Journal of VLSI Signal Processing, ?, 1–13 (????)

Aggressive Dynamic Execution of Decoded Traces

BENJAMIN BISHOP
Department of Computer Science and Engineering, The Pennsylvania State University, University Park PA 16802

THOMAS P. KELLIHER
Department of Mathematics and Computer Science, Goucher College, Baltimore MD 21204

ROBERT M. OWENS
Department of Computer Science and Engineering, The Pennsylvania State University, University Park PA 16802

MARY JANE IRWIN
Department of Computer Science and Engineering, The Pennsylvania State University, University Park PA 16802

;

Abstract. In this paper, we consider the increased performance that can be obtained by using, in concert, three previously proposed enhancements. These enhancements are aggressive dynamic (run time) instruction scheduling, the reuse of decoded instructions, and trace scheduling (both aggressive dynamic instruction scheduling and decoded instruction reuse have been used in commercial systems). We show that these three enhancements complement and support one another. Hence, while each of these enhancements has been shown to have merit in its own right, when used in concert, we claim the overall advantage is greater than that obtained by using any one singly. To support this claim, we present the results from running benchmarks representing several common multimedia kernels. Subsequent simulations show results of 7.3 instructions completed per cycle for the best-performing benchmark for a reasonably aggressive microarchitecture that combines trace scheduling of decoded instructions (i.e. decoded traces) with aggressive dynamic execution.

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. We refer to decoded instructions stored in the trace cache as decoded traces. When used in concert, dynamic instruction scheduling, trace caching, and decoded instruction reuse provide for high execution rates. Furthermore, we show the marriage of these enhancements to be particularly useful in the execution of signal processing kernels related to multimedia applications. Efficient execution of these kernels yields significant overall performance gains due to the large amount of time spent in kernel execution. For example, [12] shows that over 90% of the execution time for the MPEG encoding program is spent in the block matching kernel.

2. 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 its latency, the reuse of decoded instructions has been proposed [4, 3]. The use of this technique has seen its 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., the merits (in terms of either reduced latency, power, or instruction issue complexity) of reusing decoded instructions in RISC ISAs 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 ensure a large pool of dynamically schedulable instructions. However, while branch prediction may work for many applications, it is much less effective for several important 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 [11]:

```
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 signal processing kernels can be executed efficiently without the need for branch prediction. However, the preceding loop highlights why branch prediction may not always work for signal processing kernels. First, many ISAs (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 for typical data sets).

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 seemingly 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).

3. Background

In this Section, we briefly survey previous work in aggressive dynamic instruction scheduling, reuse of decoded instructions, and trace scheduling.

3.1. 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 its operand(s) are 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, 14] 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 functional 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.

3.2. 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 re-decode an instruction every time it is executed [4]. Hence, instead of having a cache of encoded instructions, a cache 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 buffer require no additional decoding.

3.3. Trace Cache

The fill unit, as proposed by Melvin et al. [3], effectively deals with the problem of issuing a large number of simple instructions. In this scheme, complex instructions are decomposed into RISC-like microoperations that are then stored in a fill unit. The fill unit then forms groups of indivisible or atomic microoperations. These VLIW-like groups can be more simply scheduled.

The idea of the fill unit was later extended by Smotherman and Franklin [4] to allow for the reuse of decoded instructions. They noted that while CISC machines typically have reduced fetch bandwidth requirements, they suffer from difficult, high latency decoding. As a way to combat this problem, Smotherman and Franklin suggest using a fill unit to feed a decoded instruction buffer. When the buffer contains a pre-decoded version of a needed instruction, the instruction can be fetched directly from the buffer, bypassing the decode logic. Hiraki et al. [5] propose a similar strategy in order to reduce power consumption.

The trace cache, as proposed by Rotenberg et al. [6], decreases fetch/decode traffic by the use of a cache containing dynamically stored instruction traces. They suggest that by storing segments of the dynamic instruction stream, a supplemental cache could lower latency and fetch bandwidth requirements. Lower latency is achieved through the elimination/reduction of the pre- and post-processing stages that are required in conventional Icaches. High instruction throughput is maintained since it is constrained only by the width of the datapath and the branch predictor throughput. Rotenberg et al. report a performance gain of 28% for a reasonable size trace cache on the IBS and SPEC92 benchmarks.

The trace cache is organized into a series of traces. A trace is a portion of the dynamic instruction stream, often encompassing several basic blocks. Multiple basic blocks, which would be non-contiguous in a typical ICache, can be fetched from the trace cache per cycle. This idea is illustrated in Figure 1 [6]. The Figure shows the static instruction stream broken into basic blocks, suggests how the basic blocks are non-contiguously "re-assembled" in the dynamic instruction stream due to taken branches, and shows how the trace cache re-establishes the continuity of the dynamic instruction stream.

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.

Vajapeyam and Mitra [16], as well as Nair and Hopkins [17], have studied VLIW-style execution of dynamically formed traces. Both groups consider schemes to effectively deal with register renaming problems. Nair and Hopkins go on to give a detailed discussion of their two-part execution engine. Both works use the SPEC int benchmarks and are targeted at mainstream processors.

4. Proposed Optimizations

While the idea of assembling large atomic units of instructions from smaller ones, as suggested by Melvin et al. is not new, our design differs from previous work in the way it executes and fills the larger units. In a manner similar to the work of Smotherman and Franklin, we suggest a type of fetch/decode bypass. Like Smotherman and Franklin, instructions are requested from the fetch/decode hardware individually. Unlike their work, when an instruction is requested that exists in a stored trace, we advocate execution directly from the trace cache. As explained subsequently, scheduling can be simplified due to the atomic nature of the cached instructions. In the case where an instruction is requested that is not contained in a stored trace, the machine receives instructions from a superscalar execution path.

The idea of the trace cache is to capture segments of the dynamic instruction stream. We use the model suggested by Rotenberg et al. for the design of our instruction trace cache. We propose several restrictions on trace formation that simplify the overall processor architecture. In order to prevent fragmentation, trace formation is stopped when a backward branch is detected. Each trace element is dedicated to a functional unit or group of functional units as a way to simplify scheduling. Each horizontal slice of the reservation station matrix can be thought of as a pseudo-FIFO queue. In these horizontal slices, instructions can be scheduled in fifo order as their operands become available. Aside from global interconnect indicating operand availability, scheduling can occur independently for each horizontal slice.

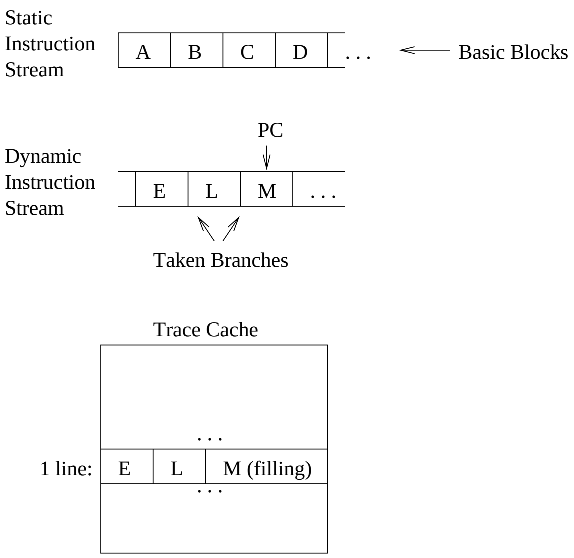

Fig. 1. The idea behind trace caching.

Parallel register renaming is allowed by halting trace formation when an instruction is encountered that has the same destination register as an instruction already in the trace. When a trace is initially created, preprocessing is performed to ensure that no two instructions write the same register. This approach greatly simplifies the combinational logic for dependency checking in the register renaming hardware. This combinational logic increases rapidly in complexity with increasing issue width in the typical superscalar model. The one write per register limitation did not cause any significant adverse effects on trace utilization as shown later in Table 1.

One alternative to this scheme would be to lift the one write per register restriction and store dependency information along with the trace. This scheme would further reduce logic in the renaming hardware, since no dependency check would be required. The main disadvantage of this approach is the overhead of storing extra trace information [19].

We extend the work of Vajapeyam and Mitra, and Nair and Hopkins by optimizing our design by preprocessing instructions prior to their placement within traces. The goal of this optimization is to do as much work as possible before initially writing a trace. Although our optimizations were crafted with multimedia benchmarks in mind, we believe that our techniques have broad, general applicability. In our design, groups of decoded instructions are stored with preprocessing assuring that the entire trace can be register renamed concurrently. In optimizing, we have limited the number of traces per starting address to one since the benchmarks we studied are typically repetitive and have only one heavily used dynamic instruction path in any reasonable time interval. This optimization and the repetitive nature of our applications allow for a very small (< 200 instructions) and highly utilized on-chip trace cache. Although only a small trace cache was needed, a large instruction pool (> 128 words) was necessary in order to maintain a high level of parallelism in execution. The large size of this instruction pool is not an area/speed problem due to the simplified scheduling policy mentioned above.

In summary, there are several important differences between our proposed enhancements and other work. These include a simplified scheduling algorithm, simplified register renaming, constraints on trace formation, and specific instruction pool and trace cache parameters. These optimizations take advantage of the inherent properties of the target application area. Since these applications typically have a small number of often repeated execution paths, trace execution can be simplified at the expense of trace formation.

Our enhanced architecture uniquely blends features from static VLIW and dynamic instruction scheduling architectures. Like static VLIW architectures, Our architecture achieves a high instruction issue rate by creating and caching decoded traces of the dynamic instruction execution path. These traces are better optimized than static VLIWs because they are formed at run time rather than compile time, as with static VLIWs. Once issued, instructions are executed out of order and committed in order.

4.1. Overall Processor Architecture

Figure 2 illustrates the combined use of the three previously described architectural techniques. Our architecture features hardware for trace and superscalar execution. In both cases, instructions are ultimately supplied by a decoupled fetch/decode unit. The unit has its own shadow program counter, which it uses to independently fetch instructions. Address generation has a latency of one cycle. The single-ported ICache has a latency of three cycles. The separate four-port (two read and two write) DCache, which is not shown in the figure, has a latency of one cycle. We assume ideal caching.

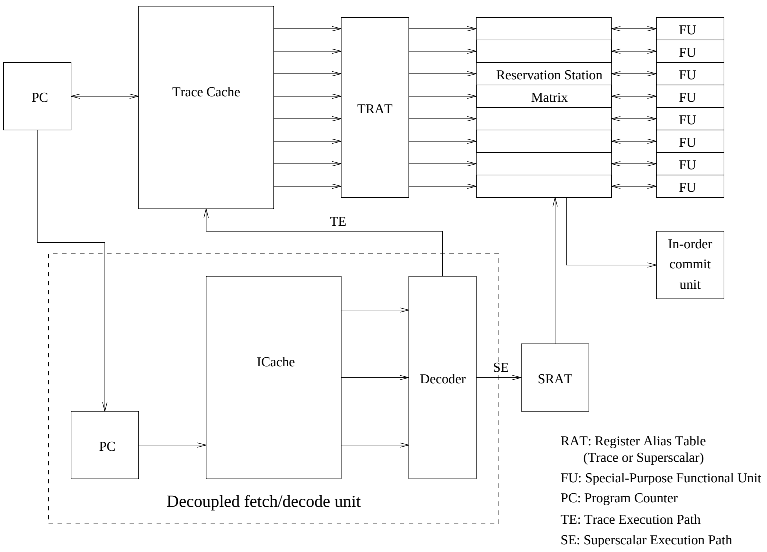

Fig. 2. Block diagram of suggested optimizations.

The two register alias tables (SRAT for the superscalar execution path and TRAT for the trace execution path) record the mappings from ISA registers to physical registers within the reservation station matrix (RSM), which is a rectangular array of reservation stations. A reservation station contains an operation code, pointers to source operands, result storage, and status bits. The RSM has horizontal slices feeding the functional units of the execution unit. These horizontal slices, in conjunction with trace formation, facilitate simplified scheduling. Additionally, traces are added to the RSM as vertical slices, simplifying trace placement within the RSM.

When instructions are supplied by superscalar execution path, the needed decoded instructions are directly provided by the superscalar unit. They are then renamed and passed into the vertical slices of the reservation station matrix. When the trace execution path is active, instruction blocks are fetched from the trace cache, renamed, and copied directly to a vertical slice in the reservation station matrix. Each instruction can be copied to the horizontal slice of the proper special purpose functional unit(s), since the instructions are separated into horizontal slices according to functional unit when the trace is formed. In both cases, instructions are retired in the in-order commit unit as in the Pentium Pro.

5. Simulation Methodology

We designed a simulator to evaluate our processor. It uses single path trace storage and the execution engine described previously. Other details and features of the architecture are modeled after the Intel Pentium Pro processor [1]. The simulator used the following run-time modifiable parameters: number of instructions per trace, number of functional units and their assignment to reservation station slices, number of trace cache lines, number of vertical slices in the reservation station matrix, fetch/decode bandwidth, number of mappable registers, and latencies for all functional units. The simulator was constructed using the DLX ISA [8]. The simulator was tested with a number of popular multimedia benchmarks compiled under an enhanced version of GCC[9].

5.1. The Architectural Simulator

The architecture of the simulator follows that of the machine described in Section 4.1. Where possible, the overall structure of the simulator mirrors that of the Pentium Pro architecture. It is, however, possible to make modifications to the machine's architectural parameters at run-time. The simulator does not make use of a specialized instruction set (such as MMX) for multimedia applications, although multimedia instruction sets could be used in such an architecture. Like the Pentium Pro architecture, the simulator employs branch prediction.

5.2. Modifications to GCC

The version of GCC used was extensively modified in order to produce reasonably efficient code for multimedia applications. This was done by enhancing the GCC compiler, which is available from the University of Minnesota [9]. The final compiler produced reasonably efficient code by using a number of non machine-unique optimizations. Such optimizations included unification of induction variables, strength reductions, invariant code migration, delay slot filling, and optimized register allocation. However, machine-unique optimizations, including machine-unique instruction ordering, were not used. 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 size of the trace cache. Small loops could be unrolled to reduce branch overhead and larger loops could possibly be split so that the resulting smaller loops fit in the trace cache. Such complicated optimizations are beyond the scope of the current compiler.

A major goal of this work was to study the effect of the combined architectural features on the execution of existing multimedia codes. Thus, we did not attempt to modify GCC to produce code with more ILP, as outlined in the previous paragraph. Implementing those, and other suggestions, should improve the IPC performance of our architecture.

6. Benchmarks

As seen in industry [10], there is a great demand for architectures with features specifically adapted for multimedia applications. In order to demonstrate the efficiency of our architecture for such applications, a variety of multimedia based benchmarks were evaluated.

6.1. LPC — Speech Compression Benchmark

This benchmark makes use of linear prediction kernels for speech encoding. This is accomplished through solving a system of linear equations by computing a matrix of covariants. In order to perform the matrix operations associated with this benchmark, the daxpy routine for vector addition was borrowed from linpack. This 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;
}
```

6.2. DCT — Discrete Cosine Transform Benchmark

This benchmark uses an algorithm that implements a radix-eight Discrete Cosine Transform as described by Bergland [13]. An eight point kernel is used. Discrete Cosine Transforms typically use a great deal of looping, but the code used was unrolled except for the primary loop:

```
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;
```

This had a very positive effect, as seen by its large performance gains in Section 7. The importance of the Discrete Cosine Transform in multimedia applications can be seen by the discussion of it in [10].

6.3. MPEG — Video Compression Benchmark

We used the Berkeley MPEG encoding benchmark [11] executed with the Find Best Match Exhaustive search algorithm. This algorithm finds the best match using an exhaustive spiral search. The program was used on a number of images with proportional results. Two versions of the benchmark were simulated, one using least squares (LSQ) and the other using the mean absolute distance (MAD) error approximation. In the algorithm, these error approximations are used to judge the goodness of compression choices.

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, 10].

6.4. CPOLY — Zeros of a Complex Polynomial Benchmark

The CPOLY benchmark [18] finds all the zeros of a complex polynomial or equivalently finds the eigenvalues of the polynomial's characterization matrix. This is accomplished by iteratively finding a zero of roughly increasing modulus and reducing the degree of the polynomial. This algorithm has applications in general signal processing, especially speech processing.

7. Simulation Results

The simulation results for our multimedia processor are promising. For the simulation, it was assumed that there was a 3 cycle delay in initially forming a trace. By default, other parameters were set to: 128 trace cache lines, 32 reservation station vertical slices, 256 mappable registers, 2 instructions fetched/decoded per cycle, and a total of 18 specialized functional units. Assuming 16 instruction could be stored per trace, it was possible to achieve as many as 7.3 instructions completed per cycle for the best performing benchmark. The worst performing benchmark achieved 1.7 instructions completed per cycle. On the average, 5 instructions were completed per cycle. Note that these figures show substantial improvements over the individual techniques on which this work is based.

Figure 3 compares the parallelism of our simulated enhanced architecture to a simulator which uses only dynamic instruction scheduling. Note that the vertical axis, probability in percent, is on a log scale.

Four figures show the overall performance of our multimedia processor with varying parameters. Figures 4 and 5 show the performance of each benchmark for different trace cache sizes. Figures 6 and 7 show the performance with a varying number of reservation station vertical slices. These figures show that only a modest amount of hardware is necessary to fully exploit the parallelism inherent in the simulated benchmarks; this can be seen by the convergence of the performance graphs to constant values. As a rule of thumb, one could expect to need 192 (24*8) to 240 (15*16) trace cache lines and 120 (15*8) to 256 (16*16) reservation stations to fully exploit the parallelism in this type of code as shown by the bars in Figures 4, 5, 6, and 7 respectively.

Figure 8 provides information about functional unit utilization. This information is significant in that it provides further insight into the nature of the benchmarks. It is possible to see which system resources are in the highest demand for each benchmark. Using this information, an architect could determine how many functional units of each type should be included in a processor, based on the target application of the machine.

More detailed information about general machine performance for each of the benchmarks is presented in Table 1. The information was collected in order to supplement the preceding performance data. The statistics are significant since they demonstrate that the fetch/decode hardware is rarely used, the trace cache is highly utilized, and branch predictability varies greatly for each benchmark.

Even though relatively high branch prediction rates were achieved for most benchmarks, branch prediction was found to be fairly unimportant in trace execution due to the machines ability to maintain a large instruction pool. This ability reduces the need for branch prediction since many branches can be resolved before the instruction pool empties.

As an interesting aside, even though the MPEG/MAD algorithm is generally thought to be the better algorithm, the simulated trace machine achieves significantly higher performance with the MPEG/LSQ algorithm due to its greater inherent parallelism. It was later found that an additional 13% performance increase could be achieved for the MPEG/LSQ benchmark by optimizing the algorithm for trace execution. See [15] for more information.

8. 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 kernels.

As indicated by the benchmarks that were tested, this architecture performs very well. The architecture gets its largest advantages when operating on repetitive, computationally intensive, applications. Specifically, such applications include digital signal processing and compression/decompression algorithms. Furthermore, subsequent simulation results show 7.3 instructions completed per cycle for the best-performing benchmark for a reasonably aggressive microarchitecture featuring aggressive dynamic execution of decoded traces.

Benchmark	Decodes/ cycle	Instructions/ trace	Branch prediction accuracy
LPC	0.020427	7.013724	0.921919
DCT	0.108866	7.224249	0.987534
CPOLY	0.070986	6.368976	0.888869
MPEG/LSQ	0.046884	7.781594	0.947648
MPEG/MAD	0.024478	4.172714	0.975317
AVERAGE	0.054328	6.512251	0.944257

Table 1. Benchmark statistics assuming 16 instructions per trace.

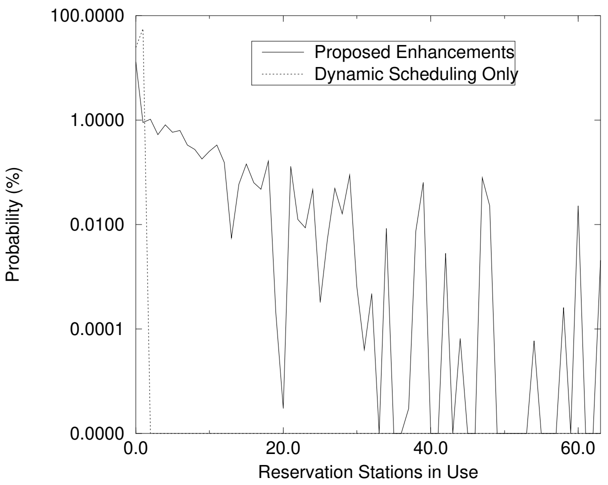

Fig. 3. Probability of reservation station utilization with the proposed enhancements.

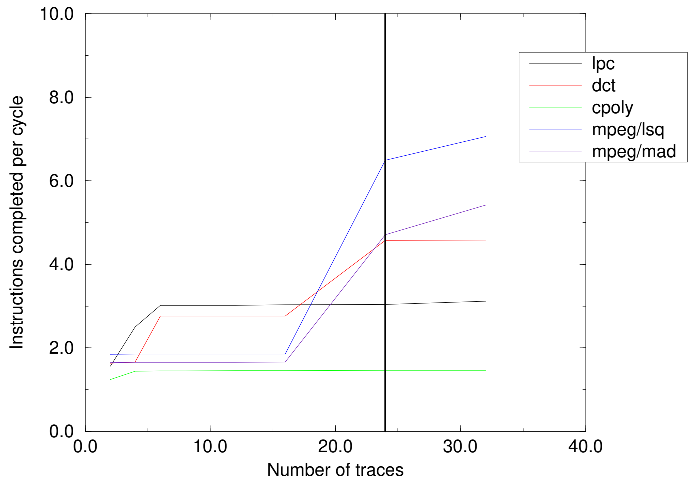

Fig. 4. IPC with a large number of reservation station vertical slices and varying number of trace cache lines assuming 8 instructions per trace.

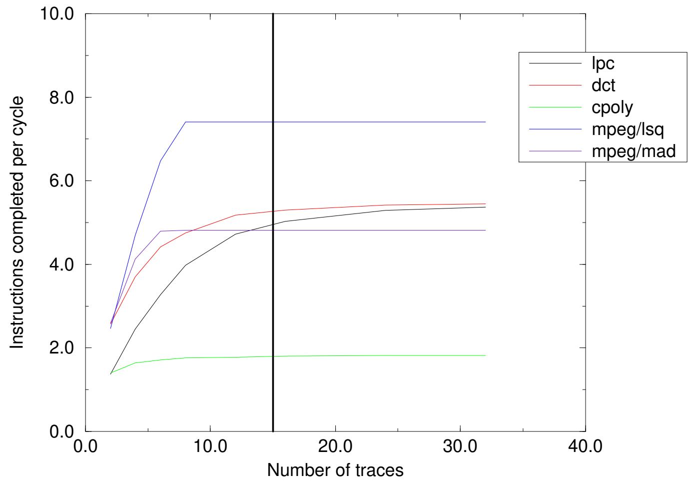

Fig. 5. IPC with a large number of reservation station vertical slices and varying number of trace cache lines assuming 16 instructions per trace.

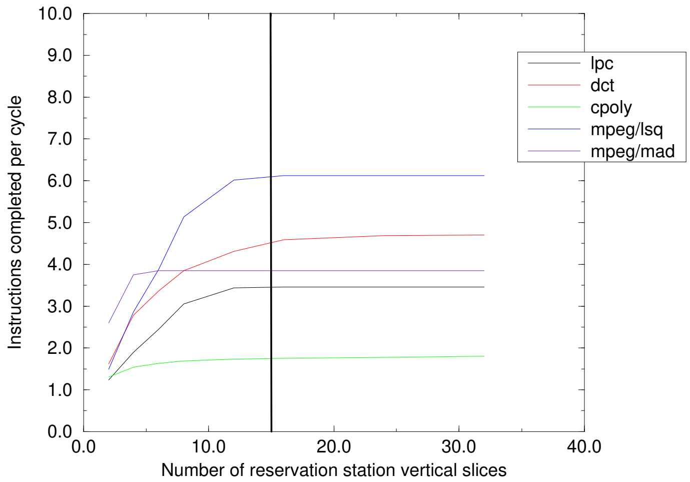

Fig. 6. IPC with a large number of trace cache lines and varying number of reservation station vertical slices assuming 8 instructions per trace.

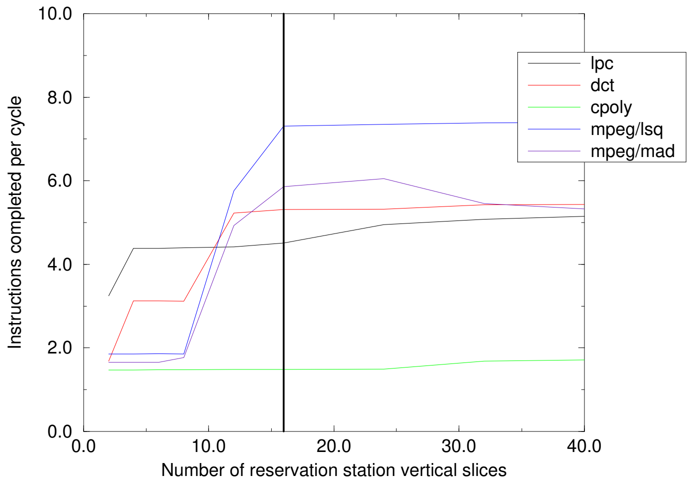

Fig. 7. IPC with a large number of trace cache lines and varying number of reservation station vertical slices assuming 16 instructions per trace.

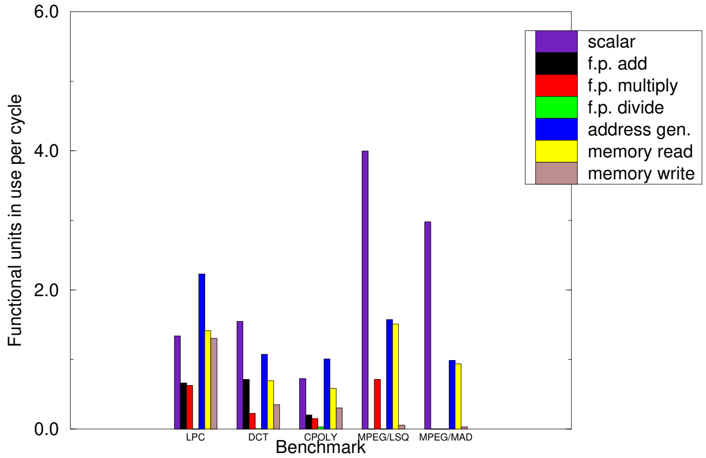

Fig. 8. Analysis of functional unit utilization.

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. "The GCC Compiler - Version 2.7.2" http://ftp.cs.umn.edu/pub/gnu/gcc-2.7.2.tar.gz.
10. A. Peleg, S. Wilkie, U. Weiser, "Intel MMX for Multimedia PCs" *Communications of the ACM*, vol. 40, no. 1, pp. 25–38, Jan. 1997.
11. L. Rowe et al., "Berkeley MPEG Tools" ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/mpeg/bmt1r1.tar.gz.
12. K. Gong and L. Rowe, "Parallel MPEG-1 Video Encoding" *1994 Picture Coding Symposium*, Sacramento, CA, September 1994.
13. 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.
14. Y.N.Patt, W.M. Hwu and M.C. Shebanow, "HPS, A New Microarchitecture: Rational and Introduction" *18th Annual International Symposium on Microarchitecture*, Asilomar, December 1985, pp. 103-108.
15. B. Bishop, T. P. Kelliher, R. M. Owens, and M. J. Irwin, "Re-Evaluating MPEG Motion Compensation Search Criteria." 1998 IEEE Workshop on Signal Processing Systems, Cambridge, MA, October 1998, pp. 123–131.
16. S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences" *Proc. 24th Annual International Symposium on Computer Architecture*, Denver, June 1997.
17. R. Nair and M. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups" *Proc. 24th Annual International Symposium on Computer Architecture*, Denver, June 1997.
18. M.A. Jenkins and J.F. Traub, "Algorithm 419: Zeros of a Complex Polynomial" *Communications of the ACM*, vol. 15, no. 2, p. 97, Feb. 1972.
19. D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors" *31st Annual International Symposium on Microarchitecture*, Dallas, November 1998, pp. 173–181.