Memory disambiguation and speculation

Lecture #9: Wednesday, 26 April 2000
Lecturer: Ayodele Embry
Scribe: Brian Towles
Reviewer: Mattan Erez

1. A. Nicolau, "Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies", *IEEE Transactions on Computers*, Vol. 38 No. 5, May 1989.

2. G. Reinman and B. Calder, "Predictive Techniques for Aggressive Load Speculation", in Proceedings of the *31st International Symposium on Microarchitecture*, December 1998.

3. A. Moshovos and G. Sohi, "Streamlining Inter-operation Memory Communication via Data Dependence Prediction", in Proceedings of the *30th International Symposium on Microarchitecture*, December 1997.

1 Run-Time Disambiguation

Compilers perform code transformations in order to expose the maximum amount of parallelism to the target processor. While these techniques can be effective, they are often limited by the fact that the compiler must ensure semantic correctness of the program. This is especially true for memory accesses: if the compiler cannot guarantee that memory locations are independent it must enforce strict ordering of accesses to these locations. This is known as the problem of *memory anti-aliasing* or *disambiguation*. As noted in the paper, the ability to perform accurate disambiguation of indirect references (pointers) is crucial for a parallelizing compiler.

Prior parallelizing compilers, such as Bulldog, use purely static or compile-time techniques for memory disambiguation. However, the authors are quick to point out the shortcomings and potential computational expense of these methods to produce results, if they produce results at all. This observation motivates run-time disambiguation in which the burden of disambiguation is partly shifted from compile-time to run-time. All decisions are still performed statically, but run-time tests ensure correctness. In essence, rather than requiring the compiler to be *always right*, run-time disambiguation weakens this condition, so that the compiler only needs to be *usually right*.

Two issues with run-time disambiguation were discussed in class: (1) semantic correctness and (2) code explosion. Ensuring semantic correctness guarantees that the program will execute the same before and after run-time disambiguation optimizations

are inserted. In the case presented in the paper, changing dependencies in one trace can "break" dependencies in other traces. The solution is to detect dependencies dynamically and if they exist branch out of the trace to fix-up code and then rejoin.

Code explosion, associated with the additional fix-up code required by run-time disambiguation is another potential drawback. However, the paper mentions several techniques for minimizing the impact of this. First, run-time disambiguation is only applied when it helps - i.e. when it exposes more parallelism in the underlying code, allowing a more compact schedule. Also, since traces are often built using profiling information, run-time disambiguation could be applied to those traces that are executed the most frequently. These first two techniques fall under what the paper refers to as *trace reduction*. Another method, called *assertion unification* combines several disambiguation assertions such that a single piece of fix-up code is used if one of many memory aliases is detected.

Performance of run-time disambiguation is evaluated using the Bulldog research compiler and significant speedups are observed over a small set of algorithms/numeric applications.

2 Load Speculation

Load latency is identified as a major bottleneck in modern superscalar processors. This paper examines the interaction and performance tradeoffs of four techniques designed to deal with this problem: (1) dependence prediction, (2) address prediction, (3) value prediction, and (4) memory renaming. Before discussing these methods, the class noted the importance of the two mis-speculation recovery techniques presented in the paper. The *squash* approach invalidates all instructions that occur after the misprediction point. The *re-execute* approach only reissues those instructions dependent on the mis-result. Note that it may take several iterations before all instructions indirectly dependent on a mis-result are detected and reissued. In a highly speculative system, the performance increase from using the more complex re-execute policy can be significant.

Without speculation, modern processors must allow the addresses of all pending stores to be resolved before satisfying a waiting load request. In dependence prediction, a load can be predicted as either being independent of prior stores or dependent on a particular store. The most naive form of this prediction discussed is *blind prediction*. In blind prediction, loads are always assumed to be independent of previous stores. If a store's address is later resolved and conflicts with this assumption, the load is reissued. *Wait prediction*, a single bit is added to each instruction in the instruction cache. This bit is initially cleared, indicating that if the instruction is a load, it may issue as soon as possible (as in blind). However, if a misprediction is later detected, the wait bit is set for that load instruction. When a load with a set wait bit executed, it waits for all pending stores to be resolved before issuing. Finally, *store sets* attempt to build the sets of loads and stores that alias to the same memory location. Memory operations are given store set id's, such that those operations aliasing to the same location have the same id. Load dependencies can then be resolved by checking for stores with the same id. For squash

recovery, the techniques have increasing performance in the order presented here. For re-execution recovery, the techniques have roughly similar performance.

Address and data value prediction attempt to resolve memory dependencies early by either predicting the address of a load or by simply predicting the result of a load, respectively. The paper explores several techniques for both of these approaches. *Last value prediction* guesses that a the next value for a particular load (either the address or data) will be the same as the last value. A *stride predictor* tracks the last value plus a stride - the difference between the last and one before last value. The next predicted value is the last value plus the stride. A *context predictor* chooses between the last several values seen for a load. All of these single predictors use confidence counters to decide whether or not to predict, and in the case of the context predictor what value to use. The *hybrid predictor* presented in the paper combines a context predictor and a stride predictor plus confidence counters to guide selection between the two. Results showed the expected trends for both address and data prediction: the more complex the predictor, the more accurate it is. Although the addresses could be more accurately predicted, data value prediction provided "more bang for the buck" in that the memory access could be skipped (of course the loads must still actually occur to confirm the prediction).

Memory renaming identifies load/store dependencies in order to directly communicate a value from a store to its corresponding load, bypassing memory. Recently executed stores are cached and if a later load hits in this cache, the load/store dependency is recorded. If the load is executed again, it's value is predicted from the store it was paired with. In essence, this load has been renamed so that it does not conflict with other loads or stores being executed, eliminating potential false dependencies. Two implementations of this idea are discussed. *Original renaming*, records the address of stores in a store address cache and their values into a value file. When a load hits in the store address cache, the next prediction of the load is the value file entry associated with that particular store. In *merging renaming*, the authors try to apply the idea of store sets to memory renaming. Instead of a single load/store pair being identified, the original renaming technique is extended to clusters of loads and stores. Both of these approaches provided only modest speedups and merging renaming performed generally worse.

Finally, the paper investigates the interaction between all of the different prediction schemes. The class concluded that the benefits of value prediction seemed to "swamp" that of the other prediction methods. Mattan decided to speculate further, noting the relative unimportance of the other methods might be due to the short SimpleScalar pipeline used in the paper's simulations.

3 Data Dependence Prediction

In this paper, the authors explore speculative techniques for turning implicit communication between instructions into explicit communication. Specifically, they are concerned about communication through the memory name space (i.e. memory). To streamline this, dependencies must be established as quickly as possible and storage structures that

best meet the communication requires must be provided. *Data dependence prediction* establishes dependencies based on history information. Then dependent loads and stores communicate through a speculative name space derived from these dependencies without incurring the overhead of address calculation, disambiguation, and memory access. Of course, since this technique is speculative, the communication must eventually be verified through the memory system.

In class, we briefly discussed the proposed techniques for implementing data dependence prediction. In *speculative memory cloaking*, synonyms are dynamically associated with load/store pairs and are stored in a synonym file (cache). The operation of this structure begins with a store/load dependence to an address in memory, which is detected as stored as an association. When a later dynamic instance of the same store is executed, the store value is allocated an entry in the synonym file. If the load associated with that store is then executed, its association with the previous store is used to index the synonym file and access the value of the store. This value is speculatively used as the result of the load. The author's stated that cloaking still requires that the load is verified through the memory system, so while the load latency may be avoided through speculation, no memory bandwidth is saved. However, in class it was noted that only the address of the speculative load needs to be verified. So, it does seem that memory bandwith can be saved in the cases where the speculation is correct.

An addition to speculative memory cloaking is *speculative memory bypassing* in which load/store pairs that coexist in the instruction window are speculatively detected and matched. In this case, def-store-load-use chains are converted def-use chains. Finally, the *transient value cache* (TVC) is suggested to eliminate wasted memory bandwidth for short-lived memory values (i.e. ones that are quickly killed by another store). The TVC operates in parallel to the L1 cache, but for accesses that are likely to be quickly killed. So, stores with predicted output dependencies are sent only to the TVC and loads with predicted true dependencies first check the TVC before accessing the L1 cache. Also, evictions from the TVC are directed to the L1 cache.

While the techniques presented did provide fairly accurate prediction of dependencies, the overall IPC speedup proved to be modest. Perhaps the most interesting result was the potential reduction in memory system traffic produced by the transient value cache.