-9.1	-15.8	-6.0	-9.8	-9.7	-14.1	-17.7	-11.2

Table 2: Measurements for the SPEC integer benchmarks compiled with `gcc`.

	tomcatv	swim	su2cor	hydro2d	mgrid	applu	turb3d	apsi	fpppp	wave5	avg.
Branch freq.	9.3	1.2	3.5	5.5	0.5	2.0	2.7	2.0	0.7	3.4	3.1
Call/Ret freq.	2.3	0.3	0.7	2.1	0.0	0.0	0.3	0.2	0.2	0.3	0.6
Avg. branch dist.	4.6	2.2	3.0	1.8	1.0	1.0	2.0	2.5	3.8	2.1	2.4
% bad blocks	18.8	7.7	15.7	3.8	0.1	0.0	8.7	5.5	6.7	5.5	7.2
% unique insns	24.4	24.8	23.4	22.5	24.1	22.5	25.8	21.0	27.3	19.9	23.6
% shared bblocks	19.9	19.3	19.1	19.7	19.8	19.0	20.4	20.8	18.9	18.3	19.5
% change in code	-8.1	-7.2	-6.3	-7.8	-7.8	-4.9	-7.2	-6.8	-4.8	-5.7	-6.7

Table 3: Measurements for the SPEC floating-point benchmarks compiled with `gcc`.

and requires a functional compiler for the branch processor architecture. As of this writing, we do not have a compiler. Still, we wanted to get some idea of what the branch distance was in real programs, so we measured the average branch distance in each of the SPEC95 integer and floating-point applications. The results of this study and the dynamic percentage of basic blocks that could potentially result in worse-case branch processor behavior (% bad blocks) are shown in Table 2 and Table 3. The results should be interpreted somewhat qualitatively, since the compiler that is used can dramatically affect the branch distance. We used `gcc`, which makes no effort to maximize branch distance. A compiler for the branch processor architecture would attempt to schedule the register definitions away from branches in the same manner that modern compilers attempt to schedule loads away from uses to hide read latency. Despite using a standard compiler, we find the branch distance numbers encouraging: the average branch distance is 2.3 for SPEC95 integer programs and 2.4 for SPEC95 floating-point programs. Another observation is that the dynamic percentage of "bad blocks" is only 9.6% for integer programs, and 7.2% for floating-point programs. Given no explicit compiler support and the fact that the percentages are conservative, we believe this bodes well for a complete branch processor implementation.

In addition, many of the branches in this branch distance study (especially in the floating-point programs) are loop branches that would actually be removed by the branch processor and converted into its `floop` instructions. Interestingly, these loop branches are often distance one branches since the loop induction variable is updated immediately before the branch. After a compiler for the branch processor schedules register definitions away from branches and removes loop branches, only truly data-dependent branches will remain. If these branches have large branch distances then the branch processor architecture should perform well. If these branches are of distance one, then ideally predication could be used to remove these branches, allowing the branch processor to run far enough ahead of the data processor to improve performance.

Throughput Improvement: A Case Study. One of the bottlenecks in the performance of the asynchronous MIPS processor from Caltech [15] is the fetch loop described in Section 2. In asynchronous systems, the throughput of the fetch loop is limited by the latency around the loop as well as the throughput of each pipeline stage in the loop [25]. HSPICE simulations using HP's nominal parameters for its 0.5μm CMOS process available through MOSIS indicated that the latency of the fetch loop was close to 7.2 ns, limiting the system throughput to 280 MHz [15]. In addition, the load on the output of the fetch was sufficient to limit the internal throughput of the fetch to 285 MHz. However, the cache and the rest of the execution units were capable of delivering an average-case throughput of 300 MHz. Adding buffers to the output of the fetch would add 0.6 ns to the loop latency; while this would have increased the internal throughput of the fetch to over 300 MHz, the overall system throughput would drop (due to the loop latency constraint) to 258 MHz. The introduction of the branch processor would have eliminated this loop latency constraint, permitting the fetch to operate at a throughput of 300 MHz thereby boosting system throughput. This is an overall 7% improvement in throughput as a result of improved timing, without considering any of the benefits outlined in Section 2. Indeed, this throughput improvement was the original motivation behind the branch processor architecture.

5. Related Work

There is a considerable body of work on hardware techniques for improving microprocessor branch performance. Branch prediction was first described by Anderson et al. in 1967 [2] and was expanded to include more modern techniques in the 80's and 90's by Smith [22], Lee and Smith [12], McFarling and Hennessy [16], and Yeh and Patt [29,30] among others. These modern techniques include two-level prediction schemes, branch-target buffers, and return address prediction. While the branch processor removes many of the branches from the control stream, some data-dependent branches remain. Although for some applications the branch processor can run ahead of the main processor and build up a queue of PCs that obviates the need for branch prediction, nothing precludes these traditional branch prediction techniques from being applied to the branch processor as well. However, since the branch processor has removed many of the easily predicted branches (loops and returns), prediction rates from traditional schemes for the remaining branches are likely to be much lower than usual. For prediction to make sense in the context of a branch processor, new branch prediction techniques or different predictor organizations may be necessary.

Like the branch processor, predication can remove branches from the instruction stream, converting them into conditional computational operations [1,10]. Recent work on hardware predication includes Mahlke et al. [13] and August [3,4]. We believe that a branch processor architecture should contain a predicated data processor. The effect of predication is to change control dependences into data dependences that are executed entirely on the data processor. The result would be an increased basic block size, a property that benefits the branch processor architecture because it increases the block fetch size and reduces the rate of synchronization with the data processor.

In fact, both the branch processor and the data processor could have predicated instructions. Short conditional branches can be replaced with predicated code on the data processor, whereas long conditional branches can be implemented with block predication instructions on the branch processor.

One of the potential benefits of the branch processor architecture is its ability to effectively prefetch instructions and hide fetch latency from the main processor. Other work has adopted compiler-directed solutions to this problem that require some hardware assistance [17,18,20]. Again, these techniques are orthogonal to the branch processor architecture and may help extend the branch processor's innate instruction prefetching ability. The branch processor architecture may further enhance these techniques because it performs block fetches of instructions and can run ahead of the data processor (even "exiting" a loop long before the data processors does) potentially hiding larger latencies. The branch processor also prefetches only instructions that the processor will actually execute, so it is not a guess—it simply fetches these instructions as early as possible.

Perhaps the work most closely related to the branch processor architecture is Smith's work on decoupled access/execute architectures in [21] and Goodman's follow-on work in [8]. The branch processor is also a decoupled architecture in that a program to be executed is divided into two or more instruction streams, and a number of "processors" cooperate in the execution of the program. While Smith's STRETCH and Goodman's PIPE machines decouple the execution of computational instructions from memory accesses, the branch processor architecture decouples the execution of computational instructions from control flow instructions. Both techniques use a set of queues that in best-case operation completely hide long latency events from the main computational engine. In Smith's case those events were memory accesses, and in the branch processor case those events are a combination of instruction fetch and branch resolution. All modern high-performance processors have incorporated some form of latency hiding for main memory and mechanisms that decouple their execution from the main microprocessor (multiple outstanding misses, lockup-free caches, prefetching, etc.) As clock rates increase and the fetch-branch feedback loop becomes more critical, techniques to decouple branches like those in the branch processor may find similar acceptance.

6. Concluding Remarks

The branch processor architecture decouples control flow from computation, allowing the branch processor to run ahead of the data processor and potentially providing benefits such as instruction cache prefetching, loop unrolling, code compression, reduced call/return overhead, and improved throughput. Furthermore, the architecture is orthogonal to traditional techniques such as branch prediction, predication, and instruction prefetching.

Despite the introduction of a new code stream, we showed that the total code size is smaller than the original code size for every application in the SPEC95 suite even using a conservative estimate. We provided an analysis of the conditions when the branch processor does not perform well, and discussed necessary features of a branch processor compiler that would reduce the frequency of those conditions. Note that the conditions under which the branch processor cannot run ahead of the data processor are the ones under which traditional processors exhibit poor performance. Techniques that increase effective basic block sizes such as trace scheduling, predication, and loop unrolling would allow the branch processor to run ahead of the data processor, boosting performance whenever the basic block size is increased.

We believe the branch processor architecture provides a framework for studying control-flow issues in modern processors, because of its clean separation of control and data information. Among the most interesting problems we plan to explore are novel compilation techniques for decoupled control architectures, the incorporation of a branch processor in high-performance asynchronous processors, and the effect of decoupling control and data on multi-threaded processors.

References

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

[2] D. W. Anderson, F. J. Sparaciso, and R. M. Tomasulo. The IBM 360 Model 91: Processor Philosophy and Instruction Handling. *IBM Journal of Research and Development*, **11**(1):8–24, January 1967.

[3] David I. August, et al. The Program Decision Logic Approach to Predicated Execution. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 208–219, May 1999.

[4] David I. August, et al. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227–237, July 1998.

[5] Dileep P. Bhandarkar. Alpha Implementation and Architecture. Digital Press, 1996.

[6] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale Patt. Branch Classification: a New Mechanism for Improving Branch Predictor Performance. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 22–31, December 1994.

[7] E.W. Dijkstra and C.S. Scholten. Termination Detection for Diffusing Computations, *Information Processing Letters*, **11**(1), August 1980.

[8] J. R. Goodman et al. PIPE: A VLSI Decoupled Architecture. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 20–27, June 1985.

[9] J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. 2nd Edition, Morgan Kaufmann, San Francisco, CA, 1996.

[10] P. Y. Hsu and E. S. Davidson. Highly Concurrent Scalar Processing. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 386–395, June 1986.

[11] Intel Corporation. Pentium Pro Family Developers Manual. Intel Corporation, 1996.

[12] Johnny K. F. Lee and Alan Jay Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, **21**(7):6–22, January 1984.

[13] Scott A. Mahlke, et al. A Comparison of Full and Partial Predicated Execution Support for ILP processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 138–150, June 1995.

[14] Rajit Manohar. *The Impact of Asynchrony on Computer Architecture*. Ph.D. thesis, California Institute of Technology, July 1998. Available as Caltech Technical Report CS-TR-98-12.

[15] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström et al. The Design of an Asynchronous MIPS R3000. *Proceedings of the 17th Conference on Advanced Research in VLSI*, pages 164–181, September 1997.

[16] Scott McFarling and John Hennessy. Reducing the Cost of Branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, 1986.

[17] Chi-Keung Luk and Todd Mowry. Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998.

[18] J. Pierce and T. Mudge. Wrong-Path Prefetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 264–273, December 1996.

[19] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, **3**(4):34–43, Winter 1995.

[20] Wayne A. Sawdon, Fay W. Chang, and Steven Lucco. A Preliminary Report on Software Prefetching in the Instruction Stream. *Workshop on Compiler Support for System Software (WCSSS)*, February 1996.

[21] James E. Smith. Decoupled Access/Execute Architectures. In *Proceedings of the 9th International Symposium on Computer Architecture*, pages 12–19, April 1982.

[22] James E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, 1981.

[23] Richard Stallman. Using and Porting GNU CC. Free Software Foundation, Cambridge, MA, June 1993.

[24] Standard Performance Evaluation Corporation. The SPEC95 Benchmark Suite. Details on-line at *http://www.specbench.org/.*

[25] Ted Eugene Williams. Self-timed Rings and their Application to Division. Ph.D. thesis, Computer Systems Laboratory, Stanford University, May 1991.

[26] M.E. Wolfe and M.S. Lam. A Loop Transformation Theory and An Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[27] Kenneth Yeager. Personal Communication.

[28] Kenneth Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, **16**(2):28–40, April 1996.

[29] Tse-Yu Yeh and Yale N. Patt. Alternative Implementations of Two-Level Adaptive Branch Predictions. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, May 1992.

[30] Tse-Yu Yeh and Yale N. Patt. A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993.