Value Prediction

Lecture #10: Monday, 1st May 2000
Lecturer: Mahesh Madhav
Scribe: Anamaya Sullerey
Reviewer: Mattan Erez

1 Introduction

Data dependencies in execution are one of the main hurdles in extracting parallelism out of sequential programs. One possible approach followed to overcome this dependency bottleneck is speculation. In the context of data this speculation can be in two respects. One can speculate on the address of the data, and on the value of the data. In this session we discussed following three papers.

- An Architectural Alternative to Optimization Compilers
- Exceeding the Dataflow Limit via Value prediction
- An empirical analysis of instruction repetition

2 An architectural alternative to optimization compilers [1]

This paper concentrates on an architecture that is geared towards identifying common subexpressions and reusing the previously calculated results. By doing this, the overall processor speed will increase due to lesser computations and memory traffic.

A common subexpression is part of an expression, appears in multiple expressions, with the same operand values and results. For example, in the expressions,

X=A+B

Y=A+B+C

'A+B' is a common subexpression.

Identifying common subexpressions can lead to reuse. It is not possible to determine all common subexpressions at compile time. This is because not all dependencies can be determined at compile time. One example is when the input operands to the expression are pointers.

Two subexpressions are congruent if they have the same sources. Congruent subexpressions are a superset of common subexpressions. If the value of the operands is the same, then congruent subexpressions are common subexpressions.

The paper suggests a two-operand stack based architecture (called Tree Machine(TM) architecture). The architecture tries to detect common subexpressions at runtime. The code in this architecture is arranged like a parse tree.

There are 'phrases' in the static code. A phrase is a segment of code that is more like a function and is executed by calling a PUSH operation to the address of the first instruction of the phrase. In this way potential common subexpressions execute the same phrase. The last instruction of the phrase is indicated by a bit in the instruction. To load a data word in the stack, a PUSH operation is used to branch to the address of the instruction that loads the data onto the stack. This way, any data loaded using the PUSH operation gets associated with the address of the instruction that loads it. This property is used later to detect dependencies. This method of loading data is referred to as executing a data word in the paper.

Results of all phrases are stored in the value cache with their dependency bits. Dependency sets are represented by bit vectors of length N. Bit i represents dependencies on variables whose address is 'N modulo i'. Whenever a data word is executed, its address is used to set a bit in the current phrase's dependency set.

Before executing any instruction, the value cache is checked for its contents. If a valid result for the phrase exists in the value cache, it is used directly. Whenever any store instruction is executed, the address of the instruction is used to invalidate all the entries that are dependent on the same dependency set that the store belongs to. Each entry in the value cache maintains explicitly the address ranges on which it is dependent, and whenever location in that range changes, the entry is invalidated.

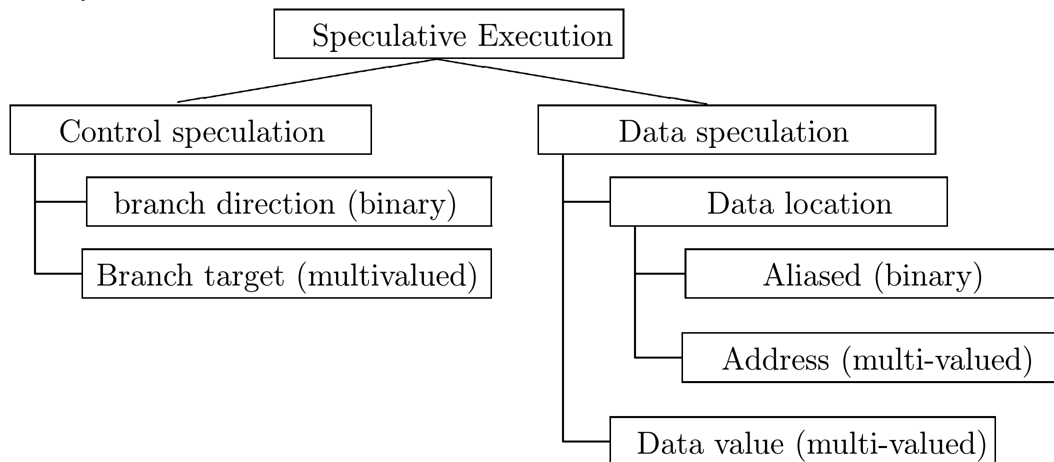
There are unnecessary evictions from the value cache. These can occur due to limited capacity of the value cache, or because there are multiple locations in any of the N dependency sets. A change to any of the data locations can invalidate a value cache entry, though it may not be dependent on it. This will depend on the number of dependency sets, as the number of addresses associated with a dependency set is detemined by the total number of dependency sets.

For testing, six FORTRAN programs were run using a compiler and emulator for the TM architecture, and the results were compared against the PDP-10. The results showed an average of 11% improvement in the code size and 18% improvement in the execution time. The value cache size was 32 entry, and there was a 80% hit rate in the value cache.

This paper was published in 1982(when MIPS group was doing tower of hanoi) and hence the benchmarks and the result section of the paper are different from what recent papers have. It gives some idea of level of sophistication of experiments at that period. This scheme is not doing speculation of any form, only reuse. Another question came up was how difficult it would be to pipeline such architecture, or have out of order execution, but the reason for including this paper was because it was one of the first on this subject.

3 Exceeding the dataflow limit via value prediction [2]

This paper proposes a mechanism for value prediction. Section two of the paper talks about a taxonomy of speculative execution. It sets it up in the way shown below. It was mentioned that there could be more categories, though the discussion on complete taxonomy was left for later on.



Taxonomy of speculative execution

The paper defines value locality as the likelihood of a previously seen value recurring in a storage location. The scope of the paper limits their study to examine the value locality of general purpose and floating point registers.

The locality is measured for history depths of one and four. In the case of history depth of four, the value is matched with one of the previous four values. Locality data for various SPEC'92 benchmarks (figure 2 in the paper) showed that for depth one, the average locality in the benchmarks is about 49% while for the history depth of four this figure is 61%. Locality on the basis of instruction was also explored (figure 3 in paper). Locality for an instruction is taken as the predictability of the value that the instruction writes.

To exploit this locality the authors propose a value prediction unit. It has two parts, the classification table (CT) and the value prediction table (VPT), and both of them are indexed with the program counter. The CT figures out whether an old value can be used. It classifies instructions into predictable and not predictable. It has a bunch of saturating counters and a valid field. The (VPT) contains the value. It is not clearly mentioned whether four or one value is stored for history depth of four. If there are four values, the paper does not clarify how one of them is chosen. Hit rate data (Figure 4) shows the sensitivity to the size of the VPT. It saturates after VPT has 4K entries.

Various schemes (table 3) with different sizes and counter lengths were used for the CT. There is no explanation of why those configurations were chosen. Results (Figure 6)

show the hit rates for these configurations. The graphs crisscross and do not give clear insight into pros and cons of various CT configurations. The configuration having a two bit counter, with 1024 entries in the CT gave the best performance.

A penalty occurs when there is misprediction while speculatively executing an instruction. Since the instruction is in the reservation station, it takes an extra cycle to reissue it. Also, there is extra penalty due to resource usage by the instruction.

Experiments were done to show speedups for three different architectures, PPC620 with a maximum of 16 instructions in flight at a time, PPC620+ with a maximum of 32 instructions in flight at any time, and PPC620+ with any number of instructions in flight at a time. The average speedups were about 5%, 7%, and 27% respectively (figures 9, 10, and 11 in the paper). The last experiment showed that investing the same amount of space in the data cache brought a 1.3% improvement, which is less than value prediction.

It was assumed that when having a history depth of 4 or 8 , the right value can be chosen perfectly. Since this scheme solely depends on history, it would not be able to exploit highly predictable operation like increment. Moreover it was pointed out that the results were optimistic because the setup did not have a mechanism to wait till any instruction has retired, before using it for prediction. Also, the architecture was not aggressive enough to exploit the full potential.

4 An empirical analysis of instruction repetition [3]

This paper tries to study the instruction repetition during execution. It tries to find the various sources of instruction repetition.

A static instruction, i.e., an instruction in the compiled code, can be executed a number of times. Each such instance is called a dynamic instruction. An instruction is said to be repeated if multiple dynamic instances of it have the same outcome. Another definition may have same inputs as requirement, though this paper follows the former definition.

The various sources of repeatability are

- Repetition of input data
- Overhead of loops and other software constructs
- Calculation of data addresses

The paper defines a unique repeatable instance as the first dynamic instance of a static instruction that re-occurs multiple times with the same outcome. Figure 1 shows that most of the dynamic instructions are due to less than 20% of the unique repeatable instances. The authors do analysis at three levels, global, function and local, to study instruction repetition.

The global analysis is at the program level. In this analysis, the instructions are divided into three categories (there is a fourth *uninit* category also, which is not significant).

1. Instructions whose inputs are affected by external program inputs.
2. Instructions whose inputs are affected by global variables.
3. Instructions whose inputs are affected by program internals.

The authors run various benchmarks. Data shows that the repeated instructions mostly come from the third category. This is because a lot of instructions use internal values.

The data in figure 3 (a bit confusing) shows the breakup in three different ways. First part shows the breakup for all dynamic instructions for the whole program. Second one shows the breakup for all the repeated dynamic instructions for the whole program. The third one shows what percentage of dynamic instructions in each category of the first part resulted in repeated instructions.

The function level analysis explores the repetition in function calls. Table 4 in the paper shows the result of the analysis, though this data is useless as the function may be passing pointers, which point to locations that hold different values at different time.

In the local analysis, the instructions are divided into two broad categories based on following criteria:

1. the source of input data used by the instruction
2. the specific tasks performed by groups of instructions.

Both of these categories are further subdivided into five categories. Tables 5,6, and 7 show the results. The data is different than one generated in global analysis. This is because at local level some information is lost. For example, compile time constant passed using a function is not seen as a constant at local level.

Section 5 of the paper comments on the software exploitation of the instruction repetitions. The challenges include lack of information on dynamic path at compile time, problems posed common techniques such as loop unrolling, or cases such as recursive functions.

Section 6 of the paper discusses hardware techniques to use instruction repetition. The authors show that a large part of repetition can be captured in 8K entry, 4 way set associative buffer instruction reuse buffer.

References

[1] Harbison S. P., "An Architectural Alternative to Optimization Compilers", Proceedings of the *first Symposium on Architectural Support for Programming Languages and Operating Systems*, 1982.

[2] Lipasti M. H., Shen J. P., "Exceeding the Dataflow Limit via Value prediction", Proceedings of the *29th Annual International Symposium on Microarchitecture*, December 1996.

[3] Sodani A., Sohi G. S., "An Empirical analysis of instruction repetition", Proceedings of the *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.