

Applying Data Speculation to Loop Invariant Code Motion

Giselle Agosto, Robert Mullenix, and William Yeung
{gagosto rbmullen [willay](mailto:willay@umich.edu)}@umich.edu

I. INTRODUCTION

Data speculation is a compiler technique that aggressively removes the dependence edges between memory operations to allow optimizations normally prohibited. Because the compiler often cannot determine conclusively whether the addresses will alias, memory operations are often treated conservatively. If, through memory profiling, the compiler can identify memory operations where a false dependency exists, data speculation can allow a relaxation of the restriction to take place. As long as recovery code exists in the event of a misspeculation (i.e. the two addresses did in fact alias), correct execution can be guaranteed.

For our advanced compilers course, we examined the benefit of applying data speculation to loop invariant code motion. This optimization would allow loads with invariant addresses to move outside of the loop body, even in the presence of a store. As far as we can tell, no literature exists that employs data speculation in this fashion.

II. BACKGROUND

Our work for this project is based on previous work done in the University of Minnesota[1]. They used data speculation to perform classical optimizations like copy propagation, dead store elimination, and redundancy elimination on memory operations. Currently, most compilers

conservatively assume there's a dependency between every load and store, so potential optimizations opportunities are usually missed. Most of the previous work on data speculation has been based on scheduling and register allocation and this Minnesota paper is the first work to actually use it to enable optimizations. They perform memory profiling to determine which memory operations are most likely to alias. Any dependency with a low probability is ignored. Then they proceed to do a specialized code motion across other memory operations of the memory operations that were not likely to alias. This code motion enables many types of optimizations that were previously impossible. Each of these movements is speculative because, there's always a small chance that they could alias at run time. In order to ensure program correctness, the authors of the Minnesota paper propose inserting a check instruction in the original position of the instruction that was moved. This check will verify if there has been an alias and will jump to a block of recovery code.

In order to verify if there has been an alias at runtime, hardware support is needed. The authors of the Minnesota paper use the IA-64 architecture which provides support for data speculation with the addition of the Advanced Load Address Table (ALAT). This small table acts as a fully associative address cache. In the Itanium2, the ALAT has 32 entries. A set of new instructions was added to the ISA to interface with this table.

During compilation, the speculative load candidate is replaced with an advanced load (ld.a) and the dependence edge resulting from a potential alias is removed. In addition, the compiler inserts a check instruction (chk) underneath the load. It is prohibited from moving. It takes the destination register of load as a source and also includes a target to the recovery code block. This advanced load is now free to move above the store.

During runtime, when the ld.a is executed, the ALAT stores the address, size (e.g. byte, half-word, word, long, etc.), and the destination register. It marks that entry as valid. On every store, the ALAT performs a table lookup using the store's address as a key. If a hit occurs in the table (i.e. the address, or address range, matches and the entry is valid), the ALAT invalidates that entry. When execution reaches the check instruction, the ALAT performs another table lookup, this time using the destination register as they key. If a hit occurs, the valid bit is examined. If the entry is no longer valid, then a memory alias occurred and the speculation was too aggressive. The check then branches to the recovery code where the load is re-executed.

III. IMPLEMENTATION

For our project, we decided to use data speculation to implement an optimization that was not discussed in the previous work. We decided to use data speculation in order to move potentially invariant loads outside of a loop. We used the Trimaran Research Compiler [1] as our framework. Although Trimaran has some support for data speculation implemented, we discovered that the existing code was largely inoperable. So we implemented our own data speculation support.

We divided the work into two problems: modifications to the hardware simulator (simu) to facilitate the address lookup, and additions to the backend compiler (elcor) that would actually perform the data speculation.

Backend additions

The additions to elcor had four main purposes: use memory profiling to determine aliasing probabilities between loads and stores, speculatively move load instructions outside the loop, generate a recovery code block for each moved load and insert a check instruction in to the original loop code. In order to collect the number of potential aliases for each load, we use Trimaran's built-in memory profiling. We then calculate the probability of aliasing by dividing this number by the number of loop iterations. If this probability is less than a predetermined threshold that load is marked as "movable". Afterwards we proceed to analyze the load for register dependences and liveness, ignoring potential memory dependences with any stores that are also in the loop. If the load passes all the analysis checks, then it will be moved to the header of the loop. A noop instruction is then inserted in the original position of the load, in order to mark this location for later purposes. A recovery block that will hold the same load instruction that was moved to the loop header will be created and identified with the id of the added noop. After the whole loop has been analyzed and all the loop invariant instructions have been moved up, the loop will be scanned in order to add check instructions and predicated jumps to the recovery block in the appropriate places. The checks are basically a new operation that we created and called LSC. It functions similarly to a cmpp because its destination is a predicate register that is set whenever there's an alias. This LSC instruction will be inserted where the original noop was inserted and will be followed by a predicated branch to the

recovery code. The branch's predicate will be the destination predicate of the LSC. That way we guarantee that whenever there's an alias, we will branch into the recovery code. The main loop block must then be split right after the predicated branch and the control edges updated, so that we can jump to the recovery code in case of an alias or continue the regular program flow when there's no alias. An LSC instruction and predicated branch must also be added in to every postloop block to ensure that any aliasing caused in the last loop iteration is covered.

Simulator additions

In order for the LSC operations to set the branch predicate to false whenever there's aliasing, some hardware support is needed. Inside the Trimaran simulator, we added code that approximated the IA-64 data speculation support with a few modifications.

In our version of the ALAT, we added an additional bit to mark an entry as open (i.e. available for insertion). That guaranteed no chance for squashing the address of speculated load once added to the table before the associated check instruction could inspect the entry and mark it as available. If the ALAT filled up, the data speculative load would just not get added to the table. During each store, the operation worked exactly as in the IA-64.

For the check instruction, we first had to create it inside Trimaran. We based our check instruction on a predicate setting compare operation and then guarded a branch-and-link to the recovery code with that predicate. Once the operation was implemented, we needed to modify the behavior of the check from the IA-64 to allow for the proper function in during loop invariant code motion.

One key nuance of data speculation is the need to place another speculative load into the recovery block rather than a nonspeculative one. If the load aliased inside the loop body, we cannot guarantee that it will not happen in future iterations. By marking the load inside the recovery block as speculative, we can continue speculating the load despite a single alias.

Instead of clearing the ALAT, the check needed to keep the entry alive in the table in case of a future alias. If the entry was cleared, the check would execute the recovery block unnecessarily (this would only happen once, since a speculative load would reenter the address into the ALAT).

If the check instruction found an invalid hit in the table, only then should it clear the ALAT entry, since execution of recovery code would then put the same speculative address back into the ALAT. This is to prevent polluting the ALAT with duplicates.

Finally, once the loop exits, one final check needs to be called in the post loop block to clear that entry in the ALAT. Otherwise we potentially issue a speculative load (inside a recovery block or in the pre-header if no alias occurs) that never gets cleared by a check. So we added another flag inside the simulator to support this.

We had a great deal of difficulty in modifying the statistics gathering code for Trimaran to handle our check instruction and use the correct amount of delays. We were unable to get accurate statistics from the simulator, although we did manage reconstruct reasonable data through detailed trace analysis and observation of the ALAT performance (number of accesses, hits, and recovery requests performed). The next section shows and discusses our findings.

IV. RESULTS

We run our simulation on nine benchmarks, and we compared the total number of simulated cycles, the number of dynamic operations, and the number of load operations the benchmarks executed. From figure 1 and figure 2, we can observe that our optimizations are not getting any significant performance improvement, instead for most of the benchmarks we have run, our modification made the simulations take more cycles and dynamic operations. This result contradicts the diagrams in figures 3 and 4, where we observe that we have reduced the number of loads by 20~30% in average and with a best case of 66%. One of the main reasons is that we are replacing one load operation with one check operation, but both operations are taking the same amount of cycles. Since we are not moving any other possible non-load loop invariant operations into our recovery block, we are simply not getting any optimization on cycle or operation count. Then when the programs flow into the recovery block, we are simply getting extra cycles. This explains why we have reduced number of load operations but the number of cycles still increased. We ran our simulation on some benchmarks with floating point load operations which take two cycles instead of one, and we prove that our reasoning is correct. The performance is improved and is directly proportional to the number of loads we reduced and inversely proportional to the number of times we have to recover.

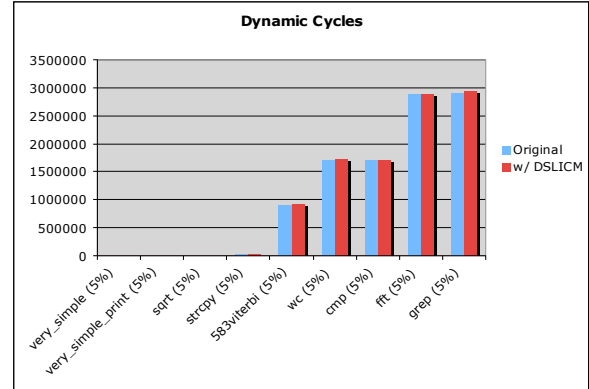


Figure 1 Dynamic Cycles

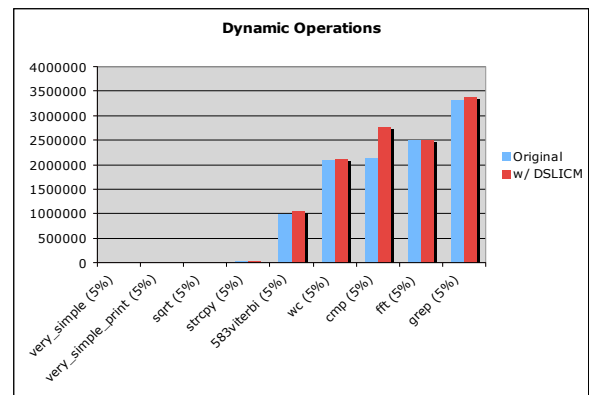


Figure 2 Dynamic Operations

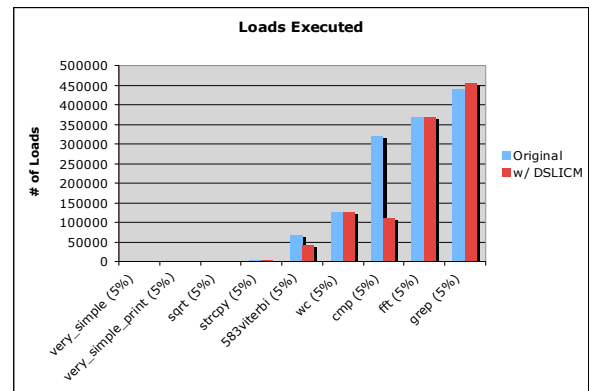


Figure 3 Number of Load Executed

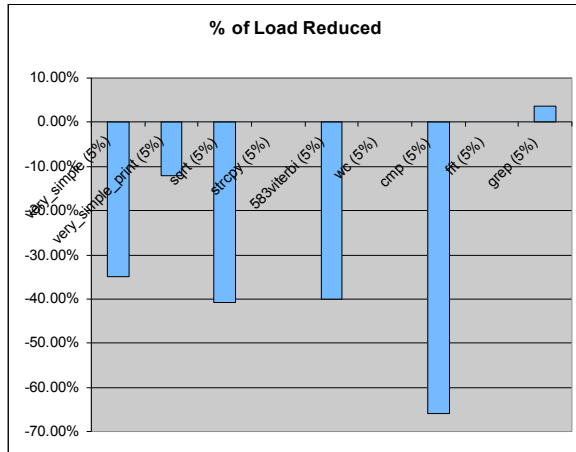


Figure 4 Percent of Load Operations Reduced

V. CONCLUSION

From the results we got, we can conclude that if the latency of load operation is exactly the same as a check operation and we are only moving the load out of the loop, we cannot get any improvement from data speculation, but with a good threshold value, the overhead of recovery codes does not downgrade our performance very much. Hence, if the load operation is more expensive than a check operation, we will break the tie and probably have good performance improvements. If more invariant code, especially high latency instructions like multiplies and divides, can be moved out of the loop, there will definitely be performance gain.

Because of the limited timeframe and complexity of modifying the ELCOR backend compiler, we weren't able to extend our data speculative loop invariant code motion algorithm to move invariant operations dependent on the speculative loads.

VI. REFERENCES

- [1] X. Dai, A. Zhai, W.C. Hsu, P.C. Yew, "A General Compiler Framework for Speculative Optimizations Using Data Speculative Code

Motion", in Proceedings of the International Symposium on Code Generation and Optimization (CGO), 2005

- [2] Intel Itanium Processor Hardware Developer's Manual. August 2001.

<http://www.intel.com/design/itanium/downloads/2487012.pdf>

- [3] Trimaran Research Compiler. April 2006. <http://www.trimaran.org/>