Hardware Loop Buffering

Scott DiPasquale, Khaled Elmeleegy,
C.J. Ganier, Erik Swanson

Abstract

Several classes of applications can be characterized by repetition of certain behaviors or the regular distribution of the data sets. The extensive use of loops to organize the control flow of the program and data processing magnify the value of performance improvements in these regions of code. The continuing demand for higher performance machines in areas of media processing and floating-point programs also leads to application-targeted architectures being developed to meet the market's needs. To accommodate loop intensive applications we propose specific ISA and architectural additions to improve the performance of certain classes of loops found in media and scientific applications. Compilers frequently practice loop unrolling to hide latency, but the compiler lacks sufficient information to work with many loops. Through the elimination of branch instructions and other unrolling techniques, significant speedup can be achieved; however, compilers lack information on the bounds of many loops whose limits are determined only at runtime. By implementing semi-dynamic and static branch elimination in hardware, our architecture takes better advantage of compiler information to extract greater efficiency in loops and yield a speedup with a moderate increase in complexity. We tested our modifications in simulation on an out-of-order architecture, and the results demonstrate the utility of our changes in the scope of the benchmark.

1. Introduction

The goal of improving the performance of certain stretches of common code is an essential goal of hardware modification. Driven by the specter of Moore's Law and by a demand for better performance on media, scientific, and other applications, the pressure to squeeze small performance gains out of modern high-performance processors makes moderate gains seem appealing at even a high complexity cost. Targeting loops for performance gains has traditionally been handled by compiler loop unrolling algorithms in which instructions from subsequent iterations are blended into each other. The benefits of this can be two-fold. Delays from high-latency instructions can be hidden by interleaving them with instructions from the following iteration. This benefit sometimes suffers from additional instruction overhead incurred to detect and recover from the instructions executed from iterations after the end of the loop; however, if sufficient instruction level parallelism (ILP) is present, the compiler can hide most delays with instructions that are only from the same iteration of the loop. Second, if the bounds of the loop are known at compile time, branch instructions can be eliminated by reducing the number of conditional checks and recovery instructions involved in the loop. In both cases, loops whose bounds are determined at runtime hamper compiler efforts to efficiently resolve loop speculation.

To further exploit the available information in loops, we propose modifying the instruction set architecture (ISA) with appropriate additions to supply the hardware with information the compiler has about the loop but is unable to exploit on its own. These cases are static and semi-dynamic loops that are well-formed.

Static loops are those loops where the number of iterations is known at compile time.

Dynamic loops are loops whose termination conditions are unknown
when the loop begins execution at runtime.

Semi-dynamic loops fall in between static and dynamic; they are loops whose bounds are determined at runtime but before the loop begins execution.

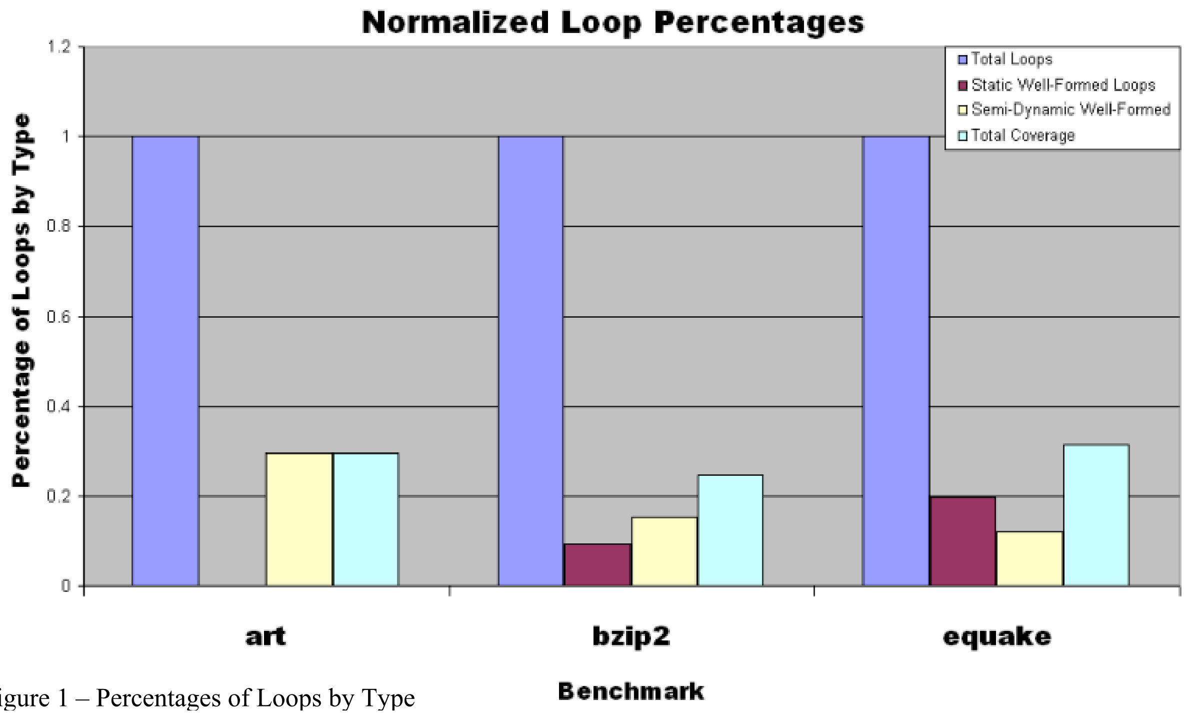

Figure 1 – Percentages of Loops by Type

Well-formed loops are loops whose instructions are all consecutive with no branches, jumps, etc.

Compilers frequently encounter loops whose indices are either known or set just before the beginning of the loop. With foreknowledge of which register will be used to determine the bounds of the loop, the compiler has done the analysis that is useful to hardware loop speculation, but too complicated to be done in hardware. Our ISA additions provide the hardware with the number of iterations that will be executed in the loop, the beginning instruction, and the ending instruction of the loop. With these specific bounds, the hardware is able to eliminate branch logic in the loop. This provides instruction savings inversely proportional to the length of the loop, proportional to the number of times the loop is executed, and limited primarily by the stalls that compiler techniques attempt to minimize. Through the use of semi-dynamic and static hardware loop buffering, the number of instructions per clock cycle should increase and the total number of instructions should decrease.

In the subsequent sections of this paper, we will cover the purpose, methods, and results of this project. Section 2 shows the frequency of semi-dynamic, static, well-formed loops in several benchmarks to explain the motivation for this project; this section also addresses the reasons for not performing actual unrolling in hardware. Section 3 explains the ISA changes, and Section 4 describes the hardware support used in the simulations and the way this differs from other implementations of similar ideas. Section 5 explains the simulation architecture and its results. Section 6 gives a brief description of extending buffering to loop unrolling. Finally, we discuss the conclusions and future possibilities for the idea.

2. Motivation

According to Amdahl's Law, in order for an improvement to microprocessor operation to be significant, there must be an observed frequency of occurrences such that the improvement will be used a good percentage of the time. To determine whether or not it was worthwhile to pursue hardware loop buffering as a viable way to increase processor performance, the total number of loops, number of static loops, number of well-formed static loops, and the total number of semi-dynamic well-formed loops were observed for the following SPEC2000 benchmarks. 179.art is an object recognizing neural network, 256.bzip2 is an integer data compression utility, and 183.equake is a floating-point earthquake simulator.

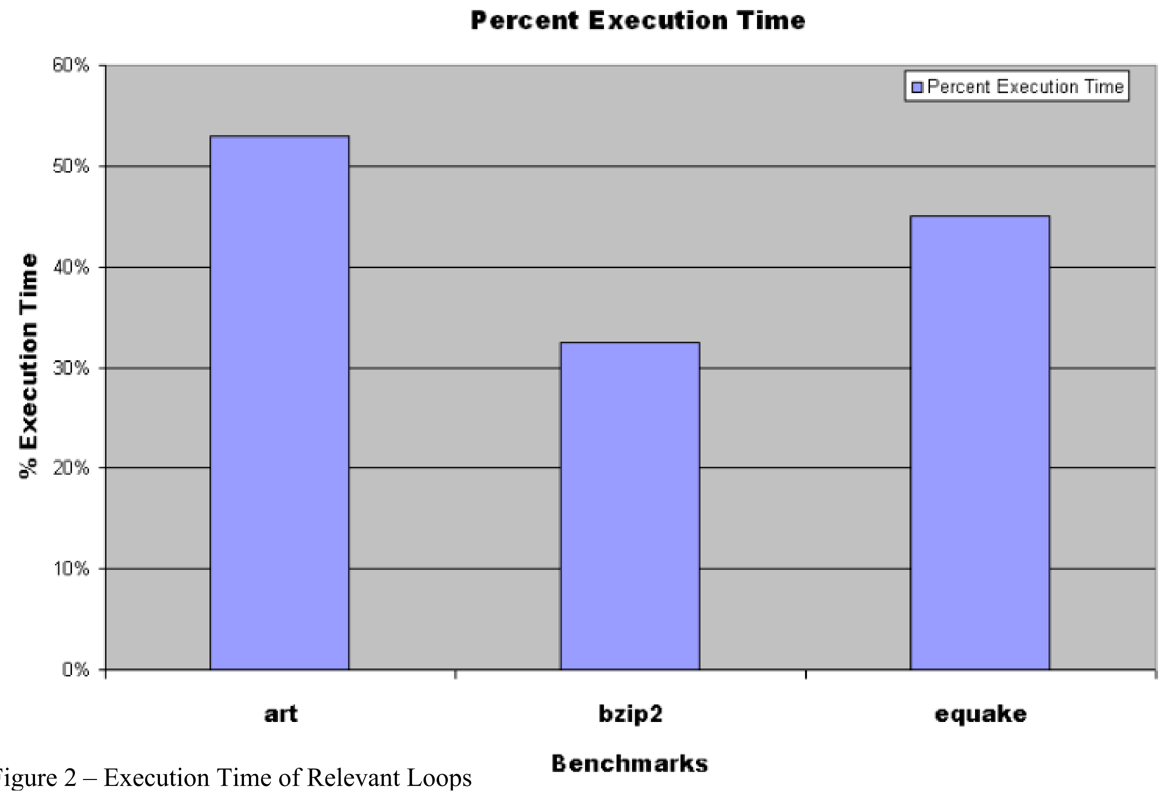

Figure 2 – Execution Time of Relevant Loops

Figure 1 presents the total number of each type of loop normalized to the total number of loops. As is observed, the hardware loop buffering mechanism would cover 29.6% of loops in art, 24.6% in bzip2, and 31.5% in equake. The average percentage of loops that meet our constraints across the three benchmarks is 28.6%. Not only is the percentage of loops important to show that improvement in loop execution would affect performance, but the percentage of execution time that these loops consume during program execution is also important. Figure 2 presents the amount of execution time spent in the relevant loops within the benchmarks. In equake, the loops consume 45% of the total execution time. The loops in art represent 53% of the execution time, and the loops for bzip2 32.5%. This is more than sufficient to show that any speedup of these loops would provide significant performance improvements of the overall program.

Semi-dynamic loops are those that the compiler cannot unroll or modify due its inability to determine vital information at compile time. The majority of the speedup gained by hardware loop buffering over compiler-optimized code would come from the ability to optimize untouched loops. All of the loops covered in art are actually semi-dynamic loops that the compiler would be unable to unroll. The semi-dynamic loops make up 12% of the loops in equake and 15.2% in bzip2. These loops would represent the majority of our speedup over compiler-optimized code.

Loop unrolling was not implemented in this project, although it would very likely provide additional improvement over hardware buffering. Developing a generalized algorithm to remap registers in duplicated instructions would require redirecting much of the simulator's execution pipeline, and the specific case of index dependent loads required altering registers on the fly. Since we are already modifying the execution pipeline, which has proved to be a very difficult task, these further modifications that would allow correct functioning of unrolling would be a major undertaking and are beyond the scope of this project.

2. ISA Changes

In most ISAs, a loop is a branch instruction whose target is a previous instruction. The execution of the program passes through the same instructions to this branch until eventually the branch is not taken, and the control flow leaves the branch. Standard branch prediction techniques such as 2-bit branch prediction are guaranteed to mispredict, while more complicated schemes require executing the loop fully once to correctly predict it in the future. The irony of

Original Loop	**Hardware Buffer Setup Loop**
-	-
LOOP:	LOOP r6, REG
	-
SLL r3, r2, 2	LOOP:
ADD r3, r3, r5	
LW r1, 0(r3)	SLL r3, r2, 2
ADD r1, r1, r4	ADD r3, r3, r5
SW r1, 0(r3)	LW r1, 0(r3)
ADDI r2, r2, 4	ADD r1, r1, r4
SGT r5, r2, r6	SW r1, 0(r3)
BEQ r0, r5, LOOP	ADDI r2, r2, 4
	ENDL

Table 1 – Loop modification example, usage of LOOP and ENDL instructions

these mispredictions is that in many cases the information the branch predictor needs is held by the compiler. The programmer often makes the conditions of the loop explicitly a count down or up, basing the termination condition on the increment of a variable. To extract performance benefits from compile-time information, it is necessary to pass the information to the hardware. Our proposal is to change the ISA of the system in question. In our case, we modified the PISA for use in Simple Scalar simulations. At the most basic level, this requires adding two instructions to the ISA: a begin loop instruction (LOOP) and an end of loop instruction (ENDL). Due to complications in modifying the simulation environment, we chose to use two instructions to handle the beginning and ending of the loop. Using one instruction is also possible, with a relative PC of the last instruction being included in the LOOP instruction or by detecting the branch instruction at the end of the loop.

LOOP signals the beginning of a loop and includes two pieces of information. The first is whether the source is an immediate or a register. The second is either the number of iterations of the loop as an immediate or the register where the number of iterations is stored.

These two cases correspond to static and semi-dynamic loops, respectively. After receiving a LOOP instruction with the register format, the machine is free to write over this register because the machine will have stored the value. The program counter will point to each instruction being executed during the loop. This will preserve correct behavior during context switching and interrupts. ENDL marks the end of the loop. From the LOOP instruction to the ENDL, the hardware will execute the intervening instructions the number of times specified in the LOOP instruction. When the number of iterations specified in LOOP has occurred, the program counter will proceed directly to the instruction after ENDL. Table 1 demonstrates a code section including a loop before and after the LOOP and ENDL instructions.

The transformation of the above loop illustrates the nature of our improvement's speedup. Each iteration 2 instructions are saved, with the actual benefit being contingent on a low latency between the end of the loop and the beginning. The speedup over the loop varies inversely to the length of the loop in cycles while the overall speedup depends on a large number of iterations. The greatest benefit from out modifications goes to applications dominated by short loops with little dependency between the end of the loop and beginning. These stalls between the end and the beginning of the loop can negate the benefits of eliminating the branch instructions, but the performance can never degrade. In a way, our instructions will grant the benefits of perfect branch prediction with less instruction overhead. There is no change in code size overall, assuming a two instruction branch and compare in most ISAs.

While it would be possible to have the same instruction begin and end the loop, this would prevent nesting of loops with these instructions.

To enable the hardware to better execute semi-dynamic and static loops, the compiler must collect certain information for the hardware.

The number of iterations that will be performed must be included as an immediate or in a register. This limits the maximum number of iterations to a relatively low 1024 iterations if the immediate is used. However, if the number of iterations is stored in a register, the iteration count is allowed to be the full numeric range of the register (2^32 in our case).

The loop register where the value is stored is the full 32 bits. Furthermore, the compiler must ensure that the loops enclosed by LOOP and ENDL are well-formed. The other requirement the compiler must enforce is the instruction limit on the length of the loop. In some cases, this will make in-lining functions beneficial because it will create a well-formed loop still within the instruction limit. While fixing a specific instruction limit into the ISA might make the instruction an undesirable artifact, this is unlikely because the savings are minimized by very long loops anyway.

Should a program attempt to buffer a loop that is not well-formed, that has too many instructions, or that has too many iterations, an interrupt will be generated.

The changes necessary to utilize these instructions in the compiler are not large. The register with the loop bound is easily determined and the additional instructions to calculate this are minor additions. The bookkeeping by the compiler to verify the correctness of the instruction usage is also not difficult. An important difference in this respect is the effort to hide the latency between the end of the loop and the beginning. However, the instructions can also be used alongside traditional interleaving/latency hiding techniques like loop unrolling because the ISA visible behavior is largely the same. Even if this latency cannot be hidden, the performance will not degrade. One complicating factor for the compiler is the tradeoff between spatial and temporal locality in determining things such as array accesses. In these cases, memory issues determine the way in which loops might be nested; but because the cost of poor locality is likely to outweigh the loop benefits, compiler algorithms to determine nesting can efficiently remain the same. The LOOP and END LOOP scheme does not allow for using the loop instructions and then branching into the middle of the loop to gain a different functionality. All looping

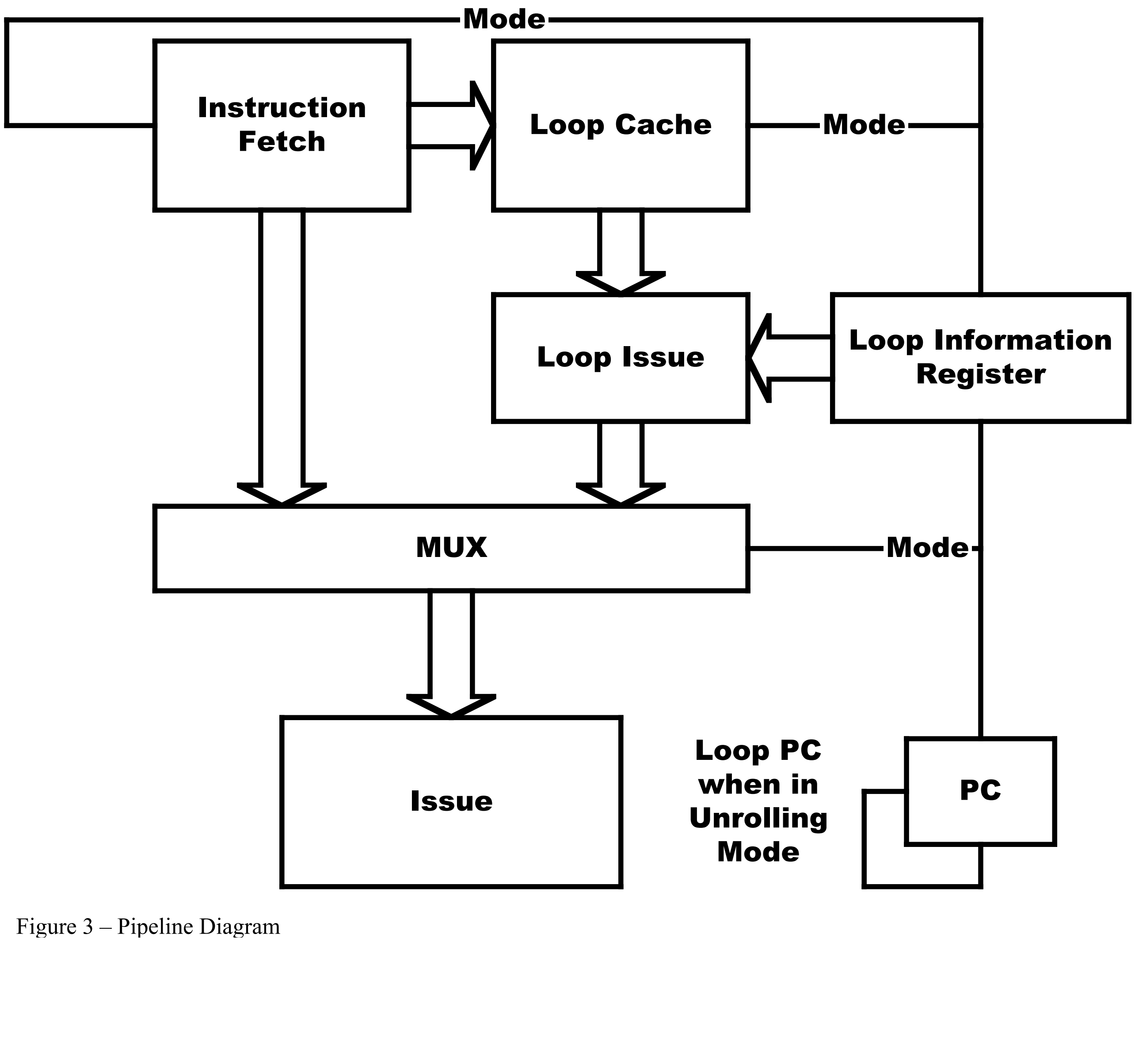

Figure 3 – Pipeline Diagram

must begin with a LOOP instruction.

The idea for instructions specifically to handle certain repetitive branches is not new. The Power PC ISA has a two branch conditional to count register instructions which branch to an incrementing counter based on certain conditions defined in the instruction parameters. This functionality can be used to handle case statements and loops, but the essential difference is that the instruction does not eliminate the branch instruction itself. The Itanium has a prefix to the opcode on branch instructions to pass hints to the processor; again, however, the branches are not eliminated and the hints are not guaranteed to be correct. Additionally, hints do not indicate when the end of the loop will be.

3. Hardware Additions

The hardware requirements for hardware loop buffering are relatively simple. A 32-bit LOOP register holds the contents of the loop register load. The LOOP instruction loads its immediate or the contents of the named register into the loop register. This loop register is accessed in order to determine the number of iterations in the loop. After receiving the loop register load instruction, a 64-bit x 1024 entry cache is loaded sequentially as the instructions execute until the END LOOP branch instruction is reached. The total number of entries in this cache is determined by the ISA and enforced by the compiler. An interrupt will be generated if this buffer overflows. The size of the cache in the SimpleScalar architecture is 8KB. While 8KB is large, there is less overhead than in regular caches because there are no tags, no addressing logic, and only one port. Valid bits are replaced by storing the last instruction in the buffer, and all subsequent instructions are known to be invalid. This cache will later be used to retrieve and duplicate instructions for subsequent executions of the loop. The instruction cache as present in most architectures would not be suitable for serving as a loop buffer because of the additional complexity, non-consecutive blocks, and evictions during context switches. Many of the specific functionalities needed

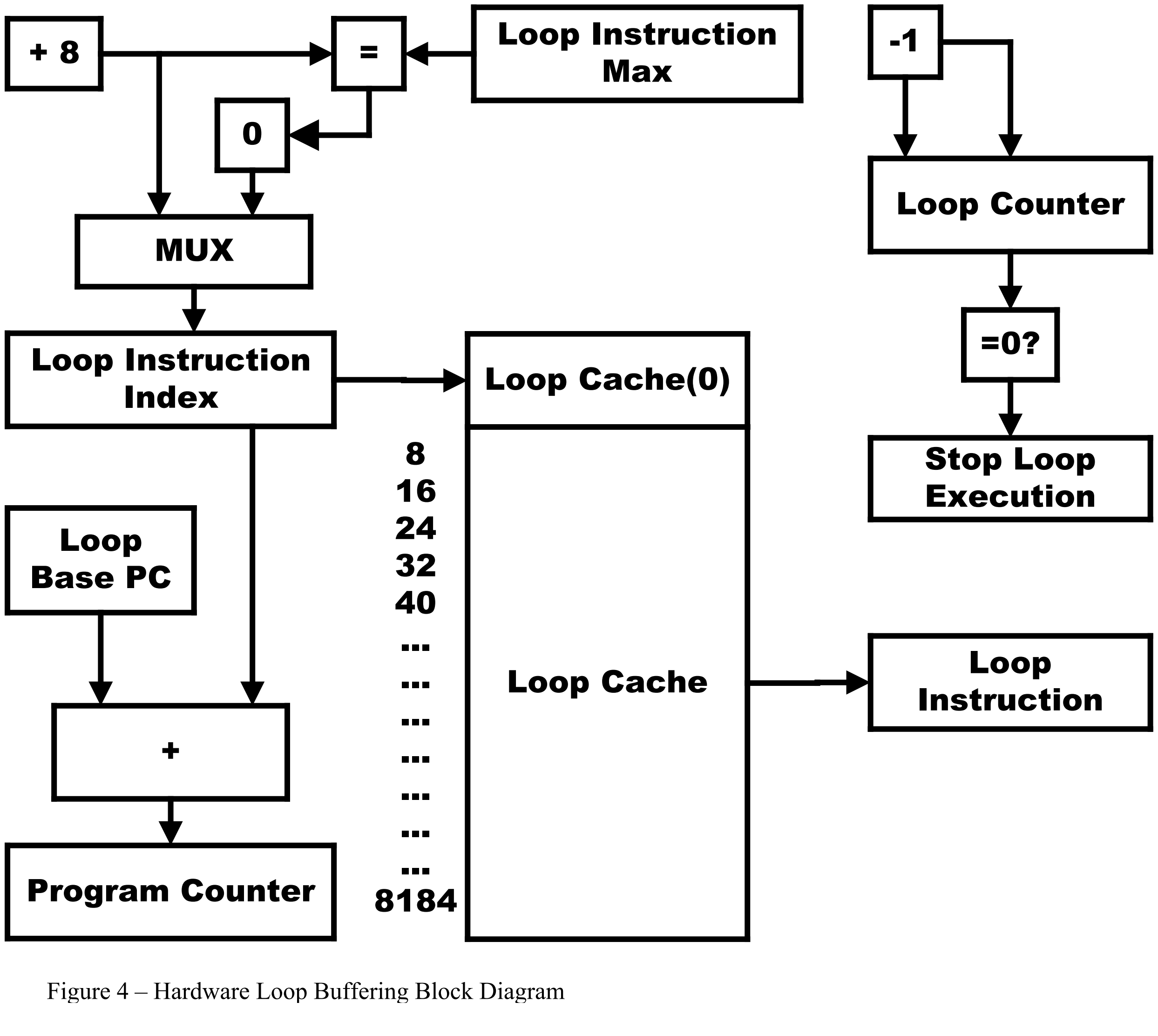

Figure 4 – Hardware Loop Buffering Block Diagram

are not present, and changing the instruction cache to meet the requirements would slow the cache down further.

Once the first iteration of the loop is completed, no instructions after the loop are fetched from the instruction cache until the loop execution is done. In future revisions this may be changed to allow for detection of the imminent loop termination, and the instruction fetch mechanisms will be allowed to work in parallel. This is consistent with the behavior if branches were still present, since Simple Scalar fetches instructions in a cycle until its maximum is reached or until a branch instruction is encountered. It then reads instructions sequentially out of the loop cache. The program counter will be incremented each cycle to point to the instruction currently being issued, and the current instruction to be fetched is referenced by a register that counts up to the loop buffer's size. When the current instruction equals the last instruction, the pointer will roll over. This system effectively makes the loop buffer the new source for instructions for the pipeline, and the structure acts in place of the regular instruction fetch mechanisms. Because the order of instructions is preserved, behavior remains the same. Each time the loop rolls over to the beginning of the buffer, the loop register is decremented. The instructions will repeat until the loop register is zero.

In the case of context switches and similar reloading of hardware, the system will need to move the entire loop buffer, loop register, loop buffer pointer, and loop buffer end pointer along with the rest of the context data to maintain state. This is a significant burden and in eventual implementation might require decreasing the size of the buffer.

Because the behavior of the loop is entirely governed by the additional structures, the branch prediction structures are bypassed. A side effect of focusing on loops with separate structures is that the branch predictor handles fewer loops, and it becomes less important to implement elaborate branch prediction schemes that track specific patterns. Branch target buffers, 2-bit branch prediction, and most branch prediction algorithms would still retain their utility. Part of our approach in handling loops has been to avoid high complexity algorithms and structures. This decision was made both because complexity itself is undesirable and because fully-dynamic loop unrolling schemes had already been discussed in [1]. In this paper, loops that cannot be unrolled by the compiler are targeted with a special buffer and executed speculatively. The affected loops are largely dynamic and non-well-formed. The highly complex system tracks multiple paths into the loop, registers modified, and loop predictions based on the paths taken with structures similar to a branch target buffer. We believe that this degree of size and complexity exceeds what is necessary to extract performance from the loop.

4. Performance Evaluation

4.1 Methodology

Evaluating the performance impact of our technique requires extensive analysis and modification to either the assembly or binary output of a benchmark. The algorithm to do these is of sufficient generality that it could be implemented in a compiler, at which point the compiler could choose to include instructions for our technique if the target architecture includes proper support. However, modifying the compiler to do this is outside the scope of this paper, since significant time constraints were present and modifying either gcc or SUIF would have occupied too large a percentage of the available work time.

Due to this limitation, analysis and the necessary modifications were performed manually. This restricted the possible benchmarks, since any benchmark chosen needed to possess sufficiently small code size to make conversion a manageable task. Additionally, the time needed to convert a single benchmark, regardless of size, was so large as to make it only feasible to do a performance analysis of one benchmark. To this end, 183.equake was chosen from the SPEC 2000 benchmark suite. It meets the requirements of small code size as well as having loop characteristics that are useful.

SimpleScalar was run with wrong path execution turned off. A known limitation of our architecture change is that once the LOOP instruction is run, the processor will continue storing instructions until the ENDL instruction is reached. If the LOOP instruction is reached during the wrong-path execution that occurs during a branch misprediction, then it is likely that the loop buffer will overflow. Also, the instructions that will be stored in the buffer are very likely to be the incorrect ones. Turning off wrong-path execution solves this problem.

For reasons that could not be discovered, a full simulation of a modified equake would not complete. Simple Scalar would throw a segmentation fault during execution. However, when the loops in equake were removed and put in there own C programs, each of them would run properly. Therefore, we profiled a representative sample of the loops and extrapolated their behavior to the rest of the benchmark. This is possible because there are only a few general types of loops in the program.

LOOP	Total Loop Cycle Count	Percentage of Loop Execution Time	Cycle Count Reduction	Speedup
1	907848	0.6	140100	1.18248175
2	1140	7.66E-04	100	1.09615385
3	21788352	14.6	3218126	1.17329493
4	907848	0.6	79602	1.09610913
5	126652	8.50E-02	19500	1.18198447
6	3934008	2.64	318409	1.08806535
7	3026160	2.03	241672	1.08679226
TOTALS	30692008	20.6	4017509	1.15

Table 2 - Results

For each profiled loop, runs with both modified and unmodified loops were done. The difference in cycle count between the runs is the number of cycles that are removed by the hardware buffering and perfect branch prediction for that loop.

4.2 Results

Table 2 presents the cycle count reductions measured for the loops we profiled. The last column is the speedup measured per loop.

The last item in the last column is a weighted average of the per-loop speedups. Because these loops are representative of the program as a whole, this is approximately the speedup that the loops in the program will have.

Referring to figure 2, the total time spent in the loops in equake is approximately 45% of the total execution time. Using Amdahl's law [2] we find that the total speedup (using a enhanced program fraction of 0.45) is 1.062. Therefore, a full run of equake will perform approximately 6.2% faster with the new architecture than without.

Since the other benchmarks for which we have analyzed the loop execution time have similar loop characteristics to equake, we expect that other benchmarks will show similar speedups to that shown for equake.

5. Future Work

Hardware Loop Buffering could be modified to allow simpler loops to be unrolled instead of just buffered. A simpler loop could be classified as a loop without inter-loop dependencies. If an inter-loop dependency is discovered, the hardware should proceed with loop buffering rather than loop unrolling. In order to successfully unroll the loop, it would be necessary to keep track of data dependencies, incremented values, and register allocation. The loop predictor must also be modified to allow for multiple iterations of the loop in flight at one time.

Unrolling the loop in a simple fashion would be to duplicate instructions and make the necessary adjustments to maintain correct loop function. For example, when marching an array, it would be necessary to increment the address for the load and/or store for each unrolled loop iteration, as well as the loop iteration number j. Any instruction that consumes j must be incremented by the value INC prior to consuming j. Likewise, any instruction that uses a value that is incremented each time through the loop must be adjusted accordingly. To accomplish this feat, the compiler must pass some additional information to the hardware. The compiler must not only give the hardware the register or value of the loop index and loop maximum, but also the value that the index is incremented by each iteration, INC. Other registers that are incremented every iteration of the loop must be identified while saving the instructions into the loop buffer. This is relatively difficult. However, they will usually have the form, Add/Sub $rt, $rs1, imm/$rs2. Not only do the instructions themselves need to be modified, the registers must be remapped such that no value from the first iteration is overwritten by the unrolled instructions.

Register allocation must also be considered when unrolling a loop. Normally, microprocessors have more physical registers than there are specified in the ISA. When duplicating instructions, the registers not used by the ISA will be used as unrolled iteration registers. The values held in these registers will be committed upon completion of each loop.

In order to determine how many loop iterations are left until the loop execution completes, the number of remaining iterations must be calculated during every unrolled batch. If i is the loop index, INC, the loop increment, U, the number of times that the loop is unrolled per loop batch, and MAX, the value held in the register for the maximum number of iterations, then MAX >= i + INC * U characterizes the check for whether or not to proceed unrolling. This calculation is done in parallel with the loop execution for each batch of instructions. This way, as soon as the check is detected to be false, the loop controller can adjust U, the number of times the loop is unrolled such that the loop will complete on the following batch execution. This method allows for perfect branch prediction in all loops that fall into the category that can be handled by the loop un-roller. This also allows the un-roller to bypass the loop branch, thus reducing the instruction count and in most machines resulting in a savings of the branch instruction, compare, and any assorted branch delay slots that could not be filled.

6. Conclusion

As has been shown in the results, hardware loop buffering can be an effective way to speed up loop-intensive code. By simply removing the branch and compare instructions in a loop, loop buffering provides a 6.2% speedup to execution. Compilers often do a good job of unrolling static loops, but are not able to unroll loops where the bounds are not known until run-time. This leaves a fair amount, 28.6%, of loops untouchable for the compiler. In addition to just buffering loops, hardware has the ability, with compiler hints, to perfectly predict loop branch behavior, thus removing the need for the branch and any other necessary compare instructions needed for branching.

Hardware loop buffering captures a large percentage of the loops that the compiler is incapable of unrolling without the large hardware overhead of fully-dynamic unrolling [1]. A few minor changes to the ISA allow hardware to take advantage of compiler hints, thus making the buffering possible without speculation. It is speculation and recovery that make full-dynamic hardware unrolling very complex.

Loop buffering does a good job of hiding latencies in single loop execution and in removing branch overheads, but can be limited in cases where it would be more optimal to schedule the duplicated instructions. Scheduling the instructions would require an analysis of the instructions while they are being stored and a method for picking and choosing which instructions to execute. The addition of this hardware would greatly increase the complexity and may or may not pay off in performance. However, future research into hardware loop unrolling rather than just buffering could lead to further performance improvements.

Hardware loop buffering has been shown to be a good way of speeding up loops in code without drastically increasing the complexity of the microprocessor. Buffering increases the observed IPC of both static and semi-dynamic loops in code. Given the results, it appears to be worth further pursuit.

References

[1] de Alba, M. and Kaeli D. *Characterization and Evaluation of Hardware Loop Unrolling* URL: http://www.ece.neu.edu/info/architecture/publications/ispass.pdf

[2] Hennessy J.L. and Patterson D.A. *Computer Architecture: A Quantitative Approach* Morgan Kaufman, 3rd edition