Aggressive Dynamic Execution of Multimedia Kernel Traces

Benjamin Bishop Robert Owens Mary Jane Irwin
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802

Abstract

There has been relatively little analytical work on processor optimizations for multimedia applications. With the introduction of MMX by Intel, it is clear that this is an area of increasing importance. Building on previous work [4, 5, 6, 7, 13, 14], we propose optimizations for multimedia architectures that support independent parallel execution of instructions within dynamically assembled traces, resulting in dramatic performance improvements.

Specifically, we propose simplified instruction scheduling and register renaming algorithms due to constraints on trace formation. In addition, we suggest specific instruction pool and trace cache parameters. We constructed a simulator in order to measure the benefits of these processor optimizations for multimedia applications. The simulated machine, which could fetch/decode 2 instructions per cycle, performed better than a superscalar machine that could fetch/decode 8 instructions per cycle. Execution rates as high as 7.3 instructions per cycle were achieved for the benchmarks simulated, assuming 16 instructions per trace.

1. Introduction

Recently, there has been a growing interest in aggressive dynamic execution through the use of cached dynamically formed VLIW-like instruction sequences [13, 14]. This research has focused mainly on applications in the realm of general purpose scientific computing. While previous research concludes that there are significant benefits to be had though the use of this technique in scientific computing applications, we claim that even greater benefits can be achieved by applying this technology to multimedia-related problems. The importance of solving these problems is illustrated by the incorporation of MMX [9] into the Intel processor line.

In order to build an effective multimedia processor, it is necessary to understand the nature of the target application area. Upon examination, the codes for multimedia applications do not appear to readily fit into a class for which a benchmark suite exists. Some use floating point heavily; some do not. Distinct multimedia-related codes do share many important characteristics, however. For instance, they tend to be highly repetitive. These codes also have been heavily optimized to reduce, on average, the number of operations that must be performed. This is done through the use of special cases, early exits, loop unrolling, strength reduction, unification of induction variables, non-conventional algorithms, etc. However, these optimizations, while improving performance on a non-superscalar machine, make exposing parallelism more difficult due to the more complex control structures of the code.

1.1. Background

Dynamic instruction scheduling is very useful for exposing parallelism, which in turn can be used to increase performance. For this reason, many processors, including the HP PA 8000, IBM and Motorola PowerPC 604, Intel Pentium Pro, and SGI/MIPS R10000, use some type of dynamic instruction scheduling. Dataflow processors, where any instruction is eligible for execution as soon as the values for it's operand(s) become known, are capable of the highest degree of parallelism. However, the implementation of a dataflow processor can be difficult due to the complexity of scheduling a potentially very large pool of instructions.

What has been called restrictive dataflow [3, 4, 12] can be viewed as a compromise between completely static and fully 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. A number of prior works have proposed optimizations that can be used to improve dynamic scheduling. The following subsection discusses these works.

1.2. Prior work

The fill unit, as proposed by Melvin, Shebanow, and Patt [4], 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 [5] 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. [6] propose a similar strategy in order to reduce power consumption.

The trace cache, as proposed by Rotenberg, Bennett, and Smith [7], 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.

Vajapeyam and Mitra [13], as well as Nair and Hopkins [14], 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.

1.3. Proposed Optimizations

While the idea of assembling large atomic units of instructions from smaller ones, as suggested by Melvin, Shebanow, and Patt, is not new, our design differs from previous work in the way it executes and fills the larger units. Building on 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. 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 enters a conventional superscalar mode.

We use the model suggested by Rotenberg, Bennett, and Smith for the design of our instruction trace cache. In optimizing for multimedia applications, we have limited the number of traces per starting address to one since multimedia kernels are typically repetitive and have only one heavily used dynamic instruction path in any reasonable time interval. This optimization and the repetitive nature of most multimedia 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.

We build on the work by Vajapeyam and Mitra and the work by Nair and Hopkins by optimizing our design for multimedia applications. The goal of this optimization is to do as much work as possible before initially writing a trace. In our design, groups of decoded instructions are stored with preprocessing assuring that the entire trace can be register renamed concurrently.

In summary, there are several key differences between the proposed multimedia-specific enhancements and prior 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 multimedia applications typically have a small number of often repeated execution paths, trace execution can be simplified at the expense of trace formation.

2. Architectural Overview

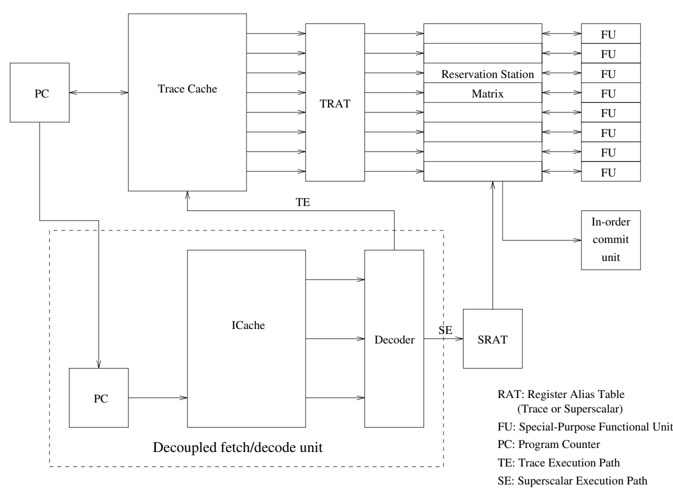

Figure 1. Block diagram of suggested optimizations

Figure 1 shows the overall architecture of a machine employing the suggested multimedia optimizations. It features hardware for trace execution mode and typical superscalar mode. In both cases, instructions are ultimately supplied by a decoupled fetch/decode unit. The unit has it's own shadow program counter, which it uses to independently fetch instructions.

When operating in superscalar mode, the needed decoded instructions are directly provided by the unit. They are then renamed and passed into the vertical slices of the reservation station matrix. When operating in trace execution mode, 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.

2.1. Trace Formation

The idea of the trace cache is to capture segments of the dynamic instruction stream. 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. 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.

2.2. Simplified Scheduling

Section 2.1 mentions that dedication of trace elements to functional units or groups of functional units can lead to simplified 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.

2.3. Register Renaming

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.

3. Simulation Methodology

A simulator was designed to evaluate our multimedia processor. It uses single path trace storage and the two-part execution engine described above. 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 by using a simple RISC ISA. The simulator was tested with a number of popular multimedia benchmarks compiled under an enhanced version of GCC[2].

3.1. The Architectural Simulator

The architecture of the simulator allows for superscalar dynamic instruction scheduling or trace execution mode as described in section 1.3. The simulator was further extended to include the ability to selectively disable trace execution mode in order to facilitate comparisons with typical architectures. 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 when operating in either superscalar or trace mode.

4. 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 our architecture for such applications, a variety of multimedia based benchmarks were evaluated.

4.1. 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.

4.2. DCT - Discrete Cosine Transform Benchmark

This benchmark uses an algorithm that implements a radix-eight Discrete Cosine Transform as described by Bergland [11]. 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. This had a very positive effect, as seen by its large performance gains in section 5. The importance of the Discrete Cosine Transform in multimedia applications can be seen by the discussion of it in [9].

4.3. MPEG - Video Compression Benchmark

We used the Berkeley MPEG encoding benchmark [10] 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 [8, 9].

4.4. CPOLY - Zeros of a Complex Polynomial Benchmark

The CPOLY benchmark [15] 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.

5. 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. In the superscalar simulator, there was no delay penalty for decoding instructions. 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. Unless otherwise stated, data for all figures was generated using these default parameters. 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 2 makes comparisons between the trace execution machine that is able to fetch/decode two instructions per cycle against superscalar architectures. These figures show that, for only a modest investment in hardware, it is possible to achieve superior performance for a number of multimedia benchmarks through the use of trace execution.

One would expect an 8-way superscalar machine to perform as well as a trace machine using the same number of functional units. The lower performance of the 8-way superscalar machine is mainly due to conflicts in the fetched instruction group. The trace machine forms groups of instructions dynamically in vertical slices, each element corresponding to a functional unit or group of functional units, assuring a balance of instructions in each stored trace. In this way, it reduces scheduling problems due to fluctuating demand for particular functional units.

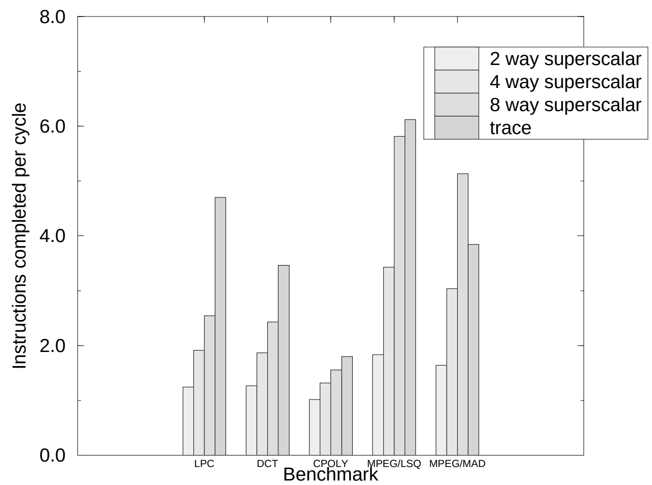

Figure 2. Comparison of trace execution with 8 instructions per trace to typical superscalar architectures with varying issue width

Four figures show the overall performance of our multimedia processor with varying parameters. Figures 3 and 4 show the performance of each benchmark for different trace cache sizes. Figures 5 and 6 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 3,4,5, and 6 respectively.

Figure 7 analyzes the amount of time spent in each of the modes of execution. This information was gathered in order to further understand the performance of each benchmark. For example, CPOLY's relatively poor performance can be explained by the numerous stalls that occurred during execution. The MPEG/MAD benchmark spent very little time in stalls. Its poor overall performance in Figures 3 through 6 is due to low trace utilization, as shown in Table 1.

Figure 8 provides information about functional unit uti-

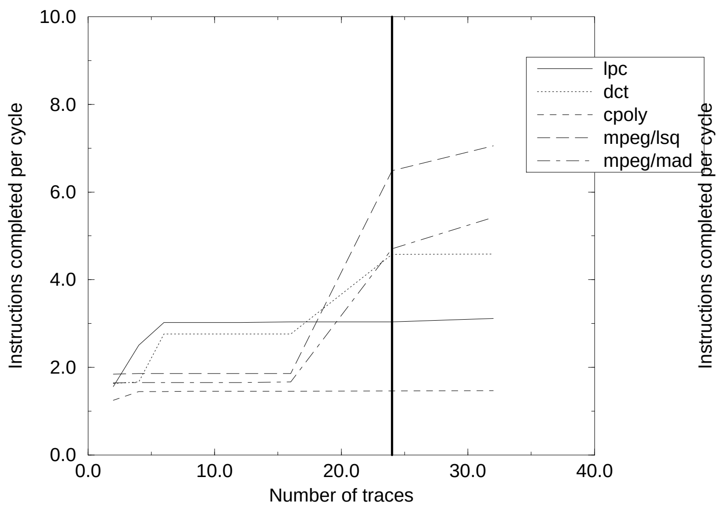

Figure 3. IPC with a large number of reservation station vertical slices and varying number of trace cache lines assuming 8 instructions per trace

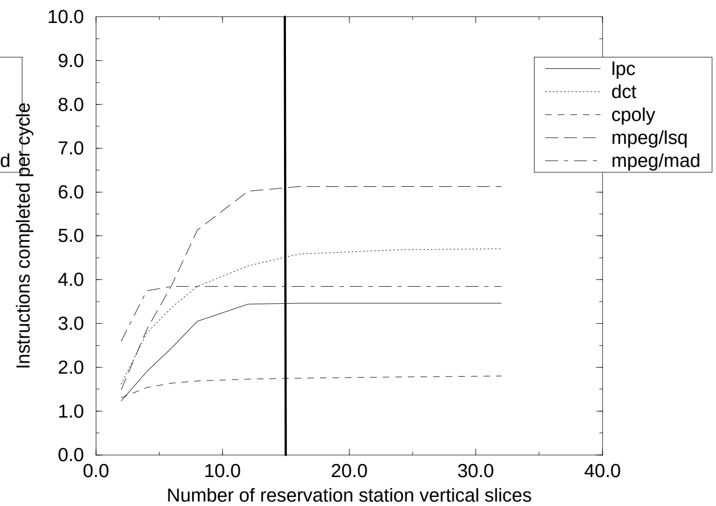

Figure 5. IPC with a large number of trace cache lines and varying number of reservation station vertical slices assuming 8 instructions per trace

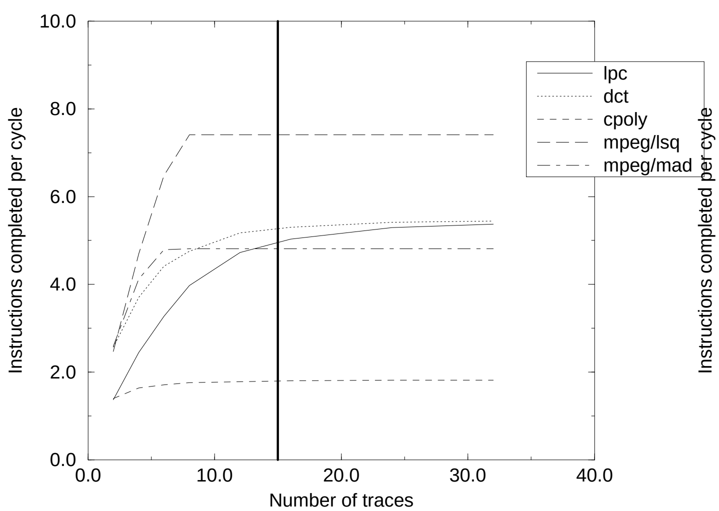

Figure 4. IPC with a large number of reservation station vertical slices and varying number of trace cache lines assuming 16 instructions per trace

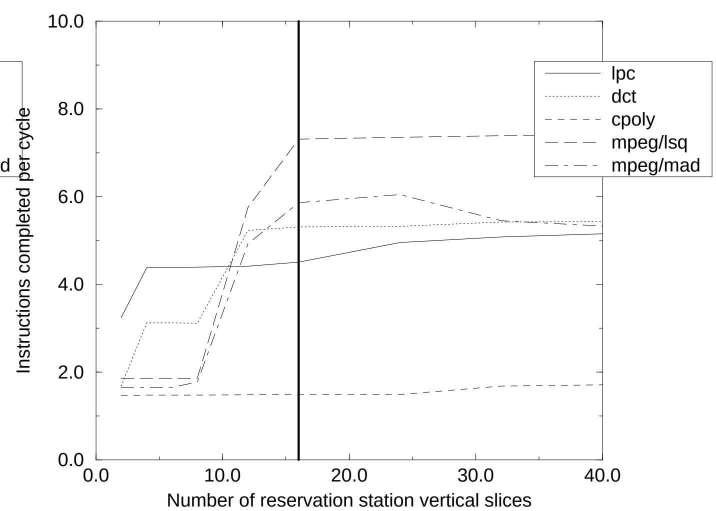

Figure 6. IPC with a large number of trace cache lines and varying number of reservation station vertical slices assuming 16 instructions per trace

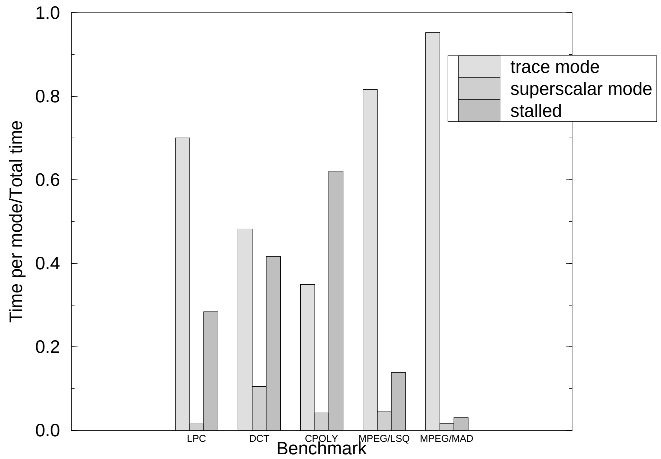

Figure 7. Analysis of time per execution mode

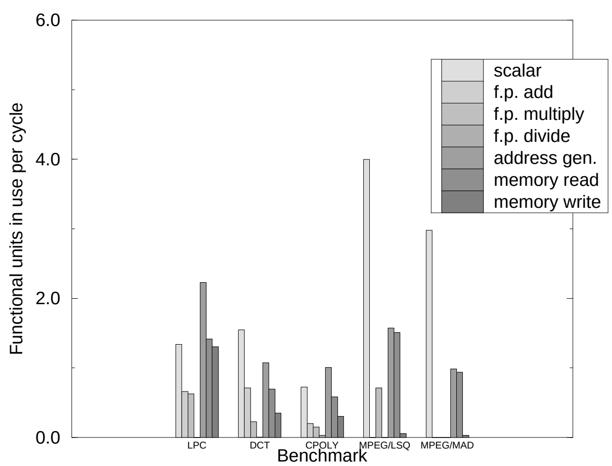

Figure 8. Analysis of functional unit utilization

lization. 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 mode 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.

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

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. As shown by Table 2, the MPEG/LSQ algorithm was 92% faster than the MPEG/MAD algorithm. 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. A subsequent paper will explore this phenomenon in greater detail.

Benchmark	Total Cycles
MPEG (LSQ)	2156577
MPEG (MAD)	4138861

Table 2. Comparison of least squares and mean absolute distance MPEG algorithms assuming 16 instructions per trace

6. Conclusion

We have presented optimizations that make substantial improvements over current techniques in an important application area. A simulator was constructed to analyze the benefits of this design for a number of popular multimedia benchmarks. Data from our simulator shows that there are great benefits to be had from the use of these optimizations. In fact, the MPEG/LSQ benchmark showed an execution rate of 7.3 instructions completed per cycle assuming 16 instructions per trace. Interestingly, the performance of the MPEG/LSQ benchmark was significantly higher than for the MPEG/MAD benchmark on the simulated machine.

The performance gains mentioned above were demonstrated with only a small investment in hardware. High performance was achieved with a fetch/decode bandwidth of two instructions/cycle. The trace cache used is less than 200 words in size, which is very small by todays standards. This was made possible in part due to the efficient utilization of the trace cache. When allowing 8 instructions per trace, the average utilization was 6.8. The number of reservation stations was larger than average, however. The reservation station matrix could be kept to an acceptable size in hardware by taking advantage of the relatively simple scheduling method used. In addition, an algorithm was used that allows parallel register renaming for a given trace due to restrictions imposed on trace formation.

7. Acknowledgments

The authors would like to recognize the contributions of UC Berkeley, and the University of Minnesota for the MPEG code, and GCC respectively. This research was funded in part by NSF grant no. MIP-9705128. In memory of Dr. Robert M. Owens, 9/8/49 - 9/14/97.

References

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

[2] "The GCC Compiler - Version 2.7.2" http://ftp.cs.umn.edu/pub/gnu/gcc-2.7.2.tar.gz.

[3] W.M. Hwu and Y.N. Patt, "HPSm, A High Performance Restricted Data Flow Architecture Having Minimal Functionally" *Proc. 24th Annual International Symposium on Computer Architecture*, Tokyo, 1986.

[4] 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.

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

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

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

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

[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" *Proc. 18th Annual International Symposium on Microarchitecture*, December 1985, pp. 103-108.

[13] 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.

[14] 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.

[15] 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.