AN ANALYSIS OF THE ALPHA INSTRUCTION SET ARCHITECTURE

Allen Fung, Maureen Chesire, Valentin Razmov

1. Introduction

In this document we present the results of a set of experiments performed on the Alpha 21164 instruction set architecture using the *jpegtran* application (which is to be found at /uns/bin/jpegtran) on a sample JPEG image file. "jpegtran" is a UNIX utility for applying various kinds of transformations on image files in that format (e.g. image rotation, transposition, etc.). We chose to instrument this application because:

a) It is relatively large (168K) and therefore allowed us to generate a sufficient amount of data for analysis;

b) In a graphical application, it is more likely to encounter some of the "new" instructions introduced specifically to meet the needs of this type of software.

2. Instruction mix analysis

For our first experiment, we measured the execution frequency of different classes of instructions. The following statistics were obtained by instrumenting jpegtran and running it on a sample input JPEG image file:

Instruction Type	**Instruction Count**	**Frequency**
Integer arithmetic and logic	3,956,006	53.37%
Floating point arithmetic	0	0.00%
Data transfer	2,385,380	32.18%
Conditional branch	798,711	10.78%
Unconditional branch	238,926	3.22%
Subroutine calls / returns	18,916	0.26%
Miscellaneous	0	0.00%
Prefetch	0	0.00%
Conditional move	800	0.01%
Multimedia	0	0.00%
Total	**7,398,739**	**99.82%**

The results, summarized above, matched our intuitive expectations. As can be seen from the table, majority of the instructions executed were *ALU operations*, which is typically the case in any type of application. Interestingly enough, no *floating-point operations* were performed. This may be attributed to the fact that graphics applications tend to be very computationally intensive, performing convolutions on a large set of similar basic data structures. For this reason, this type of application is usually designed in a way that would heavily exploit the least expensive operations from the instruction set, which in turn would significantly improve performance. Since integer arithmetic is much "cheaper" in CPU time than floating-point arithmetic, we might expect to see graphics applications utilizing very few floating point instructions, if any.

Another interesting observation is that, according to our measurements, *conditional branches* accounted for roughly 75% of all control flow instructions. That means that only one in every four control flow instructions was a subroutine call, a subroutine return or a jump. This could be attributed to the type of application. Graphics applications may tend to have a slightly lower percentage of *subroutine calls and returns* because most of the computation involves transformations (convolutions) performed in a small number of subroutines, thus the number of calls and returns from subroutines is less than usual. On a different note, since control flow changes in this application are primarily attributed to branches, and since changes in the control flow of a program have a significant effect on performance, optimizing the execution of branch instructions in this architecture is important.

A little surprisingly, the "new" instructions (*conditional moves*, *data prefetches*, *multimedia*) accounted for a very small portion of the total number of instructions executed. Our explanation for the displayed infrequency is that these instructions were not widely used by compilers at the time of the creation of the jpegtran application.

3. Branch Analysis

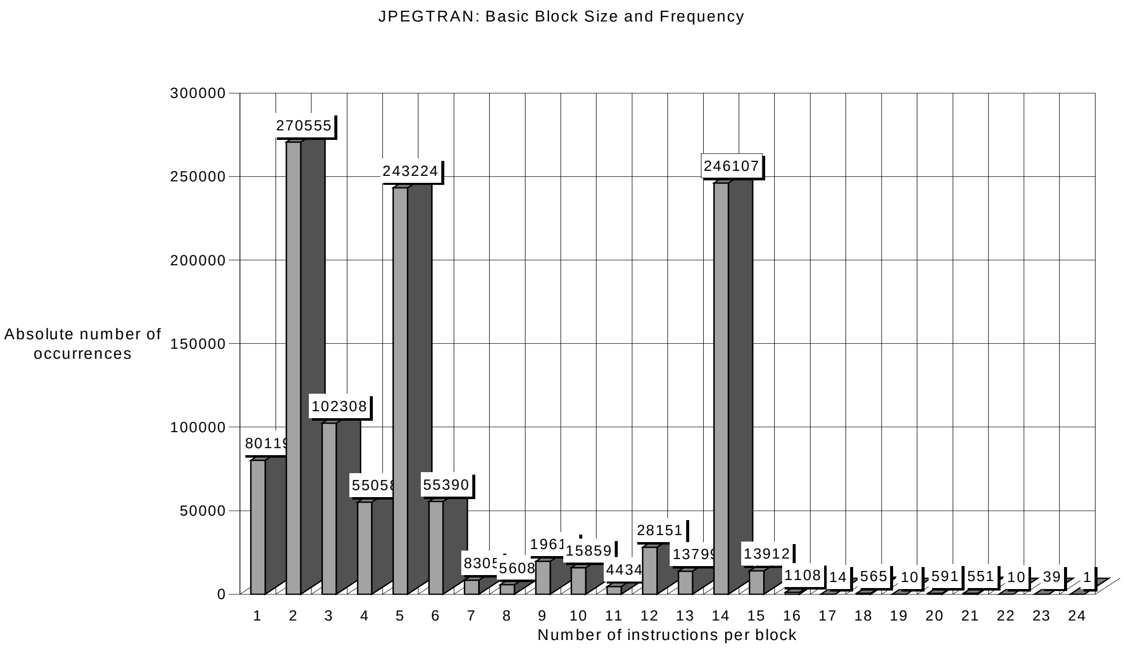
In our second experiment, we obtained measurements from a sample execution of our instrumented application for different types of branches. We recorded both the direction of the branch (forward or backward), as well as whether the branch was taken or not. Following is a summary of the results collected:

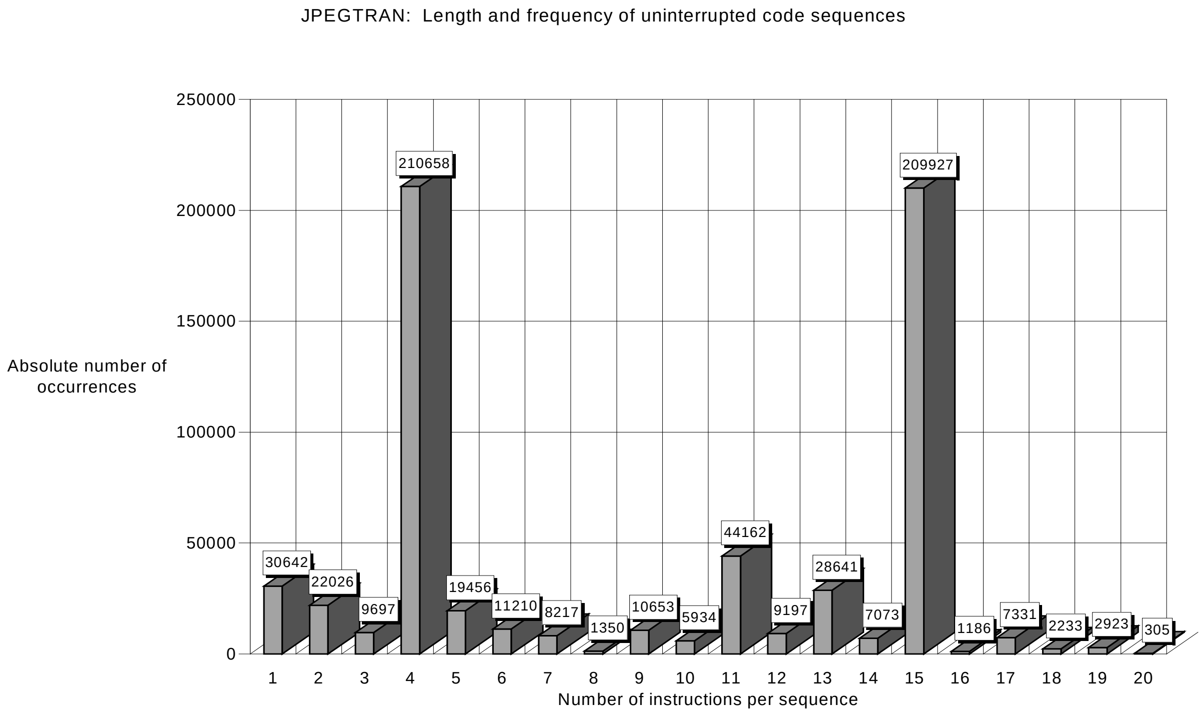
Branch Type	**Instruction Count**	**Frequency**
Forward branches taken	187,194	23.44%
Forward branches not taken	292,753	36.65%
Backward branches taken	244,617	30.63%
Backward branches not taken	74,147	9.28%
Total	**798,711**	**100.00%**

Notably, 76% of all backward branches were taken, compared to just 39% of all forward branches. The reason for the big difference is that the typical way of implementing loop structures is through backward branches. Loops in turn are usually executed multiple times, which explains why three in every four backward branches were taken and only one fell through. Another interesting observation is that the frequency of forward branches taken was lower than the frequency of forward branches not taken. Based on these observations, we claim that a static branch predication scheme which predicts all backward branches as taken and all forward branches as not taken will most likely yield the best performance for this particular application. In this case, the compiler should organize the generated code in such a way that favors forward branches not taken, in order to yield better correct prediction rates from the underlying architecture.

4. Block size analysis

In this experiment, we measured both the lengths of basic blocks and uninterrupted code sequences in the application. Following are the histograms of the results obtained:





Block Type	**Maximum Sequence Length**	**Average Length**
Basic block (static)	65	6.36
Dynamic block (uninterrupted sequence)	107	9.87

Our analysis shows that the number of instructions in a basic block (static) are typically less than the number of instructions in a dynamic block (i.e., an uninterrupted code sequence). For instance 69% of all basic blocks had 6 or less instructions, as compared only 45% of all uninterrupted sequences of executed instructions. Differences in dynamic and static sizes of straight-line code sequences can be attributed to the fact that a fairly large proportion of branches in this application are not taken. Since they account for about half (45%) of all branches, different static blocks may sometimes be executed as a straight-line code sequence. An interesting observation however is that, despite these differences, both static and dynamic lengths of uninterrupted instruction sequences are relatively short-- the average length in both cases is less than 10 instructions. Because of these short instruction sequences, we can conclude that branches occur relatively frequently in this application. The frequency of branches in our instrumented program shows that the threat of potential stalls from control dependencies is very high. This is especially true in superscalar processors, since branches will arrive n times faster in an n-issue processor. Without accurate branch prediction hardware, stalls from control dependencies could occur at every clock cycle in a multiple issue machine.

5. Instruction and block frequency analysis

In our final experiment, our goal was to determine whether certain portions of the code were executed more frequently than others, and measure the relative differences in the execution frequency. The results obtained are presented in the following tables:

Table 1. Block execution frequency

Frequency of execution (in times)	**Number of blocks**	**Percentage of the total number of blocks**	**Number of executed blocks**	**Percentage of the total number of executed blocks**
over 100,000	3	0.06%	614,451	52.72%
10,000 to 100,000	21	0.44%	230,848	19.81%
1,000 to 10,000	92	1.92%	277,482	23.81%
100 to 1,000	90	1.88%	34,060	2.92%
10 to 100	153	3.19%	6,471	0.56%
1 to 10	1,081	22.53%	2,168	0.19%
0	3,358	69.99%	0	0.00%
Total	**4,798**	**100.01%**	**1,165,480**	**100.01%**

Table 2. Instruction execution frequency

Frequency of execution (in times)	**Number of instructions**	**Percentage of the total number of instructions**	**Number of executed instructions**	**Percentage of the total number of executed instructions**
over 100,000	21	0.09%	4,353,072	58.73%
10,000 to 100,000	112	0.48%	1,219,682	16.45%
1,000 to 10,000	536	2.30%	1,563,342	21.09%
100 to 1,000	601	2.57%	232,572	3.14%
10 to 100	747	3.20%	31,947	0.43%
1 to 10	5,883	25.20%	11,858	0.16%
0	15,449	66.16%	0	0.00%
Total	**23,349**	**100.00%**	**7,412,473**	**100.00%**

Assuming constant CPI, we see that about 95% of execution time is being spent on only 2.5% of the blocks, and almost 70% of the blocks are not being executed at all! The same observation can be made for instruction execution: 95% of the execution time is spent on about 3% of the instructions, while over 65% of the instructions are not being executed at all. These results are consistent with the use of Amdahl's law - most of the execution time is spent on a small subset of the instructions in the application. Based on these observations, we can clearly see the importance of caches. Since a small part of the program is executed very frequently, the instructions in the application have high spatial locality. The architecture can exploit this locality to design a small but fast memory component, which provides a significant improvement in performance.