

Characterization and Evaluation of Hardware Loop Unrolling

Marcos R. de Alba

David R. Kaeli

Department of Electrical and Computer Engineering

Northeastern University

Boston, MA, 02115 USA

mdealba,kaeli@ece.neu.edu

Abstract

General purpose programs contain loops that cannot be optimized by a compiler. When the body of a loop contains conditional control flow instructions or if the loop is controlled by a non-constant induction variable, the compiler again cannot unroll this loop. We have found that the compiler cannot unroll greater than 40-50% of the static loops in the set of programs studied. To be able to optimize the execution these loops, we have to detect loop behavior at runtime.

We propose that loops that cannot be optimized by the compiler should be detected and unrolled using a hardware-based unrolling mechanism. Our design exploits the temporal locality found in loops to provide a higher degree of instruction level parallelism. Using hardware-based unrolling, multiple basic blocks can be retrieved from a dedicated loop cache, reducing the number of instruction cache and memory requests, while providing a large window of instructions for speculative execution.

Before designing our hardware mechanism, we characterized all loops that cannot be optimized by the compiler. Using these characteristics we construct the design of a hardware mechanism that will allow us to unroll loop iterations dynamically. To drive our prediction mechanism, we use correlation between the pattern of branch outcomes that lead up to a loop with the path of branches executed within the loop body. We capture a history of the sequence of paths followed in a loop to predict the entire loop visit. We can then unroll entire loop bodies without the aid of the compiler.

To characterize loop execution and evaluate the effectiveness of the proposed mechanisms, we study three different sets of benchmarks: mediabench, mibench and a subset of SPECint2000 (the loop intensive benchmarks). Our results show that hardware-based loop unrolling can be performed dynamically and provides us with new levels of instruction-level parallelism. We have found that we can consistently increase the IPC using this mechanism, achieving maximum speedups greater than 20%.

1 Introduction

Most programs spend a large percentage of their execution time in a small portion of their code (commonly known as the 90/10 rule). Therefore, to impact the performance of a program, we should focus our attention on these frequently executed regions. Most of this code is located within loop constructs.

Compiler-controlled loop unrolling can be used to unroll simple loops, but unrolling is generally not performed if any one of the following issues is present in the loop body:

- the loop body is large,
- the loop induction variable is not an integer,
- the loop induction variable is not incremented by 1,
- the increment value cannot be deduced at compile time,
- the initial loop induction variable value is not a constant,
- the loop exit condition value is not based on a constant,
- the loop contains conditional control flow, or
- there exist multiple entry points to the loop body.

We have found that for the multimedia, embedded and general purpose applications studied, approximately 40-50% of all loops could not be unrolled for one of these reasons. Table 1 shows the number of dynamic loops executed that contained conditional control flow instructions.

In this work, we propose and evaluate mechanisms for predicting the entry into a loop, as well as unrolling the path through the loop, using the dynamic history of a loop's execution. We base our design

Table 1. Percentage of all loops executed which contain conditional branches in their bodies.

| Benchmark | % loops with cond. branches in their body |
|-----------|---|
| gzip | 59.67 |
| bzip2 | 58.33 |
| gcc | 71.94 |
| parser | 64.89 |
| twolf | 67.81 |
| CRC32 | 69.56 |
| FFT | 81.81 |
| Dijkstra | 78.57 |
| epic | 79.54 |
| g721 | 57.69 |
| jpeg | 56.25 |
| patricia | 84.37 |
| typeset | 76.55 |

Table 2. Benchmarks information.

| Suite | Name | Inputs | Dyn Instrs | Description |
|------------|----------|-------------------|------------|---|
| mibench | dijkstra | input.dat | 254M | Dijkstra's shortest path algorithm |
| mibench | CRC32 | large.pcm | 692M | Cyclic Redundancy Check 32 |
| mibench | patricia | large.udp | 481M | A Patricia trie data structure |
| mibench | FFT | large.pcm | 249M | Fast Fourier Transform computation |
| mibench | typeset | large.lout | 84M | Typesetting tool with a front-end processor |
| mediabench | epic | test.image.pgm.E | 10.8M | Experimental image data compression utility |
| mediabench | g721 | clinton.pcm | 420M | G.711, G.721 and G.723 voice compressions |
| mediabench | jpeg | testing.ppm | 16.5M | JPEG image compression and decompression |
| spec2000 | gzip | smred.source | 2.1B | Gzip data compression algorithm |
| spec2000 | gcc | smred.c-iterate.i | 134MB | Gcc compiler |
| spec2000 | bzip2 | lgred.source | 2.1B | Bzip2 data compression algorithm |
| spec2000 | parser | smred.in | 449M | Word processor |
| spec2000 | twolf | test | 500M | Place and Route Simulator |

on the observed dynamic characteristics of loop execution [1]. We focus on loops that cannot be unrolled by the compiler. The proposed mechanism can be incorporated into a superscalar microarchitecture without modifying the underlying instruction set.

Our proposed microarchitectural features are evaluated for multimedia, embedded and general purpose applications. The selected applications possess differences in:

- the number and size of loops,
- the loop body control flow structure, and
- the maximum loop nest depth.

Table 2 shows the programs selected from mediabench [8], mibench [6], and the SPECint2000 [14] benchmark

Table 3. Loop-related characteristics for the 3 benchmark suites. Column 2 shows the number of static loops, column 3 shows the average basic block size, columns 4-5 show the breakdown of loops terminating in a conditional backward branch versus those terminating in an unconditional backward branch, and column 6 shows the maximum loop nest. Columns 3-6 are based on dynamic (runtime) characteristics.

| Benchmark | Number of Loops | Average Basic Block Size | Percent of Loops Terminating with BkwdCond | Percent of Loops Terminating with BkwdUncond | Max. NL |
|-----------|-----------------|--------------------------|--|--|---------|
| gzip | 112 | 10.00 | 54.46 | 45.54 | 4 |
| gcc | 1219 | 6.35 | 8.12 | 91.88 | 7 |
| bzip2 | 75 | 12.19 | 29.33 | 70.67 | 6 |
| parser | 306 | 6.06 | 11.11 | 88.89 | 6 |
| twolf | 241 | 7.93 | 19.50 | 80.50 | 4 |
| CRC32 | 21 | 8.66 | 61.90 | 38.10 | 1 |
| FFT | 39 | 10.19 | 43.59 | 56.41 | 2 |
| Dijkstra | 39 | 5.37 | 71.79 | 28.21 | 3 |
| epic | 76 | 9.48 | 80.26 | 19.74 | 2 |
| g721 | 26 | 8.53 | 84.62 | 15.38 | 2 |
| jpeg | 102 | 18.58 | 92.16 | 7.84 | 5 |
| patricia | 62 | 5.54 | 74.19 | 25.81 | 3 |
| typeset | 80 | 7.09 | 81.25 | 18.75 | 2 |

suites. Table 3 presents a number of characteristics for the set of applications, including:

- the number of loop bodies,
- average basic block size (for the entire application),
- percent of dynamic loops terminating with backward conditional branches,
- percent of dynamic loops terminating with backward unconditional branches, and
- maximum loop nest depth.

Next we review past work on loop prediction techniques.

2 Related Work

One goal of an aggressive fetch mechanism is to speculate across multiple branches during a single fetch cycle. Some previously proposed multi-branch prediction mechanisms include multiple prediction using a branch address cache [17], the trace cache [11], the branch-tree predictor [3] and the multiple-block ahead predictor [13]. None of these mechanisms specifically focused on the branches contained in loops.

The first study that presented the dynamic characteristics of loops was presented by Kobayashi [7]. A more recent study presented a thorough examination of the dynamic characteristics of loops [2].

Hardware-based loop caching has been used in embedded systems [5]. Modern DSP processor architects have investigated using a loop cache to implement hardware loop unrolling for *well-structured loops* [9, 12, 15] (remember that a lot of software written for DSPs and embedded systems is written in assembly, so hand unrolling is performed). To manage loops, the DSP processors have incorporated special loop flow control instructions. The main limitation with these mechanisms is that the size of the loop buffer is many times too small to hold the entire loop and that this buffer only caches well-structured loops.

In multi-threaded processors, loop locality can be exploited by executing several independent iterations of a loop on different threads [4, 16]. The techniques described by Tubella and Gonzalez, which allow for the detection of a loop entry, are used in our work.

In our proposed mechanism we focus on aggressively fetching instructions contained in loops. The originality in our work lies in our ability to predict multiple branches contained within loop bodies using a *loop prediction table*, caching the associated basic blocks in a *loop cache*, and aggressively delivering these instructions to the instruction window. This should help to expose larger degrees of instruction level parallelism that can only be found across multiple basic blocks and multiple iterations of a loop.

The rest of this paper is organized as follows. Section 3 presents loop terminology in order to facilitate the description of our loop prediction hardware. We also describe how to construct pattern states to record history of loop visit execution. Section 4 presents the design of the loop predictor and loop caching mechanisms. Section 5 presents simulation results using the ATOM and SimpleScalar frameworks, and Section 6 summarizes the contributions of this paper.

3 Loop Taxonomy

Next we present a loop taxonomy that can be used to describe both static and dynamic loop behavior. *Loops* occur in program flow whenever control resumes at a negative displacement. Programmers utilize loops to carry out iterative algorithms or repetitive actions, looping multiple times to carry out the desired operation.

We define the following terms to describe the static elements of a loop:

- *loop head* - the first instruction in the loop

- *loop tail* - the branch possessing a negative displacement that marks the end of the loop
- *loop body* - the instructions within the loop

We also define the following terms to describe the dynamic elements of a loop:

- *path-to-loop* - pattern of conditional branch outcomes leading up to a loop body
- *loop visit* - entering a loop, executing some number of loop iterations and then exiting the loop body
- *loop iteration* - one execution of a path through the loop body, starting from the loop head and ending at the loop tail
- *path-in-iteration* - pattern of conditional branch outcomes in a single loop iteration
- *path-in-loop* - pattern of conditional branch outcomes during an entire loop visit

We also define some general characteristics of loop bodies:

- *well-structured loops* - loops that are void of conditional control flow instructions in their bodies (except for the loop tail branch),
- *ill-structured loops* - loops that contain conditional control flow inside their loop bodies
- *fixed-count loops* - loops that iterate based on the value of a constant induction variable whose value can be deduced at compile time
- *variable-count loops* - loops that can iterate a variable number of times

We will utilize these terms to help explain how we can exploit the path-to-loop behavior to predict and unroll loop bodies.

From previous work [1] we observed that most loops iterate a predictable number of times during each visit and that many loops follow repeatable patterns during each iteration, so we believe we should try to predict entire loop visit execution using past history. We are primarily interested in predicting ill-structured and non-constant induction variable loops, though our mechanism will also work well for any loops that were not completely unrolled by the compiler.

When a loop is visited, different path-in-loop patterns can be produced. To build a history of path-in-loop patterns, we first record a history of path-in-iteration patterns by collecting the outcome of the set of n branch directions (taken or not taken) that occur during an iteration. When the loop iterates, and if the pattern repeats, we increment a counter to indicate that this pattern repeats. If a new path-in-iteration pattern appears, we record this pattern and record the number of times this pattern repeats. In essence, we capture the set of path-in-iteration states that appear during a single loop visit. We then use this history to predict future loop visit behavior.

Past research has shown that conditional branch behavior tends to be correlated [10]. Our own past work has also shown that the behavior of conditional branches leading up to a loop body (the path-to-loop) is a good indicator of the path-in-loop pattern that will be followed [1]. With this in mind, we propose to predict and unroll entire loop executions based on path-to-loop branch behavior.

4 Loop Prediction Hardware

Loop prediction involves four basic hardware mechanisms. Next we will describe each of these.

4.1 Path-to-Loop

In order to record (and to later use as an index) the path-to-loop associated with a loop visit, we use a shift register that holds conditional branch outcomes (similar to a BTB pattern register) and we also record the address of the last conditional branch executed prior to entering the loop body. We show our path-to-loop hardware in Figure 1. Note that we also choose to save the branch instruction address associated with each branch. Branch b_0 is the branch that controls entering the loop and a_0 is its associated instruction address.

We use the occurrence of a backward branch in the loop body to detect the tail of a loop. After the first loop iteration is completed, the address of the loop head and loop tail are pushed onto the *loop stack*. We use a stack-based structure to properly identify nested loop behavior. We utilize an 8-entry loop stack, which captures all the loop nests in the set of benchmarks studied.

Once we know we are in a loop body, we need to truncate the path-to-loop pattern register to form the index using the true path-to-loop branch outcomes, and remove the path-in-loop outcomes that were shifted into the path-to-loop pattern register. By keeping the branch addresses associated with the path-

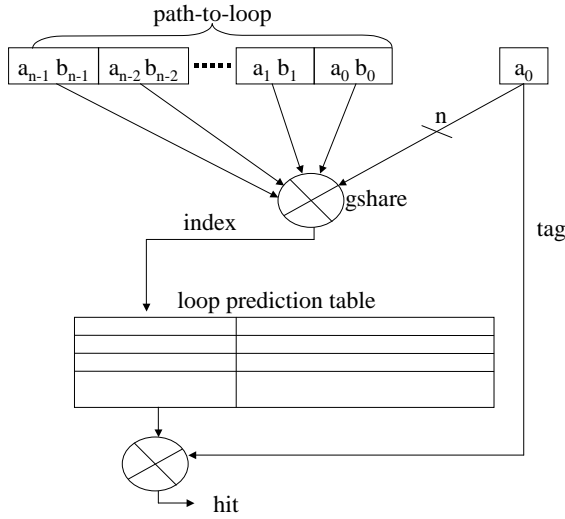


Figure 1. Path-to-loop hardware. Both the branch instruction address and branch outcomes are retained. Only the branch outcomes are XOR'ed with the address for b_0 .

to-loop pattern register, we can identify those branches that are truly in the path-to-loop. The pattern that was filtered out consisting of the branches executed in the initial iteration of this loop are saved in the path-in-iteration table. We describe this table next.

4.2 Path-in-Iteration Table

The path-in-iteration table records state transitions through the sequence of unique path-in-loop patterns generated in a loop visit. When the path-in-iteration repeats, an iteration count associated with this path-in-iteration pattern is incremented. Figure 2 shows a path-in-iteration table design. In the design considered in this paper, we use a 16-bit wide path-in-iteration pattern field, accompanied with an 8-bit wide counter value. If a pattern is repeated more than 256 times, we form an additional record in the path-in-iteration table. The table assumed in this paper contains 16 pattern entries per loop visit. While this suggests a table on the order of 48 Kbytes, we did not want to constrain this parameter. We are more interested in establishing the potential of using such a table versus trying to find the best table size.

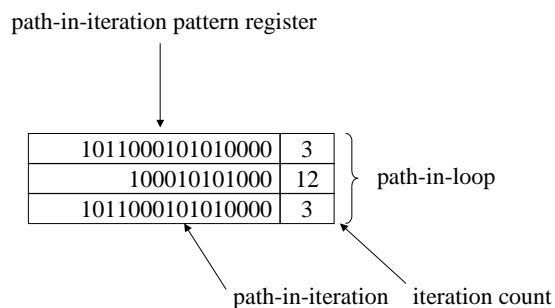


Figure 2. Path-in-iteration hardware used to capture the state transitions and iteration counts of different path-in-loop patterns.

4.3 Loop Prediction Table

The loop prediction table stores loop identification information, the number of predicted iterations, the address of the last branch in the path-to-loop (to be used as a tag) and an index into the path-in-iteration table. We model a 1024-entry, 4-way set associative, loop prediction table. Associativity is important to allow nested loops to live in the loop prediction table simultaneously. Figure 3 shows the organization of a loop prediction table.

4.4 Loop Cache

Each loop is profiled using the loop stack and once the loop visit finishes, the loop prediction table and path-in-iteration table are updated. The loop body is also written into the loop cache. This effectively unrolls the loop.

To unroll a loop in hardware, the loop prediction table and path-in-iteration table provide both the indexing into the loop cache and the basic blocks that need to be unrolled. Figure 4 shows the mechanism to perform hardware loop unrolling via loop prediction. The first time a loop executes, we will need to

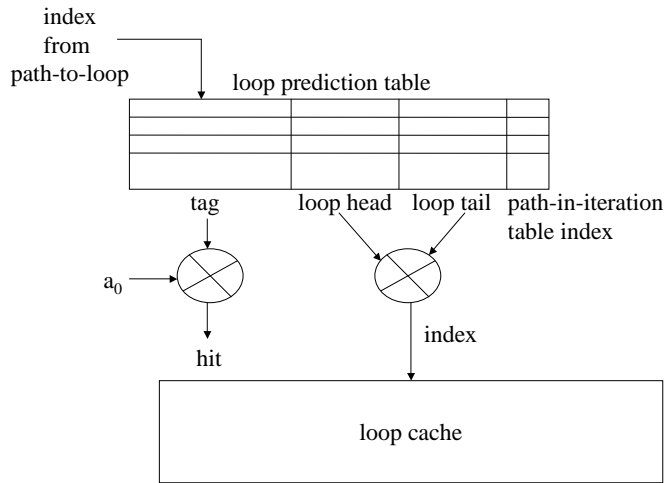


Figure 3. Loop prediction table entries used to index into the loop cache. Fields in the loop prediction table include: a tag field, the loop head and tail address, and the index into the path-in-iteration table.

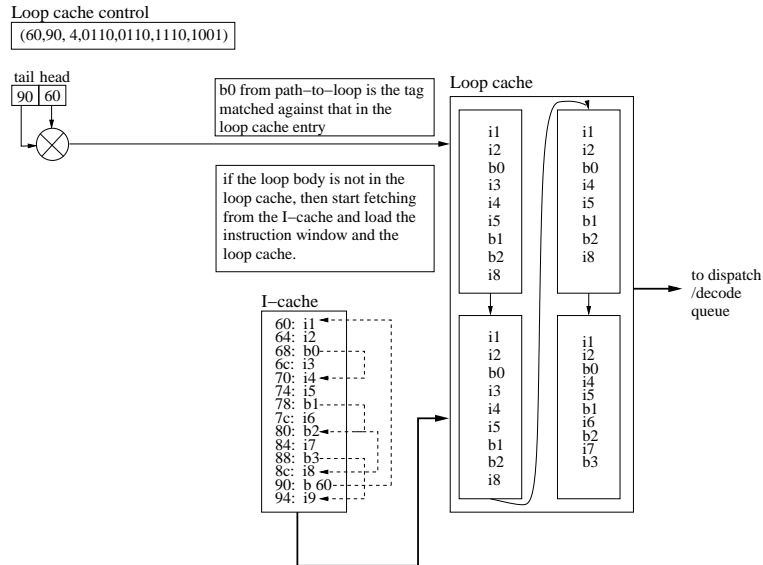


Figure 4. Unrolling a loop body in the loop cache.

Table 4. Loop Prediction Hardware Design Parameters.

| Parameter | Size |
|--|--------------|
| Loop prediction table size | 1024 entries |
| Loop prediction table associativity | 4 |
| Loop stack size | 8 entries |
| Length of path-to-loop pattern register | 16 branches |
| Loop cache size | 256 entries |
| Max number of branches in paths-in-iteration | 16 |

request instructions contained in the loop body from the I-cache. As the instruction window is being filled, we also allocate entries in the loop cache. Then future accesses to this loop that possess the same path-to-loop pattern can obtain all instructions directly from the loop cache. For loops that contain function calls, we inline the called function in the loop cache. We also determine whether the body of the loop is too large before allocating an entry in the loop cache. This prevents a single branch from dominating the loop cache space. Fortunately, the hottest program loops tend to be short loops.

When the indexed entry is present in the loop cache, the loop will unroll the loop according to the path-in-iteration information provided, which includes both the direction of each conditional branch, as well as the iteration count for each pattern.

The hardware parameters used in this paper are listed in Table 4.

5 Evaluation

Next we present prediction accuracy, instruction delivery efficiency, and overall performance assessment for our loop unrolling mechanisms.

5.1 Simulation Environment

We used the ATOM tool to obtain the branch profile data presented in Tables 2 and 3. We used the SimpleScalar toolset to study our loop prediction and unrolling mechanism integrated into a superscalar microarchitecture. We added our loop prediction table, loop cache, loop stack and hardware unrolling mechanism to the Alpha EV6 version of the SimpleScalar 3.0b. The configuration of the superscalar

Table 5. Main configuration of the superscalar processor used in this work.

| Parameter | Size/Units/Latency |
|-------------|---|
| L1 Dcache | 32 kB, 64 blocks, 4-way, 1 cycle hit lat. |
| L1 Icache | 32 kB, 32 blocks, direct-mapped, 1 cycle. |
| L2 Dcache | 256 kB, 128 blocks, 4-way, 10 cycle. |
| L2 Icache | 256 kB, 128 blocks, 4-way, 10 cycle. |
| Memory | 250 cycle hit lat., 2 read/write ports |
| Exec. Units | 4 IALUs, 4FPALUs, 2 Mul/div |

processor core used for the simulations is presented in Table 5.

All the benchmarks were compiled using the DEC C V5.2-033 compiler hosted on Digital UNIX V4.0 (Rev. 564). Programs were compiled using the compilation flags: *-non-shared -02*. Using these compilation flags, loop unrolling is aggressively performed by the compiler.

In the next section we review some interesting statistics associated with our prediction scheme, including:

- the accuracy of predicting the number of loop iterations for the entire loop visit,
- the accuracy of predicting the path-in-iteration per loop iteration,
- the percentage of all retired instructions that were issued from the loop cache,
- the gain in instruction dispatch rate, and
- and the overall gain in IPC when using a loop cache.

5.2 Evaluating Loop Prediction

All predictions are performed before the head of the loop is fetched. The update of the loop prediction table takes place when the loop terminates execution. At termination time, the collected information for the current loop execution is evaluated against the predicted values and paths. If the information is different, the table entry is updated.

Figure 5 shows the prediction rate of the number of iterations per entered loop visit. Most loops tend to follow a repeatable pattern in the number of iterations they perform per visit, although a small number of loops exhibit a more difficult to predict pattern of iteration counts. We use our confidence mechanism to turn off predictions with the loop predictor. We can see for many of the benchmarks, we

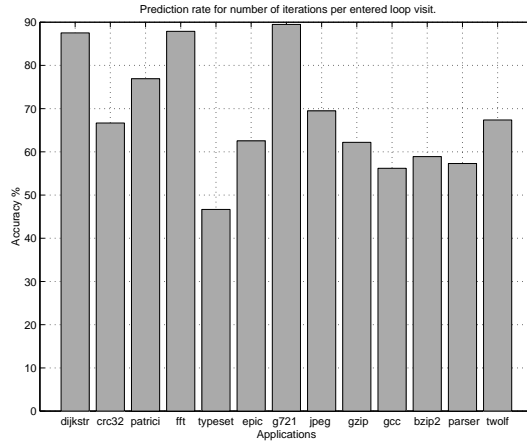


Figure 5. Prediction of number of iterations per entered loop visit.

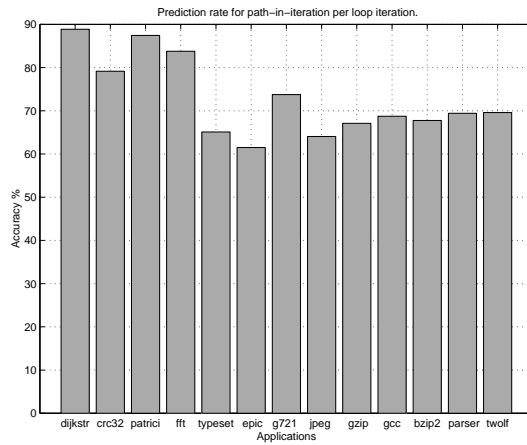


Figure 6. Prediction of paths-in-iteration per loop iteration.

can predict the frequency count of loops for more than 50% of the loop visits, and for a few applications almost 90% of the visits. When we cannot predict the number of iterations accurately, we can pay a significant penalty since many speculative instructions may need to be squashed.

Figure 6 shows our ability to predict the path followed on a single iteration of a loop. The predictability of the number of loop iterations for the entire loop visit varies across applications. We see that all benchmarks have prediction accuracies above 60%. Those that reached accuracies above 80% have a small number of loops with relatively small body sizes and a small number of branch instructions within their bodies. These characteristics generally indicate better predictability.

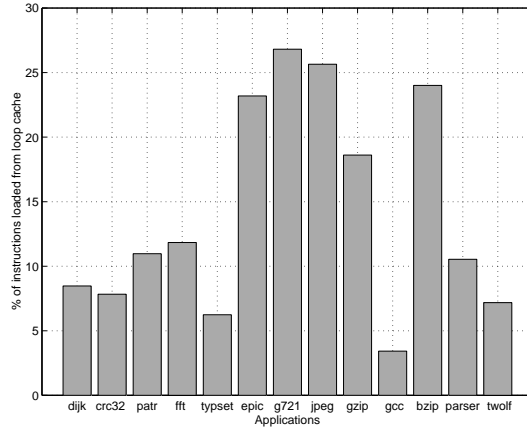


Figure 7. Percentage of instructions issued from loop cache.

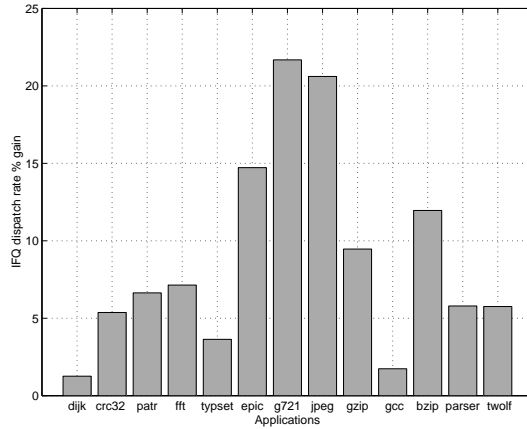


Figure 8. Percentage of instruction fetch queue dispatch rate gain.

By capturing loop bodies in a dedicated loop cache, the fetch unit reduces the number of I-cache requests performed. In addition, when loop iterations are independent and there are enough available execution resources, entire loop bodies can be speculatively executed. Figure 7 shows the percentage of instructions that are issued from the loop cache instead of from the L1 I-cache. Figure 8 shows the increase in the instruction fetch queue dispatch rate due to issuing instructions from the loop cache.

5.3 Impact on Instruction Throughput

We modeled our loop predictor and loop cache with SimpleScalar and compared our hardware predictor against a design that does not perform any hardware-based unrolling. In Figure 9 we show the gains in

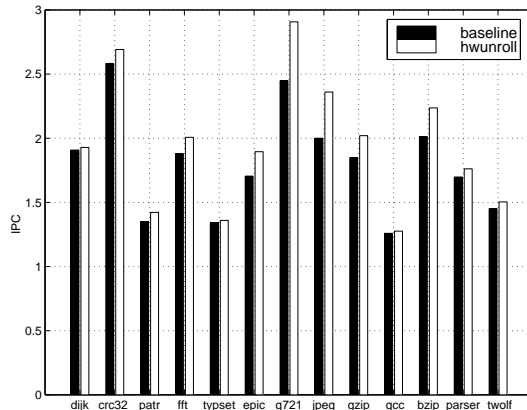


Figure 9. Performance gain produced by the hardware unrolling mechanisms.

IPC obtained by using a loop cache with the current architecture. We believe these gains are impressive, though they require a significant amount of hardware investment to obtain. We also believe we have just begun to fully explore the design space for this set of mechanisms.

In the results, the application g721 obtains the largest speedup for the set of programs. The characteristics for this program include: the average number of instructions executed per loop iteration is 26.5 (the largest in the 3 suites) and the iteration count predictability for the entire visit is close to 90%. Also, the predictability of the path-in-iteration pattern is around 75%. All of these features lead to large improvements in instruction fetch rates and IPC gains.

5.4 Discussion

There are some areas in this work that need to be studied further. Presently we fully model the impact of misspeculation with the loop predictor, paying the penalty for squashing instructions and reissuing instructions from the instruction caches on the correct path. But we believe we can develop better confidence-based schemes that can reduce the frequency of squashes.

The benefits obtained with the proposed loop predictor can be further improved by providing runtime adjusted strategies. The path-in-iteration behavior changes so often that the loop predictor cannot capture the changes as rapidly as they occur. A more elaborate hybrid prediction mechanism would be able to detect this behavior more rapidly and the overall speedup could be improved.

The loop predictor and unrolling hardware exploit temporal locality and provide higher levels of

parallelism. Using these mechanisms, the effectiveness of the fetch unit is increased and a larger number of instructions are available for speculation. The number of L1 I-cache accesses are reduced due to the reuse of loops allocated in the loop cache. We should consider a design that trades off I-cache space for loop-cache space dynamically.

One other area we are interested in pursuing is a hardware/software approach which combines compiler-based loop unrolling with knowledge of a hardware-based loop predictor. If a loop is unrolled too aggressively by the compiler, this can consume too much space in the loop cache. If we unroll simple loops just enough times to utilize software pipelining effectively, then we can still unroll the loop further in the loop cache, obtaining benefits from both approaches.

6 Summary

In this work we described a number of hardware mechanisms used to unroll and speculate entire loop executions at runtime. Our results show that hardware-based loop prediction can have a significant impact on performance, consistently improving performance on all programs studied.

Our next steps in this work look to reduce storage space requirements and develop a trace cache-like loop cache. We believe we can obtain greater amounts of IPC while reducing the complexity of our design.

References

- [1] M. R. de Alba and D. R. Kaeli. Runtime Predictability of Loops. In IEEE Computer Society, editor, *Proc. of the 4th Annual IEEE International Workshop on Workload Characterization*, pages 91–98, Austin, TX, December 2001.
- [2] M. R. de Alba, D. R. Kaeli, and E. S. Kim. Dynamic Analysis of Loops. In *Proc. of the 3rd International Conference on Control, Virtual Instrumentation and Digital Systems*, pages 93–106, Mexico City, August 27-31 2001.
- [3] S. Dutta and M. Franklin. Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors. In *Proc. of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–263, Ann Arbor, Michigan, 1995.

- [4] A. Gonzalez and P. Marcuello. Dependence Speculative Multithreaded Architecture. Technical report, Universitat Polytechnica de Catalunya, 1998.
- [5] A. Gordon-Ross, S. Cotterell, and F. Vahid. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. In *Computer Architecture Letters*, volume 1. IEEE Computer Society, January 2002.
- [6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *4th IEEE International Workshop on Workload Characterization*, pages 3–14, Austin, TX, December 2001.
- [7] M. Kobayashi. Dynamic Characteristics of Loops. *IEEE Transactions on Computers*, 33(2):125–132, 1984.
- [8] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. of the 30th International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, North Carolina, 1997.
- [9] Motorola. *SC140 DSP Core Reference Manual*, April 2001.
- [10] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating System*, pages 76–84, 1992.
- [11] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proc. of the 29th International Symposium on Microarchitecture*, pages 24–35, Paris, France, 1996.
- [12] S. Sair and D. Kaeli. A Study of Loop Unrolling for VLIW-Based DSP Processor. In *Proc. of the Workshop on Signal Processing Systems*, pages 519–527, Cambridge, MA, October 1998.
- [13] A. Sez nec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-Block Ahead Branch Predictors. In *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, Cambridge, MA, 1996.
- [14] SPEC. SPEC CPU 2000 Benchmark Set Program. <http://www.spec.org/osg/cpu2000/>.

- [15] Texas Instruments. *TMS320C62xx DSP Family*, 1998.
- [16] J. Tubella and A. Gonzalez. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In *Proc. of the 4th International Symposium on High-Performance Computer Architecture*, Las Vegas, NV, January 1998.
- [17] T-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In *Proc. of the International Conference on Supercomputing*, pages 67–76, Tokyo, Japan, July 1993.