Comparing Software and Hardware Schemes For Reducing the Cost of Branches

Wen-mei W. Hwu Thomas M. Conte Pohua P. Chang

Coordinated Science Laboratory
1101 W. Sprintfield Ave.
University of Illinois
Urbana, IL 61801

Abstract

Pipelining has become a common technique to increase throughput of the instruction fetch, instruction decode, and instruction execution portions of modern computers. Branch instructions disrupt the flow of instructions through the the pipeline, increasing the overall execution cost of branch instructions. Three schemes to reduce the cost of branches are presented in the context of a general pipeline model. Ten realistic Unix domain programs are used to directly compare the cost and performance of the three schemes and the results are in favor of the software-based scheme. For example, the software-based scheme has a cost of 1.65 cycles/branch vs. a cost of 1.68 cycles/branch of the best hardware scheme for a highly pipelined processor (11-stage pipeline). The results are 1.19 (software scheme) vs. 1.23 cycles/branch (best hardware scheme) for a moderately pipelined processor (5-stage pipeline).

1 Introduction

The pipelining of modern computer designs causes problems for the execution of branch instructions. Branches disrupt sequential instruction supply for pipelined processors and introduce non-productive instructions into the pipeline. However, approximately one out of every three to five instructions is a branch instruction[1][2]. A significant increase in the performance of pipelined computers can be achieved through special treatment of branch instructions[3][4][1].

There have been several schemes proposed to reduce the branch performance penalty. These schemes employ hardware or software techniques to predict the direction of a branch, provide the target address of a branch, and supply the first few target instructions [3][4][1][5][6]. When the prediction is incorrect, the wrong instructions are introduced into the pipeline. After the branch instruction finishes execution and supplies the correct action to the instruction fetch unit, the incorrect instructions are flushed, or *squashed* from the pipeline [1]. These schemes rely on the assumption that the accuracy of the branch prediction scheme is high enough to mask the penalty of squashing. A small increase in the accuracy of a prediction scheme has a large effect on the performance of conditional branch instructions if the penalty for squashing an incorrectly predicted branch is large. Hence, highly accurate prediction schemes are desirable.

There have been many studies that investigate the effectiveness of solutions to the branch problem. Most of these studies focus on the accuracy of the branch prediction scheme employed [3][4][1][6][7]. Some studies also discuss hardware and software approaches to reducing the penalty of refilling the instruction fetch unit's pipeline [3][1][6].

Some schemes use static code analysis to predict branch behavior. One such scheme predicts all backward conditional branches as taken and all forward branches as not-taken. This is based on the assumption that backward branches are usually at the end of loops. In the study done by J. E. Smith [4], the average accuracy of this approach was 76.5%. However, in some cases this approach performed as poorly as only 35% accurate. Since the benchmarks used in the study were FORTRAN applications, which tend to be dominated by loop-structured code, the results may have been biased in favor of scientific workloads. Another study reported a 90% average accuracy for a static scheme, however the specific prediction mechanism was not reported nor was any additional statistical information besides the average [7].

Many architectures predict every conditional branch to either be all taken or all not-taken. In [1], this scheme is reported to be only 63% accurate if all branches are predicted taken. Similarly, [3] reports approximately 65±5%, [2] reports 67%, and [4] reports 76.7% of all branches are taken. Another static approach is to associate a prediction with the opcode of the branch instruction. This prediction is derived from performance studies and is stored

in a ROM or with the branch's microcode. The accuracy of this scheme is reported to be 66.2% on-average in [3] and 86.7% in [4].

Several dynamic branch history-based prediction schemes are presented in [3] and [4]. Dynamic approaches to branch prediction usually include hardware support in the form of a specialized cache to store the prediction information. For example, some schemes calculate the autocorrelation of the history vector of a branch instruction to generate a prediction; however, there is high hardware overhead for this scheme. Another, less-expensive scheme uses an up/down counter for prediction. J. E. Smith reports an accuracy of 92.5% for a two-bit version of this counter scheme. He reports a slightly smaller accuracy for larger counter sizes, due to the "inertia" caused by large counter sizes.

Another method of reducing the cost of branches uses information gathered during profiling a program for compile-time branch prediction. Note that this is different from static techniques since it uses the observed dynamic behavior of the branches for prediction. It is also separate from the other dynamic approaches because it does not require a large amount of hardware support. Usually, the instruction set is modified to include a prediction bit in the branch instruction format. This bit is used by the compiler to specify the prediction (i.e., predicted taken or not-taken) to the hardware. For example, this approach is used in the MIPS architecture [1].

Some previous schemes provide special support to make up for inaccurate branch prediction schemes. A common approach uses condition codes and optional compare instructions [8][9]. A case for single-instruction conditional branches is given in [6]. When a compare instruction must be added, the two instructions must be placed far-enough apart to predict the branch's behavior. However, this may not always be possible. Also, conditional branches now take two instructions instead of a single instruction. In order to make up for this increase in the dynamic instruction count, a hardware mechanism was included in the CRISP project to dynamically absorb the actual branch instruction into its preceding instruction and store it in a partially-decoded form. After all these techniques were used, the compiler designers for CRISP later suggested that a compiler-supported prediction mechanism might be useful to further improve performance [7].

To mask the penalty of flushing the pipeline when the prediction is incorrect, some schemes provide the first few instructions of the branch's target path. Some hardware buffer approaches store these instructions along with the prediction information. Reduced instruction set computer architectures often use a delayed branch to mask this penalty. For example, delayed branches are used in the Stanford MIPS [1] and the Berkeley RISC I [10]. In this approach, the compiler fills the delay slots following the branch instruction with instructions before the branch. While the fetch of the target instruction is being performed, the instructions in the delay slots are executed. These schemes rely on the compilers ability to fill the delay slots. McFarling and Hennessy report that a single delay slot can be successfully filled by the compiler in approximately 70% of the branches. However, a second delay slot could be filled only approximately 25% of the time [1]. Therefore, it is hard to support moderately pipelined instruction fetch units using the delayed branch technique.

The issue of which branch prediction scheme to use for VLSI-implemented monolithic processors is a topic still open to debate. The CRISP processor used significant hardware support for a static compiler technique [7][8]. The MIPS processor used delayed branches with squashing for an architecture with a relatively shallow pipeline (five stages)[1]. Since the silicon real estate is expensive for such processors, schemes that address the branch problem for processors implemented in VLSI should use little or no hardware support and achieve high performance. As more and more systems of all classes are being designed with single-chip central processors, new solutions to the branch problem that match or exceed the performance of traditional approaches must be developed.

This paper investigates three (two hardware and one software) schemes to solve the branch problem. These three schemes are presented and compared in the context of a very general pipelined microarchitecture. An optimizing, profiling compiler assists the evaluation of the performance of the schemes using a substantial number of benchmarks taken from the Unix[1] domain [11]. The experiments are controlled to isolate the effects of pipelining the instruction fetch unit from those of pipelining the instruction decode and instruction execution units.

The remainder of this paper is organized into three sections. Section two provides a concise description of the three schemes used to solve the branch problem: a simple branch target buffer, a counter-based branch target buffer, and a software approach. Section three presents the experimental results used for evaluating the performance of the three schemes. Finally, section four offers concluding remarks and future directions.

2 Background

This section introduces the three architectures that are used for the investigation. The first two of these architectures use additional hardware to solve the branch problem. The third architecture uses a profiling-compiler-driven software approach. All three architectures share a common model of a pipelined microarchitecture. This microarchitecture is composed of four smaller pipelines, or *units*, connected in series: the instruction fetch unit, the instruction decode unit, the instruction execution unit, and the state update unit (see Figure 1).

2.1 Pipeline structure

The instruction fetch unit is divided into $k+1$ stages, one stage to select the next address, and k stages to access the address. The next address selection logic takes

[1] Unix is a trademark of AT&T Bell Laboratories

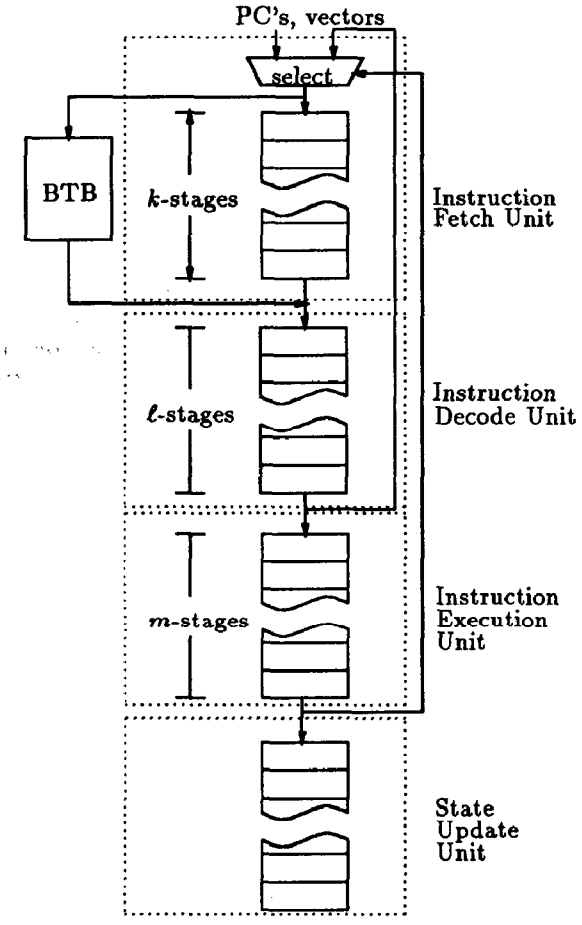

Figure 1: The pipelined microarchitecture.

various program counters and various interrupts and exception vectors to produce the address of the next instruction to fetch. Each branch instruction specifies a vector, or *branch target,* which is the address of the instruction to branch to. The subsequent k stages for instruction memory access take the instruction address generated and access the instruction memory hierarchy (i.e., instruction-buffer, instruction-cache, etc.).

The instruction decode unit is ℓ-stages in length. This stage decodes the instruction and calculates its actual operand values by decoding the operand the addressing modes and possibly accessing the register file or memory. Hence, the actual branch target and the branch action (for unconditional branches) is known at the end of this stage. This information is supplied through a feedback path to the selection logic of the instruction fetch unit.

The instruction execution unit is m-stages in length. The action of conditional branches is known when a branch reaches the end of the unit's pipeline. This information is supplied as a control signal in a feedback path to the selection logic of the instruction fetch unit. This pipeline may implement some form of interlocking, such as scoreboarding or the Tomasulo algorithm [12][13], or interlocking may be statically performed by the compiler. The effects of these interlocking strategies are parameterized to generalize the results (see below). It is assumed that comparisons are included in the semantics of the conditional branch instruction, as opposed to condition-code driven branch instructions. Finally, the state update unit is assumed to update memory, the register file, and/or the data cache with the results of executed instructions.

The issue of which instruction to fetch next is determined by the next address selection stage of the instruction fetch unit. In a simple next address selection stage, no special treatment is given to branches (i.e., branches are always predicted not-taken). If this prediction is incorrect, the wrong instructions will be introduced into the pipeline. These incorrectly-fetched instructions must be flushed from the pipeline when the actual branch behavior is determined. The instruction fetch unit's pipeline must always be flushed, and so must any incorrectly-fetched instructions in the instruction decode and instruction execution units' pipelines. A scheme should be provided for fast access to the k instructions following the branch target to hide the cost of flushing the instruction fetch unit.

Since on some machines the time to decode an instruction is not fixed but dependent on many factors (e.g., the complexity of the addressing modes used, the performance of the memory system, etc.), the penalty for flushing the pipeline of the instruction decode unit is treated as an average, $\bar{\ell}$, where $0 \leq \bar{\ell} \leq \ell$. Note that $\bar{\ell} = \ell$ for many RISC architectures. Due to interlocking, the number of instructions to flush from the instruction execution unit's pipeline may be determined by dependencies between instructions. Also, since unconditional branches are predicted with 100% accuracy, some branch instructions do not require any flush of the instruction execution unit. Hence, the penalty for flushing this unit's pipeline is also

taken as an average, $\bar{m}$. For compiler-implemented static interlocking, $\bar{m} = f_{\text{cond}} m$, where f_{cond} is the fraction of branch instructions that are conditional branches. Therefore, it is assumed that on average, $k + \bar{\ell} + \bar{m}$ instructions must be flushed from the pipeline for each branch. This observation will be used in Section 2.3 in stating the general formula for branch cost.

2.2 Three branch cost-reduction schemes

A *Simple Branch Target Buffer,* or SBTB, is used to remember as many as possible of the taken branches that are encountered in the dynamic instruction stream. To mask the penalty of flushing the instruction fetch unit, the SBTB stores the first k instructions of a taken branch's target path. For this reason, any branch instruction not in the SBTB is predicted to be not-taken. If a branch instruction is predicted taken, but when executed it does not branch to a new location, the corresponding entry in the SBTB is deleted. The SBTB may be thought of as cache that uses the branch instruction's location in memory as its associative tag. When it is full, a replacement policy is used to select an entry to replace. The accuracy of the SBTB's predictions is expressed as A_{SBTB}, the probability of the prediction being correct. The SBTB in this paper is a 256-entry fully-associative SBTB with a least-recently-used replacement policy.

Like the SBTB, a *Counter-based Branch Target Buffer,* or CBTB, is also a type of cache. It remembers as many as possible of the branch instructions encountered in the dynamic instruction stream. As with the SBTB, the CBTB also stores the first k instructions of the target branch to mask the instruction fetch penalty. The CBTB implemented for this paper stores a counter used for prediction along with each branch instruction [4]. For each new entry in the CBTB, the n-bit counter, C, is initially set to a threshold, T, if the branch was taken, or $T-1$, if it was not taken. Subsequently if the branch is taken, the counter is incremented, else it is decremented. When $C = 2^n - 1$, it remains at this value, and when $C = 0$, it remains at zero. A branch is predicted taken when $C \geq T$, else the branch is predicted not-taken. Any branch instruction not already in the buffer is predicted not-taken. The accuracy of the CBTB's predictions is expressed as A_{CBTB}, the probability of the prediction being correct. The CBTB in this paper uses a 256-entry fully-associative CBTB with a least-recently-used replacement policy for its branch prediction hardware. The counters used for prediction are 2-bits long and $T = 2$.

The SBTB or CBTB are accessed using the address from the select stage of the instruction fetch unit for every instruction retrieved from memory. This access occurs in parallel with the actual memory access performed in the instruction fetch unit. If the location causes a SBTB/CBTB hit, it is then known that the instruction is a branch. If the SBTB/CBTB's predicts the branch as taken (the SBTB always predicts a hit as a taken branch), the first k instructions following the target are sequentially supplied to the instruction decode unit (see Figure 1).

The third approach to branch prediction, the *Forward Semantic,* uses an optimizing, profiling compiler to predict the direction of all branches in a program. The SBTB/CBTB hardware shown in Figure 1 is not used in this scheme. Instead, the program is first compiled into an executable intermediate form with probes inserted at the entry of each basic block. The program is then run once or several times for a representative input suit. During the recompilation, predictions are made for each branch and stored by setting or clearing a "likely-taken" bit in the instruction format of each branch instruction [11]. The accuracy of these predictions is again represented as a probability that the prediction is correct, A_{FS}. Based on the profiling information, groups of basic blocks that are virtually always executed together are then bundled into larger blocks called *traces* [11][14]. The result is that all conditional branches that are predicted taken are placed at the end of these traces. For each branch that is predicted taken, $k + \ell$ locations, or *forward slots,* following the branch instruction are reserved. The $k + \ell$ instructions from the target path of the branch are copied into these slots. During the execution, when the instruction is determined to be a branch instruction at the end of the instruction decode unit, the instructions in the forward slots will mask the penalty of incorrectly fetching the $k + \ell$ instructions following the branch. Hence, these instructions serve the same purpose as the k instructions stored with each entry in the SBTB or CBTB.

To fill the forward slots, the traces are sorted by execution weight. The following algorithm is then used to fill the slots, where there are N traces, `trace[i]` is the trace with the ith largest weight, `trace[i]->next_trace` is the target trace, `target_addr[trace[i]]` is the target address of the branch instruction at the end of trace `trace[i]`, and `trace[i]->offset_into_trace` is the branch target address, expressed as an offset from the beginning of the target trace.

```
for i ← N downto 1 step -1 do
    next_trace ← trace->next_trace;
    offset ← trace->offset_into_trace;
    length ← size_of(next_trace) - offset;
    if (length ≥ k + ℓ) then
        Copy the next k + ℓ instructions
          of trace[i]->next_trace to
          the end of trace[i];
        target_addr[trace[i]] ←
            target_addr[trace[i]] + k + ℓ;
    else
        Copy the remaining instructions
          of next_trace to the
          end of trace[i];
        Fill the remaining forward slots
          with NO-OP's;
        target_addr[trace[i]] ←
            target_addr[trace[i]] + length;
    endif;
```

An example of the algorithm is shown in Figure 2. The branch instruction originally at location 5 is an unlikely branch. Therefore, it can be absorbed into the forward slots of the branch instruction at location 2. Note that the target for this branch is not altered when it is absorbed into the forward slots. The instructions in the forward slots at locations 3 and 4 of the altered program fragment execute using an alternate program counter register value which in the example will be set to location 7.

	1: I_1
1: I_1	2:beq $pc+3$ *(likely)*
2:beq $pc+3$ *(likely)*	3:beq $pc+3$ *(unlikely)*
3: I_3	4: I_6
4: I_4	5: I_3
5:beq $pc+3$ *(unlikely)*	6: I_4
6: I_6	7:beq $pc+3$ *(unlikely)*
7: I_7	8: I_6
8: I_8	9: I_7
9: I_9	10: I_8
	11: I_9

Figure 2: An example of the Forward Semantic: original program fragment (*left*), and after application of the algorithm (*right*).

Note that the Forward Semantic is different from the "Delayed-Branch with Squashing" scheme presented in [1]. In that scheme, no branch instructions could be absorbed into the delay slots following the branch instruction. Also, the most-recently prefetched instruction and the instructions specified in the delay slots after the branch instruction were the instructions that would be squashed if the prediction was incorrect. However, in the Forward Semantic scheme, although $k+\bar{\ell}+\bar{m}$ instructions are flushed from the pipeline, only $k+\ell$ forward slots following the branch are used. Hence, a Forward Semantic implementation for the architecture presented in [1] would have used only one forward slot following the branch instead of two, since $k=0, \ell=1, m=2$ for MIPS-X.

2.3 Branch instruction cost

Whenever an incorrect prediction is made, the entire pipeline may potentially be flushed. This means the cost for an incorrect prediction for any of the three schemes is $k+\bar{\ell}+\bar{m}$. When the prediction is correct, each of the three schemes successfully covers the flushing of the pipelines. Hence, the cost of executing a branch instruction for any of the three architectures is,

$$\text{cost} = A + (k + \bar{\ell} + \bar{m})(1 - A),$$

where $A = A_{SBTB}$, for the SBTB, $A = A_{CBTB}$, for the CBTB, and $A = A_{FS}$ for the Forward Semantic. This equation will be used in the remainder of this paper to calculate the cost of branches for the three architectures given the accuracy of the three prediction schemes. Assuming that time is measured in clock cycles, and each stage of the pipeline has a latency of one clock cycle,

3 Experimental Results

Table 1 summarizes several important characteristics of the benchmarks used for the experiments below. The *Lines* column shows the static code size of the C benchmark programs measured in the number of program lines. The *Runs* column gives the number of different inputs used in the profiling process. The *Inst.* column gives the dynamic code size of the benchmark programs, measured in number of compiler intermediate instructions. The *Control* column gives the percentage of dynamic conditional and unconditional branches executed during the profiling process. Both *Inst.* and *Control* are accumulated across all of the runs. Finally, the *Input description* column describes the nature of the inputs used in the profiling process. As reported in many other papers, the number of dynamic instructions between dynamic branches is small (about four).

The *Conditional* column of Table 2 confirms that on average 61% of the dynamic branches generated by the compiler are not-taken branches. When the SBTB or CBTB generates a miss for a given branch, the instruction fetch unit cannot fetch the target instructions in time, which forces the fetch unit to continue to fetch the next instruction. This is equivalent to predicting that the branch is not taken. Since the majority of the dynamic branches are not taken, the predictions made upon SBTB misses are actually accurate. Since only taken branches make their way into the SBTB, the low percentage of taken branches also reduces the number of entries needed in SBTB to achieve high prediction accuracy. Therefore, we can expect the SBTB performance reported below to be better than equivalent designs reported by the previous papers.

The *Known* column in Table 2 gives the percentage of availability of the target address for unconditional branches. Unconditional branches with known target address can be easily handled by all the three schemes as (extremely biased) likely branches. Branches with unknown target addresses (i.e., the address is generated as run-time data) pose a problem for all three schemes. Fortunately, almost all the unconditional branches for the benchmarks have known target addresses. Therefore, all the three schemes work well with the unconditional branches.

The performance of the benchmarks for the three architectures are presented in Table 3. The miss ratio for the SBTB, ρ_{SBTB}, is much larger than the miss ratio for the CBTB, ρ_{SBTB}. This is to be expected since only taken branches are saved in the SBTB, whereas all branches are eligible to be stored in the CBTB. Note also that the differences in prediction accuracy (i.e., A) between the three schemes increases with the complexity of the prediction mechanism used. A SBTB uses essentially information based on the most recent behavior of a branch instruction in the dynamic instruction stream. Since the counter used for the CBTB is 2-bits long, the CBTB bases its predictions on the four most-recent branches in the dynamic instruction stream. Hence, the CBTB predicts branch behavior slightly more accurately than does the SBTB. The most accurate scheme, the Forward Semantic, uses the

behavior of the branch throughout the entire dynamic instruction stream for its predictions.

Observe that the accuracy values for all three architectures are very similar. However, if context switching had been simulated, one would expect the performance of the SBTB and the CBTB to be less impressive [3]. Note though that the prediction accuracy of the Forward Semantic would not have changed in the presence of context switching. Finally, both the SBTB and the CBTB are fully associative to provide the highest possible hit ratio. With 256 entries, it may not be feasible to implement full associativity. Hence, the results are biased slightly in favor of the two hardware approaches.

The values of $k = 1, 2, 4,$ and 8 and the averages from Table 3 of A were used for the four graphs of branch cost versus $\bar{\ell} + \bar{m}$ in Figures 3 and 4, where SBTB cost is shown as a solid line, CBTB cost is a dashed line, and Forward Semantic cost is a dotted line. These figures show that as the length of the instruction fetch pipeline grows, the difference between the three architectures increases as does the overall branch cost. Increasing the length of the instruction decode and instruction execution pipeline also increases the difference between the three architectures.

Modern microprocessors have relatively shallow pipelines with a two-stage instruction fetch pipeline (e.g., $\bar{\ell} + \bar{m} \approx 2$, $k = 1$). Pipelining the on-chip cache memory system is a difficult task. Future increases in pipelining may therefore occur in the instruction decode unit. To see the effect of this possible design shift, the results for all benchmarks for $k + \bar{\ell} = 2$ and 3, and $\bar{m} = 1$ is presented in Table 4.

Note that the three schemes do have a slight increase in branch cost for the transition from $k + \bar{\ell} = 2$ to $k + \bar{\ell} = 3$ for each benchmark. The average percentage of increase in branch cost is 7.7%, 6.9%, and 5.3%, for the SBTB, the CBTB, and the Forward Semantic, respectively. Hence, the Forward Semantic reacts the best to scaling the degree of pipelining, the CBTB is next, and the SBTB is the least scalable.

Although the Forward Semantic has a slightly lower branch cost, code-size increases occur due to the copying of instructions into forward slots after each predicted-taken branch. Table 5 summarizes this effect. Because copying instructions into forward slots increases the spatial locality of the program, the expanded static code size does not translate linearly into increased miss ratios of instruction caches. Therefore, considering the saving of hardware over SBTB and CBTB, the Forward Semantic is definitely a favorable choice according to the benchmarks.

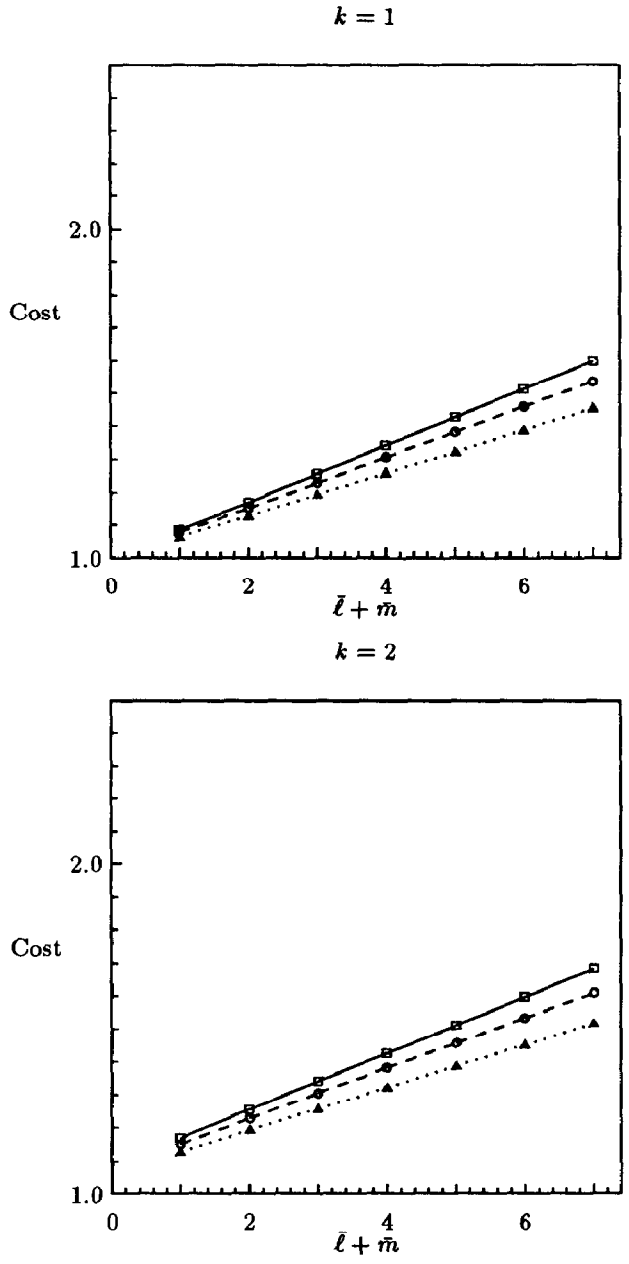

Figure 3: Branch cost vs. $\bar{\ell} + \bar{m}$ for $k = 1$ and $k = 2$.

4 Conclusions

This paper introduced a software approach to reducing the cost of branches, the Forward Semantic, which is supported by a profiling, optimizing compiler and uncomplicated hardware. A model was presented for the cost of branches which is significantly more general than previous models. One of the main features of this model is the in-

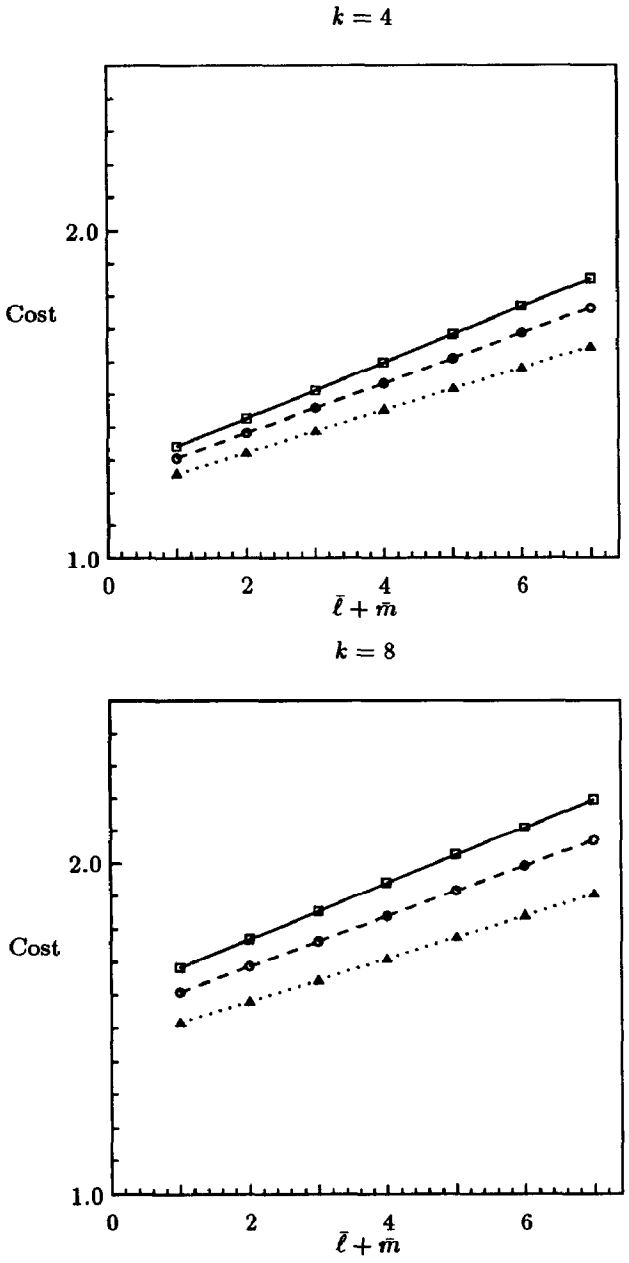

Figure 4: Branch cost vs. $\bar{\ell} + \bar{m}$ for $k = 4$ and $k = 8$.

dependent treatment of the instruction prefetch unit and the instruction execution unit.

The measurements performed for this paper were fair to all three architectures considered. The exact same benchmarks with the same inputs were used to derive the data for all three architectures, even though two architectures involved hardware schemes and one involved a software/compiler scheme. This provided a fair comparison between the Forward Semantic and the two hardware approaches.

The results of the performance study are encouraging. They indicate that the Forward Semantic compares favorably with the two other approaches. If context switching had been simulated, the Forward Semantic's performance would have remained the same, whereas the performance of the other two schemes would have suffered. The hardware needed for the Forward Semantic is considerably less complex than required for the other two schemes. Since the hardware schemes need to be accessed fast by the instruction prefetch pipeline, these schemes would have to be implemented on-chip in a microprocessor, using up valuable area. The Forward Semantic frees this area for other uses without sacrificing performance. Use of the Forward Semantic does cause an increase in code size, however. This additional code adds to the spatial locality of the program, since executing the instructions in forward slots often will cause the branch target's instructions to be in the instruction cache. For deep pipelines (e.g., $k + \ell = 4$), the Forward Semantic with its moderate 14.12% code-size increase seems to be more favorable than the the hardware of the SBTB/CBTB schemes, which increase linearly with k.

Acknowledgements

The authors would like to thank Sadun Anik, Scott Mahlke, Nancy Warter, and all members of the IMPACT research group for their support, comments and suggestions. This research has been supported by the National Science Fundation (NSF) under Grant MIP-8809478, a donation from NCR, the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS), the Office of Naval Research under Contract N00014-88-K-0656, and the University of Illinois Campus Research Board.

References

[1] S. McFarling and J. L. Hennessy, "Reducing the cost of branches," in *Proc. 13th Annu. Symp. on Comput. Arch.*, (Tokyo, Japan), pp. 396–403, June 1986.

[2] J. S. Emer and D. W. Clark, "A characterization of processor performance in the VAX-11/780," in *Proc. 11th. Annu. Symp. on Comput. Arch.*, pp. 301–309, June 1984.

[3] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, Jan. 1984.

[4] J. E. Smith, "A study of branch predition strategies," in *Proc. 8th Annu. Symp. on Comput. Arch.*, pp. 135–148, June 1981.

[5] D. J. Lilja, "Reducing the branch penalty in pipelined processors," *IEEE Computer*, July 1988.

[6] J. A. DeRosa and H. M. Levy, "An evaluation of branch architectures," in *Proc. 15th. Annu. Symp. on Comput. Arch.*, pp. 10–16, June 1987.

[7] S. Bandyopadhyay, V. S. Begwani, and R. B. Murray, "Compiling for the CRISP microprocessor," in *Proc. 1987 Spring COMPCON*, pp. 86–90, 1987.

[8] D. R. Ditzel and H. R. McLellan, "Branch folding in the CRISP microprocessor: reducing branch delay to zero," in *Proc. 14th Annu. Symp. on Comput. Arch.*, pp. 2–9, June 1987.

[9] Digital Equipment Corp., *VAX11 Architecture Handbook*, 1979.

[10] D. A. Patterson and C. H. Sequin, "RISC I: a reduced instruction set VLSI computer," in *Proc. 8th Annu. Symp. on Comput. Arch.*, pp. 443–457, May 1981.

[11] W. W. Hwu and P. P. Chang, "Trace selection for compiling large C application programs to microcode," in *Proc. 21st Annu. Workshop on Microprogramming and Microarchitectures*, (San Diego, CA.), Nov. 1988.

[12] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, Jan. 1967.

[13] J. E. Thornton, "Parallel operation in the Control Data 6600," in *Proc. AFIPS FJCC*, pp. 33–40, 1964.

[14] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. c-30, no. 7, pp. 478–490, July 1981.

Table 1: Benchmark characteristics

Benchmark	Lines	Runs	Inst.	Control	Input description
cccp	4660	20	11.7M	19%	C progs (100-3000 lines)
cmp	371	16	2.2M	22%	similar/disimilar text files
compress	1941	20	19.6M	16%	same as cccp
grep	1302	20	47.1M	36%	exercised various options
lex	3251	4	3052.6M	37%	lexers (C, Lisp, awk, pic)
make	7043	20	152.6M	21%	makefiles
tee	1063	18	0.43M	40%	text files (100-3000 lines)
tar	3186	14	11M	14%	save/extract files
wc	345	20	7.8M	28%	same input as cccp
yacc	3333	8	313.4M	25%	grammar for C, etc.

Table 2: Benchmark branch statistics

Benchmark	Conditional		Unconditional	
	Taken	Not	Known	Unknown
cccp	31%	69%	81%	19%
cmp	20%	80%	100%	0%
compress	37%	63%	100%	0%
grep	5%	95%	100%	0%
lex	49%	51%	100%	0%
make	49%	51%	100%	0%
tar	89%	11%	100%	0%
tee	44%	56%	100%	0%
wc	24%	76%	100%	0%
yacc	47%	53%	100%	0%
Average	40%	61%	98%	1.9%

Table 3: Branch prediction performance of the benchmarks.

Benchmark	Branch prediction scheme				
	SBTB		CBTB		FS
	ρ_{SBTB}	A_{SBTB}	ρ_{CBTB}	A_{CBTB}	A_{FS}
cccp	0.57	90.7%	0.018	91.5%	89.6%
cmp	0.70	97.1%	0.0032	98.0%	98.6%
compress	0.49	87.8%	0.0053	86.1%	89.1%
grep	0.76	93.7%	0.0006	95.9%	96.0%
lex	0.36	98.2%	0.0002	97.7%	98.0%
make	0.42	90.5%	0.012	92.5%	94.4%
tar	0.11	97.9%	0.005	98.4%	98.7%
tee	0.39	84.4%	0.0058	88.7%	92.2%
wc	0.54	85.4%	0.0008	85.7%	90.4%
yacc	0.46	88.9%	0.0012	89.1%	88.3%
Average	0.48	91.5%	0.0053	92.4%	93.5%
Std. dev.	0.18	5.06%	0.0058	4.92%	4.13%

Table 4: Branch cost for $k + \bar{\ell} = 2$ and 3, and $\bar{m} = 1$

Benchmark	$k+\ell=2$			$k+\ell=3$		
	SBTB	CBTB	FS	SBTB	CBTB	FS
cccp	1.19	1.17	1.21	1.28	1.26	1.31
cmp	1.06	1.04	1.03	1.09	1.06	1.04
compress	1.24	1.28	1.22	1.37	1.42	1.33
grep	1.13	1.08	1.08	1.19	1.12	1.12
lex	1.04	1.06	1.04	1.06	1.07	1.06
make	1.19	1.15	1.11	1.29	1.23	1.17
tar	1.04	1.03	1.03	1.06	1.05	1.04
tee	1.31	1.23	1.16	1.47	1.34	1.23
wc	1.29	1.29	1.19	1.44	1.43	1.29
yacc	1.22	1.22	1.23	1.33	1.33	1.35
Average	1.17	1.15	1.13	1.26	1.23	1.19
Std. dev.	0.10	0.098	0.083	0.15	0.15	0.12

Table 5: Percentage of code-size increase as a function of k.

Benchmark	Percentage code-size increase			
	$k+\ell=1$	$k+\ell=2$	$k+\ell=4$	$k+\ell=8$
cccp	2.79%	5.80%	11.75%	29.57%
cmp	1.87%	3.74%	7.48%	14.96%
compress	2.10%	4.15%	8.82%	20.26%
eqn	3.50%	7.44%	14.87%	44.26%
espresso	4.19%	8.51%	17.82%	39.28%
grep	1.55%	3.36%	6.96%	15.81%
lex	5.68%	11.34%	24.08%	53.73%
make	3.93%	7.96%	16.35%	37.76%
tar	2.82%	5.89%	12.18%	27.17%
tee	1.29%	2.52%	5.34%	10.75%
wc	1.70%	3.41%	8.52%	19.00%
yacc	7.41%	15.43%	35.21%	82.92%
Average	3.24%	6.61%	14.12%	32.96%
Std. dev.	1.84%	3.83%	8.55%	20.52%