EE482c Final Project: Stream Programs on Legacy Architectures

Chaiyasit Manovit, Zi-Bin Yang, John Kim and Sanjit Biswas
{cmanovit, zbyang, jjk12, sbiswas}@stanford.edu

June 6, 2002

1. Summary of project and findings

This project studied the mapping of stream processing onto legacy architectures. Cache memories were treated as SRF, into which data were loaded by prefetch instruction. By using cache, we could conveniently index into the SRF, and also blended the usage of the SRF with the scratchpad memory. However, we still liked to have data arranged sequentially or packed together in the memory to efficiently utilize memory bandwidth and cache capacity. Such related work was done at Utah called Impulse, which was a memory controller that does data gathering/scattering; but since we would like to limit this project to legacy architectures, we decided to emulate such functionalities in software by copying selected data into a new data structure. (This piece of code, however, is referred as Impulse later in the report.) Loading data into the cache with prefetch and the actual computation were made overlapped by software pipelining, which was manually scheduled. When the whole stream was too large to fit in the cache, it's strip-mined.

To apply these techniques, we manually transform the codes, and at the same time try to extend GCC to do some (mainly prefetch) automatically. We did our experiments with a couple of benchmarks on both the RSIM simulator and an actual hardware (Sun UltraSparc-III). Our experiment showed that all three techniques, namely, prefetching,

Impulse, and strip-mining, were effective (when used properly) for the matrix multiplication studied in detail in this project. The result suggested that the L1 cache, not the L2, should be used as the SRF, although L2 is still useful for double-buffering and/or carrying spilled data. However, we also see some limitation on the effectiveness of using cache as SRF, mainly because we lack more explicit control on cache content and current implementation of prefetch incurs quite a bit of overhead.

Table 1. The summary of mapping between legacy architectures and stream processing

Stream	Legacy Architecture
SRF/LRF/Scratchpad	L1 cache
secondary level SRF (optional)	L2 cache
stencil - data gathering/scattering	Impulse (SW gather/scatter)
strip-mining	Algorithmic techniques, e.g. loop tiling

2. Description of Our Work

2.1 Software Data Prefetching

Stream processors effectively hide memory latency by keeping all necessary data in a high bandwidth Stream Register File (SRF), which is referenced by computation kernels, thus eliminating the need for reactive caches. Although such SRFs do not exist on legacy architectures, most contemporary processors offer some form of prefetch instruction that performs a non-blocking load into either the L1 or L2 cache. By utilizing the exposed memory access patterns found in stream programs, the compiler can prefetch data prior to on-demand accesses to avoid costly cache misses penalties. We are able to expose this regularity by modeling streams as arrays of records which are accessed within loops using affine indices. This allows us to use GCC's existing induction variable analysis framework to insert prefetches within loop bodies for future iterations.

2.1.1 Prefetch insertion example

```
for (product=0, i=0; i<len; i++)
      product += a[i] * b[i]; (a)
```

As an example of loop-based prefetching, consider the code segment (a). This loop calculates the inner product of two vectors **a** and **b** (each element is 1 byte), in a manner similar to the innermost loop of a matrix multiplication. Using this code, we will examine the transformations and optimizations required to effectively hide memory latency.

```
for (product=0, i=0; i<len; i++) {
      product += a[i] * b[i];
      prefetch(&(a[i+l]);
      prefetch(&(b[i+l]);
}                                    (b)
```

The simple approach to prefetching in (b) suffers from several problems. First, we need not prefetch in every iteration of this loop, since each prefetch actually bring in multiple, say **l**, bytes (one cache line) into the cache. Although the extra prefetch operations are not illegal, they are unnecessary and thus degrade preformance. Assuming **a** and **b** are cache line aligned, prefetching should only be done on every **l** iteration. One solution to this problem is to surround the `prefetch` directive with a `if` condition that tests when `(i mod l = 0)` is true, but this such overhead would likely offset the benefits of prefetching and therefore should be avoided. A better solution is to unroll the loop.

```
for (product=0, i=0; i<len; i+=l) {
      product += a[i]  * b[i]   +
                 a[i+1]* b[i+1] +
                 ....
      prefetch(&(a[i+l]);
      prefetch(&(b[i+l]);
}                                  (c)
```

The code segment given in part (c) removes most cache misses and unnecessary prefetches but futher improvements are possible. Note that cache misses will occur during the first iteration of the loop since prefetches are never issued for the initial iteration. Also, unnecessary prefetches in the last iteration access data past the loop index boundary. Both of the above problems can be remedied using software pipelining, as shown in (d).

```
prefetch(&(a[3*l]));
prefetch(&(b[3*l]));
product += a[0]*b[0]+a[1]*b[1]+....
product = 0;
for (i=l; i<len-l; i+=l) {
      product += a[i]  * b[i]   +
                 a[i+1]* b[i+1] +
                 ....
      prefetch(&(a[i+3*l]);
      prefetch(&(b[i+3*l]);
}
product += a[i]  * b[i]   +
           a[i+1]* b[i+1] +
           ....                (d)
```

The code in (d) is said to cover all loop references because each reference is preceded by a matching prefetch. Note there's an additional improvement. The examples so far have had the implicit assumption that prefetching one iteration ahead is sufficient to hide the latency of main memory accesses. However, this is not necessarily the case. When loops contain small computational bodies, it maybe necessary to initiate prefetches δ iterations before the data is referenced. Here, δ is known as the *prefetch distance* and is expressed in units of loop iterations. Mowry, et. al. represent the computation as: $\delta = \left\lceil \frac{l}{s} \right\rceil$. Where l is the average memory latency, measured in cycles, and s is the estimated cycle time of the shortest possible execution path. In (d) δ = 3.

2.1.2 Compiler tranformations

In order to avoid unnecessary prefetches, we implement a data reuse analysis phase similar to that proposed by Wolf and Lam (please refer to paper for more details, as it's not feasible to describe the specifics here). Since stream accesses are modeled as affine accesses to array elements, we attempt to identify and act upon three forms of reuse from within the induction variable framework. In the event of **spatial reuse** across loops, we only prefetch when `(i mod l = 0)`. For **temporal reuse** we prefetch only when `i = 0`, and assume the data remains the cache. For **group reuse** (where multiple references refer to the same location), we identify and only prefetch the **leading reference**. Ideally, the compiler should apply the proper degree of loop unrolling to eliminate prefetch predicate calculation, but due to implementation issues, we resorted to performing these transformations by hand. We also estimate the volume of data accessed within a given loop, in an effort to determine if outer loop iterations can benefit from temporal reuse.

Even when restricted to well-conformed looping structures, the use of explicit `prefetch` instructions exacts a performance penalty that must be considered when using software prefetching. `Prefetch` instructions add processor overhead not only because they require extra execution cycles but also because the source addresses must be calculated and stored in the processor. Ideally, this prefetch address should be retained so that it need not be recalculated for the matching `load` instruction. By allocating and retaining register space for the prefetch address, register pressure will increase, which in turn may result in additional spill code. The problem is exacerbated when the prefetch distance is greater than one, since this implies either maintaining **delta** address register to hold multiple prefetch addresses, or storing these addresses in memory if the required number of address registers are not available.

Comparing the transformed loop in part (d) to the original loop, it can be seen that software prefetching also results in significant code expansion, which in turn may degrade instruction cache performance. Also, because software prefetching uses no run-time knowledge, it is unable to detect when a prefetched block has been prematurely evicted and needs to be refetched (perhaps after a context switch, which is beyond the application's control).

In addition to manually transforming code (next section) to include prefetch, we implemented the data reuse analysis phase and prefetch insertion transformations in GCC 3.1's loop analysis framework, but were unable to automatically unroll the loops. As a result, we observe a large number of unnecessary prefetches, which has led to less accurate results. If we had more time, we could have debugged the reuse analysis phase.

2.2 Code Transforming and Simulation

The following discusses workflow using the matrix multiplication as our case study.

2.2.1 Prefetching

We manually inserted prefetch instructions to prefetch each matrix ahead of time and mixed them with the computation code. We also used our modified GCC to automatically insert prefetch according to our data flow analysis.

2.2.2 Data gathering (Impulse)

Doing matrix multiplication, the second matrix, multiplier, will be read one column at a time. One cacheline (32 bytes) contains 8 words, each of which belongs to a different column. When we read in one column of the matrix, we will also have data for the next 7 columns in the cache. However, if the cache is too small, we will have to reread these cachelines back and waste a lot of bandwidth, cycles and energy. To solve this problem, the second matrix is first transposed and the matrix multiplication problem is changed into computing inner products.

2.2.3 Strip-Mining and Matrix Multiplication

Strip-mining is a algorithmic-level technique that aims to reduce memory bandwidth requirement. Basically, the idea is to fit a “strip” of data in a fast storage such as cache, and apply as many operations as possible before fetching the next strip. When done correctly, this eliminates unnecessary spilling to the memory, thus improving overall performance. It’s also quite important for prefetching to be effective.

The application of strip-mining in large matrix multiplication has been studied extensively, and is probably more popularly referred to as “tiling” or “subblocking.” We will not give detailed descriptions of how it works here. We did implement the strip-mined version of matrix multiplication.

We further observe that most modern microprocessors have a hierarchy of caches (at least L1 and L2). So we see an opportunity to experiment with hierarchical strip-mining: we can, in theory, fetch into L1 the strip of data that’s needed for current computations, while we use L2 to buffer both the data of the current strip that cannot immediately fit into L1, and the next strip of data. So in the case of matrix multiplication, we extended our strip-mined code to further dividing the sub-matrices into even smaller matrices, adjusting the sizes so they fit into L1. And while we are doing multiplications on these matrices, we tried using the prefetch instructions to fetch the next group of matrices into L2 cache, overlapping computation with memory operations. As with other prefetch efforts, the inner loops is unrolled to avoid excessive prefetches. And due to the fact that the second matrix are traversed multiple times in the computation stage, we replicated the code in the first loop of computation, and did prefetches only in that one loop so we would not issue redundant prefetches.

3. Experiement and Results

3.1 Experiment Environment

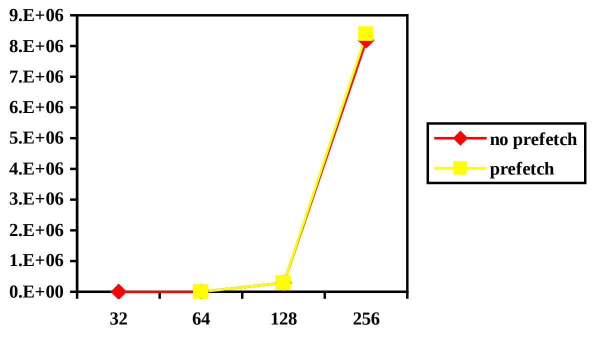
We used the RSIM simulator to simulate a legacy processor. RSIM is a user-level execution-driven simulator that models the processor pipeline and memory hierarchy in detail. The configuration that was used is illustrated in the **Table 2** below. Most of the configuration we used were the default values except processor speed, which allowed us to model the memory having a more realistic latency of 72 cycles. The 16kB and 64kB size of the L1 and L2 cache sizes are kept small to reduce the run time. From RSIM simulations, several statistics were gathered. The main data of concern was the run time from RSIM. Other important statistics included how much of the run time was actually spent doing "useful work" versus time spent waiting for functional units or waiting for a memory operation.

Table 2. RSIM Configuration

# of processor	1
Processor Speed	1.2GHz
Issue Width	4
Instruction window size	64
Memory queue size	32
Cache line size	32 bytes
L1 cache size	16KB
L1 latency	1 cycle
L2 cache size	64KB
L2 latency	5 cycles
Memory Latency (for L2 miss)	72 cycles

Besides the RSIM, we also tried to an actual hardware, UltraSparcIII, to verify the results we were observing in RSIM on a real system.

3.2 Results of Matrix Multiplication



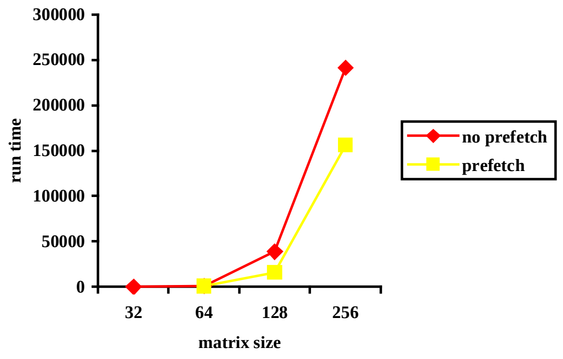
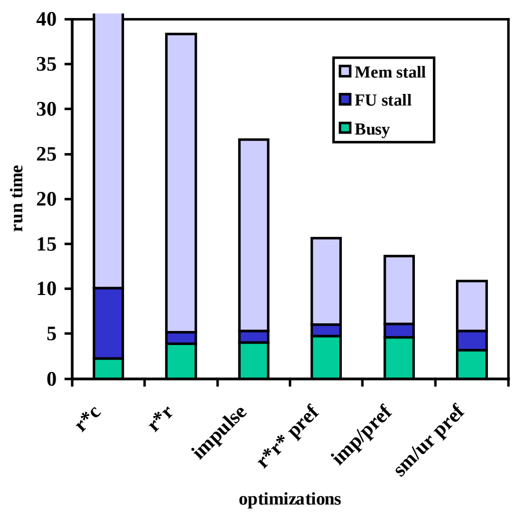


Figure 1. Matrix Multiplication Results on RSIM (a) rowXcolumn (b) rowXrow

With the modified gcc which included the prefetch option, we were able to compile our simple test case and measured the performance. With the matrix organized in a column-major format, there was clearly no benefit from inserting the prefetch as shown in **Figure1 (a)**. There was actually a loss in the performance because of the overhead involved with prefetch. However, if the data structure is more suitable for prefetch (i.e. arranged in a row-major format), the benefits of prefetching are shown in the run time (**Figure 2(b)**). Even though we see around 30% increase in performance, a good portion of the prefetches were still identified as "useless" prefetches, which means the compiler can probably be further improved. By analyzing the data further, the biggest improvement in the run time comes from the fact that smaller amount of time is spent waiting for a memory. As a result, greater proportion of the overall runtime is spent doing actual work.



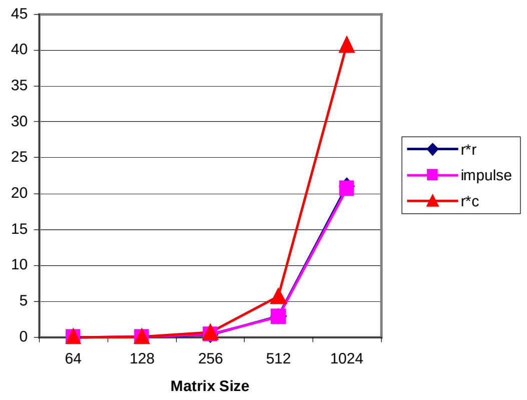


Figure 2. (a) Performance Analysis on RSIM of Matrix Multiplication (b) Matrix Multiplication performance on UltraSparcIII

However, from a given column-major matrix, the benefit of a prefetch can be obtained if we transform the data structure into a row-major data structure, which could've been done by the Impulse memory controller transparently. In this "legacy" architecture, we emulated the Impulse memory controller in software. For the test case that we were working with, the transposing of the matrix was all that was required to achieve the appropriate data manipulation.

The row-row matrix multiplication was assumed to be the maximum performance gain that we could obtain if we start with a row-column matrix multiplication. After applying the impulse transformation and including prefetch, the performance was very similiar to a row-row matrix multiplication. However, for larger data size (much greater than the cache size), the performance of this manipulation will not be better than a row-row matrix multiplication. In the smaller data size, the matrix transposing allowed us to bring in the data earlier (almost acting like a prefetch itself) and since L2 is a write-back cache, it would result in a fewer L2 cache miss. By applying other software tricks such as loop unrolling and strip mining, further enhancement in performance was observed.

Table 3. Average run time (sec) of code with different techniques applied on Elaine (UltraSparc-III+)

Version	without strip-mining		with strip-mining	
	manual[3]	gcc[4]	manual[3]	gcc[4]
rowxcol, no prefetch	30.4		11.5	11.5
rowxcol, prefetch A	30.0	29.9	11.6	
rowxcol, prefetch B	30.4		11.5	
rowxcol, prefetch A&B	30.0		11.6	
rowxrow, no Impulse, no prefetch	13.9		10.8	
rowxrow, Impulse, no prefetch	13.9		11.0	
rowxrow, Impulse, prefetch Impulse	13.9		10.9	11.0
rowxrow, Impulse, prefetch A	13.8		11.0	
rowxrow, Impulse, prefetch B	12.5		11.2	
rowxrow, Impulse, prefetch Impulse&A&B	13.0	13.3	11.6	

1. Each configuration is run several times to compensate for the variation
2. 1024x1024 matrix, manually unroll loops 8 times where appropriate
3. Prefetch A[ai] (current row of matrix a), prefetch B[bi+32] (the 4th cacheline ahead)
4. We look at the assembly code generated by gcc with our software prefetch extension to see what type of prefetches (Impulse, A, or B) are added and place the measured run-time in the corresponding row.

Table 3 shows that prefetch does help to improve performance on real machine. Several things can be noted. One is that, for row*row with Impulse without strip-mining, prefetching only matrix B gives the best result; the reason is that for each row of matrix A, the entire matrix B is “streamed” into the cache, so B gets the most benefit of prefetch. Another interesting observation is that when all Impulse, A, and B are prefetched, the performance is actually worse than just prefetch B. Our guess is that too many prefetches will conflict with each other and causes some improper eviction, a problem we further discuss in Section 4.2. The result also shows that our extended GCC can insert helpful prefetch instructions and improve the performance in most case, although it may not produce optimal results.

The strip-mined version of matrix multiplication showed substantial performance gain versus the naïve version (last bar in Figure 2(a), with prefetch). The hierarchically strip-mined version, however, did not. In fact, though it ran on real hardware, it crashed RSIM so we were not able to gather meaningful statistics on it. Some explanation for no performance improvement are suggested in section 4.2

To summarize, with all techniques applied appropriately, we saw10x speedup on RSIM, but a little tricky to show that on the actual hardware due to its large 8MB L2 cache and hence requiring a very large test data and long run time to get to the point where we can see 10x speedup.

4. Discussion

It required a considerable amount of effort to transform a normal code into a well-behave stream program. Loop tiling needed to be done carefully, taken the L1 cache size (and possibly its replacement policy) into account. In correct transformation will result in wrong computation and/or poor performance. Prefetching may seem easier, but it is difficult to verify the "correctness" of prefetching code, because it only affects the performance and not the computing result, which could be difficult to observe and reason.

4.1 Rewriting code with Stream in Mind

Generally speaking, we should have used a stricter definition of stream application and explicitly written our code as such. We could then have provided the compiler with more knowledge of the memory access and avoided "short stream effects". For example, with the matrix multiplication code written as is, GCC will treat a 32x32 matrix as 32 32-element streams, whereas we could have just used a single 1024-element stream and had fewer late prefetches.

4.2 Prefetch Problems

Given our experience of using the prefetch instruction, we see some problems with both its current implementation and its potential use to transform cache into SRF. As mentioned earlier, one problem is that it incurs too much overhead both in terms of address calculations and number of instructions. This is especially true when we just want to fetch a large block of sequential data, which really need just one prefetch instruction that specifies the starting address and length. Another problem is that, although we seem to have control on what's in the cache with prefetch, we have little control over what's evicted from the cache when it gets full. As most cache are set-associative with LRU replacement policy, data may be evicted incorrectly. For example, in the case of matrix multiplication, we would like to keep the rows of the first matrix in the cache while we stream in the second matrix, but as we do that, the first elements of the rows becomes more LRU, so they may get evicted when the columns of the second matrix fill up the cache. So LRU is not necessarily a policy we want, and we definitely can use more explicit control over the cache, such as ways to "pin" certain data in it or fetch data into a certain region of the cache (cache coloring technique?).

The solution to these problems seems to be a prefetch instruction that can fetch a large block of sequential data; it would be even more ideal if we can specify strides. This has the additional advantage not having to interleave computation instruction with prefetch instructions, making the code cleaner. Of course this makes explicit control on cache content even more critical: the large block of memory being prefetched will evict large part of the cache, which will require hints from the programmer to work well. Another solution may be to have some on-chip SRAM memories (scratch pad). For example, Intel XScale architecture allows programmers to pin down cachelines and conceptually make them become SRAM.

5. Future work

In hindsight, we should have spent less time developing such a rigorous data reuse analysis phase, and focused our efforts on cache partitioning and stream scheduling. If we can ensure data will not be evicted from the cache, we can prefetch it much farther in advance rather than just in time, as we have done. However, implementing such scheduling would require significant changes within GCC (not as simple as hacking the induction framework).

Also, since we claim to use the L1 as an SRF, it would be worth checking how much of the L1 we are able to utilize as a holding store. Ideally, we should be able to pack the entire cache with prefetched streams. And while we are on the topic of cache, it may be worthwhile to investigate how we can gain more explicit control of the cache (new instructions, etc) and otherwise to make it behave more like the SRF.

We haven't investigated the mapping of producer-consumer locality onto legacy architectures. Our benchmark was just one kernel working alone; so instead of that, we would use some more typical stream applications.   Also, we believe DLP can be mapped to SIMD/MMX/VIS instructions, which was not studied.