Reducing Cache Pollution at Compile Time with Static Reuse Distance Analysis and Insertion of Non-temporal Memory Instructions

William Liu[*]
wxl@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania

Weihang (Frank) Fan[*]
wfan@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania

ABSTRACT

Modern processors have special non-temporal store instructions to save memory bandwidth on streaming store operations – where each element of a data structure is written to only once before it is evicted from the cache hierarchy. In this work, we provide details on the compiler-based generation of these instructions via static analysis of memory accesses in loops. We also evaluate their impact on the performance of a variety of streaming benchmarks. Our results show that in many cases, our framework is able to insert non-temporal store instructions in the same code locations as in hand-optimized code and achieve speedup results in total execution time.

CCS CONCEPTS

• **Software and its engineering** → *Compilers.*

KEYWORDS

static analysis, cache optimization, non-temporal instructions

ACM Reference Format:
William Liu and Weihang (Frank) Fan. 2020. Reducing Cache Pollution at Compile Time with Static Reuse Distance Analysis and Insertion of Non-temporal Memory Instructions. In *Proceedings of (15-745).* ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

Typical store instructions place cache lines back into the highest level of the cache hierarchy and allow it to travel down the hierarchy before it is ultimately evicted. In most use cases this is an ideal approach such as if those cache lines have temporal reuse then they can be quickly accessed again rather than incurring a large latency penalty due to fetching from a lower level in the cache hierarchy. In a data streaming scenario, however, all of the cache lines that are hit are guaranteed to have no temporal reuse. For data streaming, storing a cache line back into the cache hierarchy would not be useful and simply introduces unnecessary data into the cache.

[*]Both authors contributed equally to this research.

For these types of data streaming behaviors a programmer can take advantage of the non-temporal store instructions introduced in the Streaming SIMD Extension standard (SSE) [1]. Rather than storing a cache line back into the L1 cache when the processor is finished using the data, a non-temporal store will bypass the cache hierarchy entirely and store the cache line directly back into main memory.

This behavior of the non-temporal store is beneficial mainly for the reason that it allows cache lines with no temporal reuse to not pollute the cache. If a cache line with no temporal reuse is stored back into the top of the cache hierarchy then, using an LRU-like eviction policy, that cache line will have to be evicted from every level of the cache hierarchy before making it back to main memory. This causes two issues which are resolve through non-temporal store instructions: there is increased bus traffic as more data must travel along the memory hierarchy, and the cache line will take up needlessly extra space in the cache will allow less space for data with temporal reuse causing more misses and again more bus traffic. Overall switching regular store instructions to non-temporal store instructions when the cache lines are guaranteed to have no temporal reuse will decrease the total pressure on the memory bandwidth.

Previous works that look directly at reducing cache pollution with regards to using non-temporal memory instructions have been primarily based on using statistical or analytical models in order to determine which memory accesses have no temporal reuse and thus where to insert the non-temporal memory instructions [3, 4, 10]. Other works have proposed dynamic profiling techniques that characterize reuse distance, which is the primary metric in determining whether or not a non-temporal instruction can be inserted for optimization [6, 7, 9].

The types of applications that benefit the most from using non-temporal store instructions will involve streaming large amounts of data that has no temporal reuse. Due to this, trace-based and dynamic analysis and optimization methods may suffer from a high startup cost as the unoptimized program will require streaming through a large amount of data before any optimizations can take place.

In this paper we propose a framework for identifying store instructions as candidates for swapping to a non-temporal store using global reuse distance between memory accesses to the same location. This analysis and code transformation derives reuse distance from the structure of the code rather and can run at compile time than at runtime. Performing initial optimizations before the program runs for the first time will allow for a decreased strain on the memory bandwidth and total execution time of the program immediately.

The main contribution of this work is providing a static analysis framework for approximating global reuse distance which requires low asymptotic work to insert non-temporal store instructions and reduce cache pollution. The proposed framework fits well into the existing trace-based and dynamic analysis literature by providing an already optimized starting point for those existing optimization frameworks.

2 GLOBAL DATA REUSE ANALYSIS

This section describes in detail the individual components of the static analysis framework: the data-flow analysis and the static global reuse distance analysis.

2.1 Data-flow Analysis

Upon initialization, the pass first collects every memory access instruction across the entire program. Using the set of all memory access instructions, we use a simple data-flow analysis to determine which memory access instructions can possibly have temporal reuse with one another. To do so, we collect, at every basic block, the set of load and store instructions that come before and at that basic block.

The domain of the data-flow analysis is the set of load and store instructions found within the program. The analysis is a backwards flowing analysis with the union operation as the meet operator.

$\top$ is the empty set and $\perp$ is the complete set. Since the domain of the data-flow analysis is just the memory access instructions and the height of the lattice is very short, this data-flow analysis is computationally cheap to run, takes up very little space asymptotically, and converges quickly.

2.2 Static Global Reuse Distance Analysis

For each pair of memory accesses that could have temporal reuse with respect to each other, we first check whether or not they access the same underlying memory location (eg. difference indices of one array). We can track this using the LLVM Value Tracking analysis. If two memory accesses do reference the same location, then we attempt to statically compute the minimum reuse distance between those two accesses. If two accesses to the same memory location have a reuse distance larger than the total memory of the cache hierarchy of the processor, then using a regular store instruction at the former access would guarantee that the cache line containing the memory location would be fully evicted to main memory before the latter access.

In order to statically compute the distance between two accesses, we look at all the loops between those two accesses and sum their working set sizes/cache footprints. This can be compute by multiplying their trip counts with the amount of data accessed per iteration. This will only work with loops that access a constant amount of data per iteration, which is generally the case with data streaming applications. For two accesses that are within the same (inner) loop, we take advantage of LLVM's Loop Cache Analysis [2], which uses loop dependency analysis and the SCalar EVolution (SCEV) alias analysis to determine if there is temporal reuse between two accesses in a loop within a certain maximum reuse distance.

Our analysis does not consider accesses that occur outside of loops such as singleton memory accesses. Since a non-temporal store instruction optimization only makes sense for a reuse distance as large as the entire cache hierarchy, a since singleton accesses into a typical array of data will contribute less than 10 thousandth of a percent to the total reuse distance between two memory accesses. Additionally if a singleton store instruction were to be transformed into a non-temporal store then the execution time and memory bandwidth benefits would like be unnoticeable.

Similarly to the data-flow analysis from the previous section, this static global reuse distance analysis is also not very asymptotically complex. While the cache footprint computations are not trivial, the total amount of iterations run scales at an $O(N^2)$ rate with respect to the number of memory access instructions in the entire program and takes no additional space as we do not need to store the reuse distance results.

2.3 The Evolution of Our Design

At the proposal stage of this project we knew we needed some form of alias analysis to determine whether or not two memory accesses would collide. So then our initial approach included using sophisticated range analysis implementations such as [8, 11] to do alias analysis within loops. The primary issue with this approach was that while information about temporal reuse of memory accesses can be derived with range analysis, it did not provide any information about whether or not a non-temporal memory instruction could be inserted.

Rather than relying on some complex range analysis, we needed to consider what fundamental information was actually required in order to make the decision that a regular store instruction could be swapped with a non-temporal store instruction. To make that decision, we needed to know, for each store instruction in the program, how "soon" a subsequent memory access instruction touched the same memory location. Once the fundamental design converged upon approximating the distance, in terms of memory usage, between two memory accesses to the same location we began looking into reuse distance literature.

After deciding on the final version of the overall approach to do the static analysis of reuse distance, we realized that the original plan with range analysis was not necessary as calculating the reuse distance required global analysis whereas the range analysis frameworks only performed analysis local to each loop.

3 INSERTION OF NON-TEMPORAL STORE INSTRUCTIONS

During the pair-wise reuse distance analysis discussed in the previous section, whenever a store instruction is found to have no temporal reuse it can be swapped for a non-temporal store instruction. The insertion of the non-temporal instruction involves adding a `!nontemporal` metadata hint to the instruction in the LLVM IR. This metadata hint allows the backend code generator to select the non-temporal version of the instruction. For example instead of `vmovaps` to store packed floating point values in SIMD vectorized code, the code generator can select `vmovntps`.

It is worth noting that the set of non-temporal store instructions on modern Intel and AMD processors is limited, especially for scalar stores [1]. In particular, there is no instruction for the non-temporal store of a floating point scalar in the Streaming SIMD Extensions (SSE) standard. One only exists (`movntsd`) in the SSE4a extension, which is specific to AMD. While the integer version of the instruction (`movntq`) can be used for storing floating points, LLVM's instruction selector does not elect to do so.

Overall the behavior of the instruction selector with regards to using non-temporal memory instructions seems to be quite conservative. It was difficult to force the instruction selector to select the non-temporal version of a store instruction in some of our experiments. This is perhaps an expected result as recklessly using non-temporal memory instructions can have disastrous effects on the memory behavior and total execution time of a program when used incorrectly.

4 EXPERIMENTAL SETUP

We implemented our analysis and transformation pass in the LLVM compiler framework on version 10.0. All of experiments were performed on a computer with an AMD Ryzen 2700X CPU (`znver1` architecture) and 16GB of memory, running Debian 10.

For each of the experiments, the LLVM IR bitcode files were generated using Clang 10.0 with the command line options `clang -O3 -fno-unroll-loops`. Unrolled loops presented complications for our analysis as well as caused worse performance in our experiments (likely due to increased cache contention leading to more cache misses).

The analysis and transformation pass was then run on the LLVM bitcode file to insert the non-temporal store instructions. Assembly code for debugging and evaluation was compiled using `llc -O3 -mattr=+sse4a` to take advantage of the AMD-specific non-temporal store instructions. Note that the optimal binaries which we used for our results will not run on Intel processors due to the SSE4a-specific instructions.

5 EXPERIMENTAL EVALUATION

In general, the optimization pass was able to insert non-temporal instructions in the same locations as hand-optimized code and resulted in an overall speedup in program execution time.

5.1 SAXPY Microbenchmark

The single-precision floating point version of the generalized vector addition micro-benchmark, commonly referred to as SAXPY, consists of performing a multiply and an add operation on each element of two arrays, as shown in Listing 1. SAXPY is designed to be memory-bound and completely saturate the memory bandwidth of the cores it is running on. We evaluated our optimization on a manually vectorized version of the benchmark, using 256-bit wide AVX SIMD instructions but no non-temporal ones. Indeed, the optimization was able to convert the store to `y[i]` (the only store in the loop) to a non-temporal store, as it stores to a distinct location each iteration.

Figure 1 shows the execution times for each version of the applied optimizations. With no AVX SIMD instructions, the total execution time of the program was the longest as expected. Since this SAXPY code is memory throughput bound, the version with SIMD instructions does not achieve anywhere close to ideal speedup compared to the fully sequential version of the code.

With the non-temporal optimization, the optimized program achieves 30% and 36% over the serial and SIMD versions respectively. This brings the total speedup result for SIMD plus non-temporal to 45% over the regular sequential version of the program.

For our experiment, the SAXPY program was run for a total of 3 trials. In each trial the SAXPY program streamed twenty million floats for a total of 80 megabytes of data. This should allow the cache to hit a steady state and reduce the variations caused by noise.

Listings 2 and 3 show the regular and optimized generated assembly for the serial version respectively. We can observe that a `movntss` instruction is selected by the backend code generator instead of a `vmovss` instruction as expected.

```
void saxpy(int N, float a, float* x, float* y) {
  for (int i = 0; i < N; i++){
    y[i] = a * x[i] + y[i];
  }
}
```

Listing 1: SAXPY Main Loop

```
    vmulss  (%rsi,%rdi,4), %xmm0, %xmm1
    vaddss  (%rdx,%rdi,4), %xmm1, %xmm1
    vmovss  %xmm1, (%rcx,%rdi,4)
    incq    %rdi
    cmpq    %rdi, %rax
```

Listing 2: SAXPY Main Loop Excerpt (Original Assembly)

```
    vmulss  (%rsi,%rdi,4), %xmm0, %xmm1
    vaddss  (%rdx,%rdi,4), %xmm1, %xmm1
    movntss %xmm1, (%rcx,%rdi,4)
    incq    %rdi
    cmpq    %rdi, %rax
```

Listing 3: SAXPY Main Loop Excerpt (Optimized Assembly)

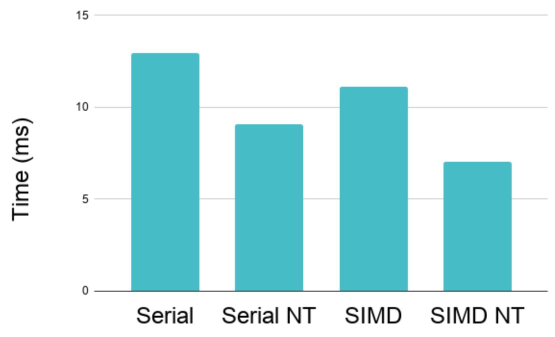

Figure 1: Execution Time Comparison for SAXPY

```
for (int m = 0; m < Nm; m++) {
  for (int g = 0; g < Ng; g++) {
    for (int l = 0; l < Nl; l++) {
      for (int k = 0; k < Nk; k++) {
        for (int j = 0; j < Nj; j++) {
          double total = 0.0;
          /* prefetch from q */
          _mm_prefetch((const char*)
            &q[m][g][l][k][j][0]);
          for (int v = 0; v < VLEN; v++) {
            /* Set r */
            r[m][g][l][k][j][v] = ...;

            /* Update x, y and z */
            x[m][g][k][j][v] = ...;
            y[m][g][l][j][v] = ...;
            z[m][g][l][k][v] = ...;

            /* Reduce over Ni */
            total += r[m][g][l][k][j][v];
          } /* VLEN */

          sum[m][l][k][j] += total;

        } /* Nj */
      } /* Nk */
    } /* Nl */
  } /* Ng */
} /* Nm */
```

Listing 4: Mega-Stream Kernel Main Loop Pseudo-code

```
...
movsd   (%rdi,%r11), %xmm6
mulsd   %xmm3, %xmm6
addsd   %xmm5, %xmm6
movsd   %xmm6, (%r10,%r11)
addsd   %xmm6, %xmm1
mulsd   %xmm0, %xmm6
movapd  %xmm6, %xmm5
...
```

Listing 5: Mega-Stream Kernel Main Loop Excerpt (Original Assembly)

5.2 Mega-stream Benchmark

The mega-stream benchmark [5] is a streaming benchmark designed to saturate the memory bandwidth on a machine. The benchmark is intended to be compiled with the Intel C Compiler (ICC), though as mentioned in the Experimental Setup section we compiled it with Clang and disabled multi-threaded parallelism from OpenMP. We ran it with the default configuration, which involves arrays of dimension $128 \times 16 \times 16 \times 16 \times 64$.

Listing 4 shows a partially redacted version (for clarity) of mega-stream's main kernel function. This benchmark is also intended

```
...
movsd   (%r14,%r8), %xmm6
mulsd   %xmm3, %xmm6
addsd   %xmm5, %xmm6
movntsd %xmm6, (%rdx,%r8)
addsd   %xmm6, %xmm1
mulsd   %xmm0, %xmm6
movapd  %xmm6, %xmm5
...
```

Listing 6: Mega-Stream Kernel Main Loop Excerpt (Optimized Assembly)

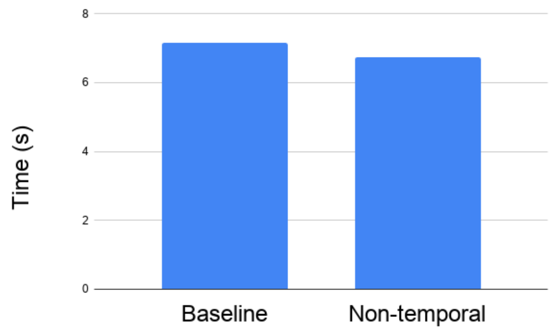

Figure 2: Execution Time Comparison for Mega-Stream

to saturate the memory bandwidth of a processor with multiple high-dimension arrays and nested loops.

Figure 2 shows the execution times for the regular and non-temporal versions of the mega-stream execution. With regular store instructions, the total execution time of the program is slightly longer as expected. The non-temporal version of the mega-stream execution achieves a 6% improvement in the total execution time over the regular version.

For our experiment, we ran 1 trial of each version of the mega-stream binary. Within each whole program execution, the main loop is run 100 times. Within each run of the main loop, the mega-stream kernel streams over 589.3 megabytes of data. This should allow the cache to hit a steady state and reduce the variations caused by noise.

Compared to the SAXPY microbenchmark, while we still see some execution time improvements they are not as significant. This is because the mega-stream benchmark deals with 8 arrays, only one of which exhibits non-temporal characteristics and can thus be optimized by our pass. So while the non-temporal optimization decreases the total pressure on the memory bandwidth, we do not see a significant speedup in the total program execution time.

Listings 5 and 6 show an excerpt from the same location in the regular and optimized versions of the assembly code respectively. It is interesting to note that in the SAXPY experiment the register allocation does not change between the regular and non-temporal versions of the code. However between listings 5 and 6, there is a clear difference in the scalar register allocation.

6 SURPRISES AND LESSONS LEARNED

We were pleasantly surprised by the amount of symbolic analysis for loops that already exist in the LLVM compiler infrastructure, such as the Scalar Evolution (SCEV) framework, the dependency analysis pass, and alias analysis passes. Not having to "re-invent the wheel" for all of the smaller components of the project allowed us to focus more directly on the fundamental conceptual underpinnings of our optimization pass.

We were unpleasantly surprised by the difficulty of the LLVM instruction selector in generating non-temporal memory instructions. If the original data was in an `xmm/ymm/zmm` register for which there is no scalar non-temporal store instruction in the standard SSE/AVX instruction set, LLVM simply gives up instead of trying to allocate the data to a general-purpose register instead. The only workaround we found other than directly modifying the instruction selector behavior was to use SSE4a instructions on the an AMD processor.

7 CONCLUSION

This paper presents a static analysis framework for optimizing data streaming programs at compile time by combining data-flow analysis with static global data reuse analysis. The main contribution of this work is validating the the effectiveness of cheap static analysis for reducing cache pollution. The empirical results presented in this paper show that static analysis can be effective at approximating global reuse distance and for optimizing non-temporal memory instructions.

8 FUTURE WORK

The clearest direction for future work from this work is considerations for dynamic optimization in addition to the findings presented in this paper. Specifically, for programs where we are unable to statically determine the reuse distance, an optimizing compiler can generate two branches based on whether the reuse distance is large enough or not, with one using non-temporal instructions and one not.

In addition, another clear direction of future work would be to incorporate non-temporal load instructions (e.g. for prefetching) as well. A non-temporal prefetch instruction achieves the same end goal of not polluting the cache hierarchy by loading cache lines only into the L1 cache and not any other levels in the hierarchy. For the purposes of this work, to demonstrate the effectiveness of static analysis on non-temporal optimization, we elected to not introduce the extra complexity of using non-temporal load instructions. However the same static global reuse distance analysis should work effectively for non-temporal load optimizations as well.

A limitation in our work currently is that we lose some level of granularity by running our analysis using LLVM's `FunctionPass` class. Due to this, the static global reuse distance analysis for instructions occurring at the end of a function may not be completely accurate. However, because of the large reuse distance required for optimizing with non-temporal store instructions, for the experiments in this paper this was not an issue. For future iterations of this work, switching over to an analysis pass that runs over the entire program at once and is context-aware would provide a more detailed analysis that can optimize more aggressively.

9 DISTRIBUTION OF TOTAL CREDIT

Frank: 50%, William: 50%.

ACKNOWLEDGMENTS

To Professor Todd Mowry, for the advice throughout the project. And to my coffee maker.

REFERENCES

[1] [n.d.]. Intel® Instruction Set Extensions Technology. https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html Library Catalog: www.intel.com.

[2] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler optimizations for improving data locality. *ACM SIGOPS Operating Systems Review* 28, 5 (Nov. 1994), 252–262. https://doi.org/10.1145/381792.195557

[3] J. P. Casmira and D. R. Kaeli. 1998. Modelling Cache Pollution. *International Journal of Modelling and Simulation* 18, 2 (Jan. 1998), 132–138. https://doi.org/10.1080/02286203.1998.11760369 Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/02286203.1998.11760369.

[4] Tom Deakin, Wayne Gaudin, and Simon McIntosh-Smith. 2017. On the mitigation of cache hostile memory access patterns on many-core CPU architectures. In *High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Revised Selected Papers*. Springer, 348–362. https://doi.org/10.1007/978-3-319-67630-2_26

[5] Tom Deakin, John Pennycook, Andrew Mallinson, Wayne Gaudin, and Simon McIntosh-Smith. 2017. The MEGA-STREAM benchmark on Intel Xeon Phi processors (Knights Landing). https://research-information.bris.ac.uk/explore/en/publications/the-megastream-benchmark-on-intel-xeon-phi-processors-knights-landing(1568f3ab-9655-411d-a42c-cf6b580d6118).html

[6] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03)*. Association for Computing Machinery, San Diego, California, USA, 245–257. https://doi.org/10.1145/781131.781159

[7] Changpeng Fang, Steve Carr, Soner Önder, and Zhenlin Wang. 2006. Path-Based Reuse Distance Analysis. In *Compiler Construction (Lecture Notes in Computer Science)*, Alan Mycroft and Andreas Zeller (Eds.). Springer, Berlin, Heidelberg, 32–46. https://doi.org/10.1007/11688839_4

[8] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. 2016. Symbolic range analysis of pointers. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. Association for Computing Machinery, Barcelona, Spain, 171–181. https://doi.org/10.1145/2854038.2854050

[9] Pavlos Petoumenos, Georgios Keramidas, and Stefanos Kaxiras. 2009. Instruction-based reuse-distance prediction for effective cache management. In *and Simulation 2009 International Symposium on Systems, Architectures, Modeling*. 49–58. https://doi.org/10.1109/ICSAMOS.2009.5289241

[10] Andreas Sandberg, David Eklöv, and Erik Hagersten. 2010. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, New Orleans, LA, USA, 1–11. https://doi.org/10.1109/SC.2010.44

[11] Suan Hsi Yong and Susan Horwitz. 2004. Pointer-Range Analysis. In *Static Analysis (Lecture Notes in Computer Science)*, Roberto Giacobazzi (Ed.). Springer, Berlin, Heidelberg, 133–148. https://doi.org/10.1007/978-3-540-27864-1_12