Aggressive Dynamic Execution of Decoded Traces

Benjamin Bishop Robert Owens Mary Jane Irwin
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA

Abstract - In this paper, we consider the increased performance that can be obtained by using in concert, three previously proposed (and in two cases used in commercial systems) ideas. These ideas are aggressive dynamic (run time) instruction scheduling, reuse of decoded instructions, and trace scheduling. We show that these ideas complement and support one another. Hence, while each of these ideas has been shown to have merit in it's own right, when used in concert, we claim the overall advantage is greater than that obtained by using each singly. To support this claim, we present the results from running several common multimedia kernels. Overall, these results show an average speedup of 3.50 times what can be had by using dynamic instruction scheduling alone.

1. INTRODUCTION

Efficient use of functional units and, hence, high instruction execution rates can be obtained by using the dynamic scheduling of the instructions in some pool. Instructions can be added to this pool at a high rate by using a trace cache. The cost (in terms of latency, power, and hardware) of adding instructions to the pool can be greatly reduced by reusing already decoded instructions. Hence, when used in concert, dynamic instruction scheduling, trace caching, and decoded instruction reuse provide for high execution rates. This is accomplished by maintaining a high rate and efficiency by which instructions can both enter and exit the pool. Furthermore, we show the marriage of these ideas to be particularly useful in the execution of signal processing kernels related to multimedia applications.

In section 2, we present background information reviewing the three abovementioned techniques. In section 3, we discuss the performance issues effecting the design. In sections 4-6, we introduce the simulator and interpret our results.

2. BACKGROUND

In this section, aggressive dynamic instruction scheduling, reuse of decoded instructions, and trace scheduling are briefly described.

2.1 Dynamic Execution. Dynamic (run time) instruction scheduling is very useful for exposing parallelism, which in turn can be used to increase performance. For this reason some type of dynamic instruction scheduling is employed by many processors. Dataflow processors, where any instruction is eligible for execution as soon as the values for it's operand(s) is known, achieve the highest degree of parallelism. However, the implementation of a dataflow processor requires that a time efficient way to schedule a possibly very large number of eligible instructions on the available functional units must be found. This, in and of itself, has proved to be a very difficult problem to solve.

What has been called restrictive dataflow [2, 3, 12] can be viewed as a compromise between completely static and completely dynamic scheduling. Conceptually, in restrictive dataflow, the instructions of a statically scheduled instruction stream are first decoded and then added to a pool of now dynamically schedulable instructions. This pool is kept to a manageable size by limiting the number of instructions that it can contain at any one time. For this reason, the problem of scheduling is much easier than for completely dynamic scheduling.

Register renaming is often used as part of the translation process to remove name dependencies. As instructions become eligible, they are scheduled on the available functions units. To preserve the outward appearance of sequential execution, results are committed to memory and to a set of retire registers in order as the dynamic instructions complete.

2.2 Decoded Instruction Reuse. VLIW and wide-issue superscalar processors require substantial instruction fetch and decode bandwidth to achieve high utilization of the functional units. To help alleviate this problem, it is possible to reuse the decoded instructions instead of having to redecode an instruction every time it is executed [4]. Hence, instead of having a pool of encoded instructions, a pool of decoded instructions is utilized. On average, the decode efficiency (in terms of latency and hardware cost) can thus be increased because instructions coming from the decoded instruction pool require little, if any, additional decoding.

2.3 Trace Cache. The trace cache technique, proposed by Rotenberg et al. [6], was created to address the instruction fetch bandwidth bottleneck. This is done by organizing stored instructions in an on-chip trace cache according to actual run-time control flow. Thus, multiple basic blocks can be fetched from the trace cache per cycle that would be non-contiguous in a typical instruction cache.

The trace cache is organized into a series of traces. Each trace structure contains a tag, branch target, and branch fall-through fields as well as branch prediction information. During instruction fetch, two conditions must be met in order to fetch from the trace cache. The fetch address must match the tag associated with one of the currently held traces. The branch predictions must also match those stored in the trace structure. Thus, an instruction trace stored in the trace cache may contain multiple branches that can be fetched in one cycle.

The advantages of this scheme are that a multiported cache is not required, less bookkeeping information is necessary since only one cache line has to be fetched, and there is no need to shift and align instructions since they are held on the same cache line. Therefore, on a trace cache hit, parallelism is not limited by the ability to fetch instructions.

3. PERFORMANCE ISSUES

Modern superscalar processors like the HP PA 8000, IBM/Motorola PowerPC 604, Intel Pentium Pro, and SGI/MIPS R10000 employ restrictive dataflow to expose run time parallelism and, in turn, increase performance through better functional unit utilization. However, substantial instruction fetch and decode bandwidth is also required to achieve high utilization of the functional units. In fact, in processors using aggressive dynamic scheduling, performance may be limited not by the effective bandwidth of functional units but instead by the fetch and decode bandwidth. For example, while the Intel Pentium Pro [1] can issue five microoperations per cycle, it can (at best) produce only three from the fetch and decode unit.

To help increase the effective fetch and decode bandwidth and to decrease it's latency, the reuse of decoded instructions has been proposed [4, 3]. The use of this technique has seen it's greatest promise in the context of a CISC instruction set architecture, ISA, where the decode process is, of course, much more complex than it is for a RISC ISA. However, in Hiraki et al.[5] as well as in this paper, the merits (in terms of either reduced latency, power, or instruction issue complexity) of reusing decoded instructions in RISC ISA are likewise shown.

For many applications, even with a high effective fetch and decode bandwidth and aggressive dynamic scheduling, maintaining a high instruction throughput would be difficult without branch prediction. This difficulty occurs because, until a conditional branch dependency is resolved, the pool of dynamically schedulable instructions empties. This in itself causes a problem, since the ability to maintain the high utilization of the functional units is dependent on having sufficient parallelism, and parallelism tends to decrease as the pool decreases. Also, when the branch dependency is resolved, the pool must be refilled. The rate at which it can be refilled is, of course, limited by the rate at which instructions can be effectively fetched and decoded. Hence, processors like the Intel Pentium Pro use aggressive branch prediction to help insure a large pool of dynamically schedulable instructions. However, while branch prediction may work for many applications, it is much less effective for several important multimedia related signal processing applications. This occurs for the following reason.

The time-consuming inner loops of many signal processing applications have already been unrolled. For example, consider the inner loop of one of the block error estimation kernels of the Berkeley MPEG encoder [10].

```
for ( y = 0; y < 16; y++ ) {
  across = &(prev[fy+y][fx]);
  cacross = currentBlock[y];

  localDiff = across[0]-cacross[0]; diff += ABS(localDiff);
  localDiff = across[1]-cacross[1]; diff += ABS(localDiff);
  localDiff = across[2]-cacross[2]; diff += ABS(localDiff);
  localDiff = across[3]-cacross[3]; diff += ABS(localDiff);
  localDiff = across[4]-cacross[4]; diff += ABS(localDiff);
  localDiff = across[5]-cacross[5]; diff += ABS(localDiff);
  localDiff = across[6]-cacross[6]; diff += ABS(localDiff);
  localDiff = across[7]-cacross[7]; diff += ABS(localDiff);
  localDiff = across[8]-cacross[8]; diff += ABS(localDiff);
  localDiff = across[9]-cacross[9]; diff += ABS(localDiff);
  localDiff = across[10]-cacross[10]; diff += ABS(localDiff);
  localDiff = across[11]-cacross[11]; diff += ABS(localDiff);
  localDiff = across[12]-cacross[12]; diff += ABS(localDiff);
  localDiff = across[13]-cacross[13]; diff += ABS(localDiff);
  localDiff = across[14]-cacross[14]; diff += ABS(localDiff);
  localDiff = across[15]-cacross[15]; diff += ABS(localDiff);

  if ( diff > bestSoFar ) {
    return diff;
  }
}
```

Normally, a large pool of dynamic instructions can be maintained without branch prediction for loops with large bodies and data independent loop control. For this reason, many of the inner loops of multimedia related signal processing kernels can be efficiency executed without the need for branch prediction. However, the preceding loop highlights why branch prediction may not always work for multimedia related signal processing kernels. First, many ISA's (including the DLX) do not have an absolute instruction. Hence, each of the absolute functions in the preceding loop is compiled into a conditional branch around an instruction that performs a negation. Furthermore, this branch is not easily predictable (overall it tends to evenly split between taken and not taken with, at best, only short spans of either all taken or all not taken).

The problem can be alleviated somewhat by including both an absolute and maximum (finding the maximum causes the same problem) instruction, but not cured because of the many other seemly unpredictable branches found in the heavily used parts of many signal processing kernels. For example, consider the early return at the end of the body of the preceding loop. This early exit is there as an optimization, since once diff becomes larger than bestSoFar there is no reason to continue executing the loop. This is quite effective in that once a near optimal match is found (which usually occurs quickly), only one or two loops through the body of the preceding loop are needed to determine that further matches are less than optimal. Hence, while biased toward not exiting, this exit exhibits only short spans of not takens. Production quality, highly optimized, multimedia related kernels are often literally littered with optimizations such as the early exit in the preceding loop. As indicated, branch prediction is not effective when dealing with the branches related to these optimizations.

Hence, for many signal processing kernels, branch prediction is not needed where it would work, and does not work where it is needed. Therefore, to maintain a large pool of dynamic instructions, we turn to a more direct approach (which has been suggested before). Using a trace cache, as proposed by Rotenberg et al. [6], allows the pool of dynamically schedulable instructions to be quickly filled. However, while Rotenberg et al. assumed a trace of encoded instructions, we assumed a trace cache of decoded instructions so as to obtain the same increase in the ease of register renaming as that obtained in [4]. As indicated in this reference, register renaming remains a major constriction on the throughput when using a larger number of instructions (in our case due to parallel trace fetching).

4. SIMULATION METHODOLOGY

Two simulators were designed based on the Intel Pentium Pro architecture. Both use dynamic instruction scheduling and were tested with the following parameters: 512 instructions in the trace cache, 64 instructions in the instruction pool, 256 mappable registers, 4 integer scalar units, 1 floating point add unit, 1 floating point multiply unit, and 1 floating point divide unit. Instruction latencies are as follows: 1 cycle for integer operations, 2 cycles for memory operations and floating point adds, 5 cycles for floating point multiplies, and 19 cycles for floating point divides. Both simulators were constructed by using the DLX ISA and interface [8]. The simulators were tested with a number of popular benchmarks compiled under a modified version of GCC for the DLX ISA.

4.1 The Architectural Simulator. In both simulators, the architecture allows for superscalar dynamic instruction scheduling as described in section 1.1. The new simulators were further extended to include enhanced performance evaluation utilities.

The conventional simulator was implemented with a typical fetch/decode strategy supporting dynamic instruction scheduling, while in the enhanced simulator, the features of decoded instruction reuse and trace caching were added to that of dynamic instruction scheduling. Where possible, the overall structure of both simulators mirrors that of the Pentium Pro architecture. It is, however, possible to modify the number of reservation stations, the number of different types of functional units, and the instruction fetch bandwidth. Unlike the Pentium Pro, branch prediction was not used (due to the highly unrolled nature of the targeted application code). This did not cause a substantial loss in performance.

4.2 Modifications to GCC. GCC for the DLX architecture was extensively modified in order to produce what was considered to be reasonably efficient code for multimedia applications. This was done by enhancing the GCC for DLX, which is available from the University of Minnesota. The final compiler produced reasonably optimized code, but is not comparable to a commercial compiler. Ideally, the compiler would go to greater lengths to take further advantage of the new architecture. For example, substantial performance gains could be made by adapting the compiler to adjust loop length according to the length of the decoded trace cache. Small loops could be unrolled to reduce branch overhead and larger loops could possibly be split so that the smaller loops fit in the decoded trace cache. Such complicated optimizations are beyond the scope of the current compiler.

5. BENCHMARKS

As seen in industry [9], there is a great demand for architectures specifically adapted for multimedia applications. In order to demonstrate the efficiency of this architecture for such applications, the simulators were tested on a variety of multimedia cased benchmarks.

5.1 FAST - Fast Fourier Transform. FAST is an algorithm that implements a radix-eight fast Fourier transform as described by Bergland [11]. For this benchmark, an eight point kernel is used. Fast Fourier transforms typically use a great deal of looping, but the algorithm used was unrolled except for the primary loop. This had a very positive effect, as seen in section 6.1.

The fast Fourier transform is an optimized version of the discrete Fourier transform. It is used in a number of signal processing and multimedia applications. The importance of the FFT can be seen by the discussion of it in [9].

5.2 LPC - Speech Compression Algorithms. This algorithm makes use of linear prediction kernels for speech encoding. This is accomplished through finding the eigenvalues for a characterization matrix. In order to do the matrix operations associated with this algorithm, the daxpy routine for vector addition was borrowed from linpack. Naturally a highly parallel task, this algorithm is well suited to gauge the success of the architectural enhancements.

5.3 MPEG - Video Compression Algorithms. In the Berkeley MPEG encoding algorithm [10], the program was executed with the Find Best Match Exhaustive search algorithm. The program was used on a number of images with proportional results. Two tests were conducted, one using least squares and the other using the mean absolute distance error approximation.

With the rising popularity of multimedia applications, MPEG compression and decompression kernels are becoming very important. This is shown by the inclusion of MPEG decompression in the Intel media benchmark [7, 9].

6. RESULTS

The simulations that have been conducted suggest that there are substantial performance gains to be made through the use of the decoded trace cache. Figures 1 and 2 compare the parallelism of the two simulators for the benchmarks tested. All benchmarks that were considered showed an overall improvement of at least 1.52 times the speed of the conventional simulator. In fact, an average speedup of 3.50 was achieved as shown in table 1.

Benchmark	Conventional	Enhanced	Speedup
FFT	168937898	49848494	3.39
LPC	116297766	29798174	3.90
MPEG (LS)	20748963	3980118	5.21
MPEG (MAD)	17760771	11690908	1.52
AVERAGE			3.50

Table 1: Total number of cycles for the benchmarks tested

Table 2 shows the average number of instructions conventionally fetched and decoded per cycle in the enhanced simulator.

Benchmark	Fetch/Decodes
FFT	0.021974
LPC	0.000721
MPEG (LS)	0.000136
MPEG (MAD)	0.000047
AVERAGE	.0057195

Table 2: Average number of instructions fetched and decoded per cycle in the enhanced simulator

6.1 FAST - Fast Fourier Transform. The enhanced simulator showed a speedup 3.39 times the speed of the conventional simulator. As seen in table 2, the enhanced simulator showed that less than 3% of the instructions had to be fetched from memory and decoded. This figure results from the fact that code is organized into one large loop that can be held in the decoded trace cache. The code fragment for the primary loop follows:

```
t0 = b0[k] + b1[k]; t1 = b0[k] - b1[k]; t2 = b2[k] + b2[k];
t3 = b3[k] + b3[k]; t4 = b4[k] + b6[k]; t5 = b4[k] - b6[k];
t6 = b7[k] - b5[k]; t7 = b7[k] + b5[k]; t8 = r2 * (t7 - t5);
t5 = r2 * (t7 + t5); tt0 = t0 + t2; t2 = t0 - t2;
tt1 = t1 + t3; t3 = t1 - t3;
t4 += t4; t6 += t6;

b0[k] = tt0 + t4; b4[k] = tt0 - t4; b1[k] = tt1 + t5;
b5[k] = tt1 - t5; b2[k] = t2 + t6; b6[k] = t2 - t6;
b3[k] = t3 + t8; b7[k] = t3 - t8;
```

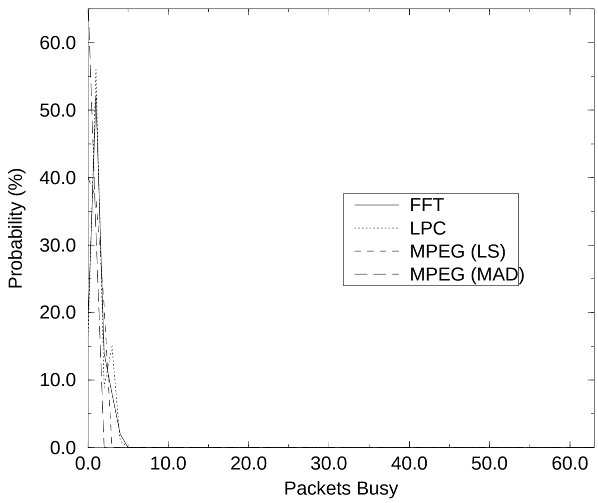

Figure 1: Pool size probability with dynamic instruction scheduling only

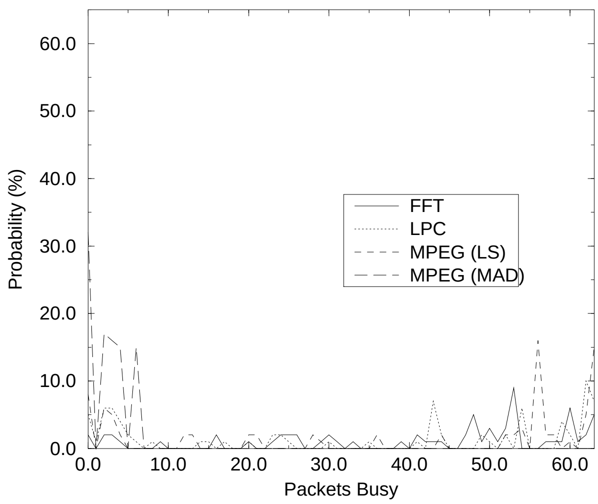

Figure 2: Pool size probability with the proposed enhancements

6.2 LPC - Speech Compression Algorithms. Overall, a speedup of 3.90 was achieved for the enhanced simulator. Except at startup, there was almost no (less than .1%) fetching and decoding in the enhanced simulator. This occurred due to the compact and iterative nature of the benchmark. The primary code segment follows:

```
while (to0 <= end)
{
   *(to0 + 1) += sca * *(fr0 + 1);
   *(to0 + 2) += sca * *(fr0 + 2);
   *(to0 + 3) += sca * *(fr0 + 3);
   *(to0 + 4) += sca * *(fr0 + 4);
   *(to0 + 5) += sca * *(fr0 + 5);
   *(to0 + 6) += sca * *(fr0 + 6);
   *(to0 + 7) += sca * *(fr0 + 7);
   *(to0 + 8) += sca * *(fr0 + 8);
   to0 += 8;
   fr0 += 8;
}
```

6.3 MPEG - Video Compression Algorithms. For the mean absolute distance error approximation, the enhanced simulator showed a speedup of 1.52. For the least squares error approximation, the enhanced simulator had a speedup of 5.21. This large difference is due to the nature of the error approximation algorithms. The least squares error approximation code has a less complex control structure and is therefore easier to execute without branch related stalls. Interestingly, for the conventional simulator, the mean absolute distance algorithm ran faster, and for the enhanced simulator, the least squares algorithm had a faster execution time. For both algorithms, the number of instructions that had to be fetched and decoded in the enhanced simulator was insignificant (again less than .1%).

7. CONCLUSION

We have shown that the decoded trace cache is an efficient scheme for further exploiting parallelism in superscalar processors. It allows the processor to execute instructions beyond what the fetch/decode bottleneck would normally allow. The scheme is especially useful for multimedia applications because of their use of compact, heavily-executed loops.

As indicated by the benchmarks that were tested, this architecture performs very well on multimedia applications. The architecture gets its largest advantages when operating on repetitive, computationally intensive, applications. Specifically, such applications include DSP algorithms, compression/decompression, and 3D graphics. Aside from further exploiting of parallelism, this architecture showed improvements in terms of latency, power consumption and the average complexity of issuing an instruction.

ACKNOWLEDGMENTS

The authors would like to recognize the contributions from Stanford, Berkeley, and the University of Minnesota; they contributed the DLX simulator, MPEG code and GCC respectively.

References

[1] "A Tour of the Pentium Pro Processor Microarchitecture" http://www.intel.com/procs/ppro/info/p6white/index.htm.

[2] W.M. Hwu and Y.N. Patt, "HPSm, A High Performance Restricted Data Flow Architecture Having Minimal Functionally" *Proc. ISCA*, Tokyo, 1986, pp. 297-306.

[3] S. Melvin, M. Shebanow, Y. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines" *Proc. 21th Ann. International Symposium on Microarchitecture*, December 1988.

[4] M. Smotherman, M. Franklin, "Improving CISC Instruction Decoding Performance Using a Fill Unit" *Proc. 28th Ann. International Symposium on Microarchitecture*, 1995.

[5] M. Hiraki et al.,"Stage-Skip Pipeline: A Low Power Processor Architecture Using a Decoded Instruction Buffer" *1996 International Symposium on Low Power Electronics and Design*, August 1996.

[6] E. Rotenberg et al., "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching" *29th Annual International Symposium on Microarchitecture*, December 1996.

[7] M. Slater, "The Land Beyond Benchmarks" *Comput. Commun. OEM*, Mag. 4, 31, pp. 64–77, September 1996.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture a Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, CA.

[9] A. Peleg, S. Wilkie, U. Weiser, "Intel MMX for Multimedia PCs" *Communications of the ACM*, vol. 40, no. 1, pp. 25–38, Jan. 1997.

[10] L. Rowe et al., "Berkeley MPEG Tools" *ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/mpeg/bmt1r1.tar.gz.*

[11] G. Bergland, "A Radix-eight Fast Fourier Transform Subroutine for Real-valued Series" *IEEE Transactions on Audio and Electro-acoustics*, vol. AU-17, pp. 138–144, 1969.

[12] Y.N.Patt, W.M. Hwu and M.C. Shebanow, "HPS, A New Microarchitecure: Rational and Introduction" *18th Annual International Symposium on Microarchitecture*, Asilomar, December 1985, pp. 103-108.