Patterns for Cache Optimizations on Multi-Processor Machines

Nicholas Chen, Ralph Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
{nchen, johnson}@cs.illinois.edu

ABSTRACT

Writing parallel programs that run optimally on multi-processor machines is hard. Programmers not only have to reason about *existing* cache issues that affect single-threaded programs but also *new* cache issues that impact multi-threaded programs. Such cache issues are surprisingly common; neglecting them leads to poorly performing programs that do not scale well. We illustrate these common cache issues and show how current tools aid programmers in reliably diagnosing these issues.

The goal of our work is to promote discussions on whether such cache issues should be included in the Berkeley *Our Pattern Language* — either by incorporating such performance issues directly into each pattern as *forces* or by creating a supplementary pattern language for general performance optimizations.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.3.3 [**Software Engineering**]: Language Constructs and Features—*patterns*

General Terms

Documentation, Measurement, Performance

Keywords

Caches, Optimizations, Patterns, Profiling

1. INTRODUCTION

> *The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet.* — Michael A. Jackson

"**DON'T**". That's the usual aphorism given whenever the topic of optimization comes up; focus on correctness instead and let the compiler and hardware handle making the program run faster. Usually such advice works very well but sometimes optimization is necessary.

Despite the advancements in current compilers and hardware architecture, we have not been able to (and it is unlikely that we will ever be able to) take a sequential program and have it *automatically* converted to a well-optimized parallel program. Programmers need to be able to reason about their parallel programs. They need to understand how to structure their programs, what algorithms to use, which libraries to use **and** *how to optimize their programs when performance is not as expected.*

Unfortunately, program optimization has earned the reputation as a form of *black art* reserved only for those with intricate understanding of both complicated compiler techniques and arcane hardware architecture. Might patterns be a way to elucidate the finer details of program optimization? We explore such an idea in this paper.

In this paper we attempt to answer the following questions:

1. How much does a programmer need to know about computer architecture to reason about memory optimizations? We present a simplified model of memory that emphasizes the idea of the *cache line*. We think knowledge about cache lines is sufficient to explain many of the memory issues that programmers encounter.

2. What is a useful way to illustrate performance issues? We carefully selected micro-benchmarks [1] that are easy to understand and clearly illustrate common memory problems. In particular, we focus on using existing profiling tools to diagnose problems instead of using ad-hoc methods or, worse, trying to reason about the code manually.

3. Do most programmers need to care about optimization? We demonstrate the performance gains of memory optimizations through our micro-benchmarks. But, more importantly, we motivate the idea that programmers need to understand such concepts to write effective parallel programs. We present three patterns in this paper that help solve some cache performance issues.

---

[1] We acknowledge that micro-benchmarks are not always representative of a problem. However, there is a tension here between what is *representative* and what is easy to *understand* for the reader. For this paper, we emphasize the latter.

We hope these questions and our answers will generate discussions on how to proceed with a general pattern language for program optimization.

2. BEFORE WE BEGIN

It is important to realize that performance optimizations can be very *specific* — they depend on the *exact* architecture of the machine (processor, memory, etc), the *exact* version of the compiler, the *exact* version of the operating system and the *particular* configuration of the program that we are trying to optimize. Thus, it is not surprising that optimizations for one particular scenario might not work as well for a different scenario.

To someone who is unfamiliar with program optimization, the previous sentence seems to imply that optimizing a program is an almost *arbitrary* process. However, there are well-defined steps[14] to follow for optimizing a program. These steps help programmers determine what works and what fails:

1. Create representative benchmarks that consider both the program and platform characteristics that we are targeting.
2. Measure performance on those benchmarks with monitoring tools that enable precise and fine-grain measurements on key processor events.
3. Identify potential hotspots from those measurements.
4. Optimize those hotspots by trying a particular technique.
5. Evaluate the impact of that optimization technique.
6. Repeat as necessary until desired performance is achieved.

By following each step carefully, we can clearly identify what works and what fails. There are many different things that need to be considered for each step. Knowing what to do for each stage requires both experience and ingenuity. Patterns work well for codifying that knowledge. While the results of optimizations are specific to each scenario, the patterns for optimization are general and can be used for different scenarios.

There are many different levels of a program that we can optimize: system-level, application-level and computer architecture-level. The patterns in this paper focus on computer architecture-level optimizations that typically improve performance 1.1 - 1.5x[18] in real applications. In our microbenchmarks, we noticed performance improvements up to **5.5x**.

3. A VIEW OF COMPUTER MEMORY

Most programmers have been conditioned by their programming languages to have a simplistic view of computer memory. In their minds, computer memory looks like Figure 1 where each cell in memory has an address associated with it. Programs read and write to the contents of memory by providing the appropriate address. Access to each block, in the programmers' minds, takes the same amount of time — and to them this time is negligible.

In reality, however, access to memory is non-negligible compared to processor speed. In fact, memory is abysmally slow when compared to processor speed. This disparity in speeds is known as the *memory wall*[23] and it is the reason

Figure 1: How programmers see memory

why memory access is one of the major **bottlenecks** for performance.

To alleviate this disparity in speeds, computer architecture has come to rely on *caches*. Caches are smaller and faster memory that are closer to the processor. Caches access memory in blocks called *cache lines*. Cache lines access memory not by individual cells but chunks of cells. Modern processors today have cache lines that access memory in 64 byte chunks. Moreover, processors today have *multiple* levels of cache (Level 1, 2 and even 3). Subsequent levels contain more data but have lower bandwidth and higher latencies for memory accesses.

Figure 2 shows an example of what a 4-byte cache line would look like. When a program accesses cell #2, the processor brings in cells #0, #1 and #3 as well since they reside on the same cache line. Because caches are small, they can only contain a subset of the total memory in a computer. When there is not enough space in the cache, the computer *evicts* some of the cache lines to make room for others to come in. In Figure 2, if the cache runs out of space and decides to evict cell #5, cells #4, #6 and #7 are evicted as well since they all belong to the same cache line.
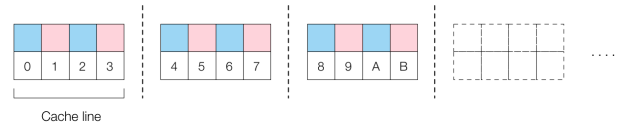

Figure 2: How processors see memory

Why do caches operate in cache line granularity? Two reasons: *spatial locality* and *temporal locality*. Spatial locality takes advantage of the fact that data close together tend to be accessed together. In Figure 2, when a program accesses cell #2, it is likely to access cells #0, #1 and #3 especially if it is operating in a loop or if its data type takes up the whole cache line. Temporal locality, on the other hand, utilizes the fact that if a program accesses cell #2 now, it is likely to access it in the future too; it is advantageous to keep it in the cache.

These two heuristics of locality usually work well to alleviate the memory bottleneck. However, there are times when programmers forget about the cache model of memory and write code that work *against* it rather than with it. We illustrate such problems in the next section and then describe patterns that optimize cache performance. As long as processors, be they single core or multi-core, continue to rely on caches to alleviate the memory wall, such patterns will remain relevant for program optimization.

Having a basic understanding of caches helps programmers understand why their parallel programs perform poorly and what they can do to improve the performance.

4. FAMILIAR PROBLEMS WITH SURPRISING CACHE ISSUES

Before discussing the patterns, we motivate the benefits of cache optimizations by showing the performance impact of cache optimizations on the following micro-benchmarks. These micro-benchmarks use familiar programs to reveal cache problems that might be less familiar to some. Knowing the symptoms of cache problems i.e. *cache smells* — named after the corresponding *code smells*[11] in program design — guides programmers on which cache optimization patterns to use. Section 5 then introduces the patterns we used to improve the performance in our micro-benchmarks.

4.1 Matrix Matrix Multiplication

```
11  // A, B = Input matrices; C = Output matrix
12  void matrix_matrix_multiplication(int A_rows,
13  int B_cols, int A_cols, double** A,
14  double** B, double** C) {
15   for (int i = 0; i < A_rows; i++)
16    for (int j = 0; j < B_cols; j++) {
17     for (int k = 0; k < A_cols; k++) {
18      C[i][j] += A[i][k] * B[k][j];
19     }
20    }
21  }
```

Figure 3: Matrix matrix multiplication

Figure 3 shows a *basic* matrix matrix multiplication function that a typical programmer would write. The function multiplies matrices A and B and stores the result in matrix C. For this experiment, A and B are square matrices of size 1024 x 1024. Running the code in Figure 3 on our machine[2] takes 15,992 ms.

Parallelizing the code with the OpenMP[4] `parallel for` construct as shown in Figure 4 reduces that time to 773 ms. Without any profiling, programmers might assume that this is the best performance that can be obtained. However, profiling under VTune reveals that the code above suffers from high cache misses. By applying the LOOP INTERCHANGE pattern, we can reduce these cache misses.

Performing LOOP INTERCHANGE on the code in Figure 3 reduces its execution time to 1,995 ms — **8x** speedup just by interchanging the loops! Figure 5 shows the executions times of each of those programs. Notice that at least 9 threads are needed for the OpenMP version to be faster than the *sequential* LOOP INTERCHANGE version. In other words, it is possible for a sequential version of a program to outperform a naively parallelized program.

Moreover, by applying **both** the OpenMP parallelizing construct and LOOP INTERCHANGE, we reduced the total time to **133** ms! This experiment demonstrates that op-

---

[2]All experiments were performed on a 24-core machine(6 cores per socket) using Intel Xeon CPU E7450 @ 2.40GHz running CentOS with Linux 2.6.18. All code were compiled using g++-4.4.1 with -O2 -g. We use -O2 instead of -O3 because the Intel VTune Performance Analyzer[18] has some problems mapping binaries compiled using O3 back to the source code. However, the code compiled with -O2 exhibits the same characteristics as the code compiled with -O3.

```
11  // A, B = Input matrices; C = Output matrix
12  void matrix_matrix_multiplication(int A_rows,
13  int B_cols, int A_cols, double** A,
14  double** B, double** C) {
15  #pragma omp parallel for
16   for (int i = 0; i < A_rows; i++)
17    for (int j = 0; j < B_cols; j++) {
18     for (int k = 0; k < A_cols; k++) {
19      C[i][j] += A[i][k] * B[k][j];
20     }
21    }
```

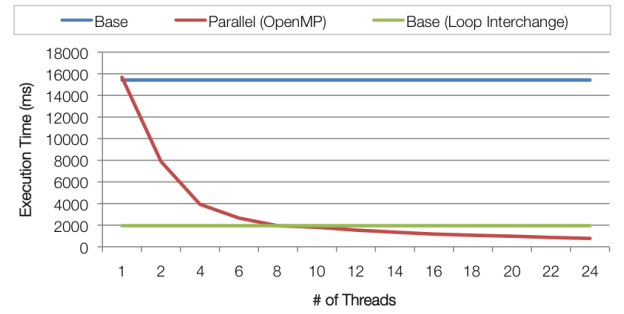**Figure 4: Matrix matrix multiplication parallelized with OpenMP**

Figure 5: Execution times of matrix matrix multiplication

timizing a sequential program for cache performance also leads to better performance in the parallelized version.

4.2 Embarrassingly Parallel Program

```
12  typedef struct {
13   double value;
14  } MyData;

24
25  MyData* data = new MyData[NUM_ELEMS];
26  double* values = new double[NUM_ELEMS];

38
39  for(int runs = 0; runs < NUM_RUNS; runs++) {
40  #pragma omp parallel for schedule(dynamic)
41   for(int i = 0; i < NUM_ELEMS; i++) {
42    for(int reps = 0; reps < ITERATIONS; reps++) {
43     data[i].value = update_operation(&data[i],
44                    &values[i]);
45    }
46   }
47  }
```

Figure 6: An embarrassingly parallel program updating the `data` array

Figure 6 shows an example of an *embarrassingly parallel* program. Lines 12-14 declare the `MyData` struct that holds a single `double` variable. Lines 25-26 initialize an array of `MyData` and an array of `double` values (to be used

later). Lines 39 - 47 run the `update_operation` in parallel on each element of the array. The `schedule(dynamic)` clause in OpenMP automatically schedules the iterations to achieve better load balancing among the different threads.

The program is embarrassingly parallel because the `update_operation` reads only the current `data` and `value` elements and writes only to the current `data` element. Such programs are easy to parallelize since they don't have any dependencies between their operations. When we parallelize it with the OpenMP, we expect to see linear speedups as the number of threads increases. Unfortunately, the program in Figure 6 fails to scale. The blue line in Figure 7 shows the speedups on our 24-core machine — the program only achieves a paltry 5x speedup even with 24 threads.
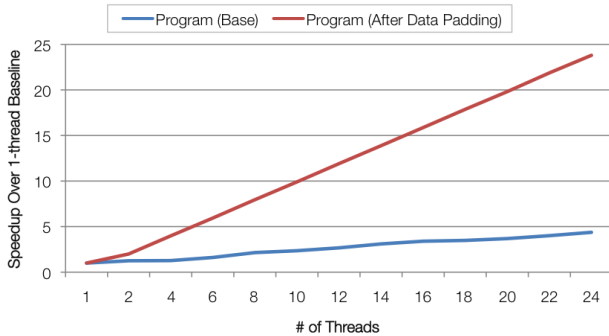

Figure 7: Speedup of program over 24 threads

The elusive culprit here is *false sharing* — the independent elements appear to be shared because they happen to share the same cache line in memory. Section 5.2 discusses false sharing in greater detail. After applying the DATA PADDING pattern, the program scales better as evident from the red line in Figure 7. The base program completes in 45,174 ms whereas the program with DATA PADDING completes in 8,192 ms. The optimized program not only scales better but also performs **5.5x** better than the original version. This experiment demonstrates that it is important to understand how caches work — without this knowledge, programmers would be left scratching their heads when their embarrassingly parallel program (the easiest kind of program to parallelize) fails to show any performance improvements!

5. PATTERNS

> *[P]remature optimization is the root of all evil.*
> — Donald E. Knuth

As a reminder, before using any of these patterns, it is important to follow the steps outlined in Section 2 for performance optimization. In particular, it is important to have a representative benchmark and a set of tools that can accurately profile the code to observe the performance before and after a pattern has been applied. Section 4 presents data on the performance impact on execution time. Here, we present data on the performance impact of each pattern as measured by the VTune profiler. Because VTune uses sampling to collect the data, the exact numbers reported might change between runs. Though those numbers change, the trends and relations between those numbers i.e. a high value vs. a low value are preserved between runs.

5.1 Loop Interchange

5.1.1 Problem

Your program uses a nested loop to access multidimensional array elements; upon profiling you discover that the loop suffers from poor cache locality i.e. it has high cache misses. What can you do to improve cache locality and improve overall performance?
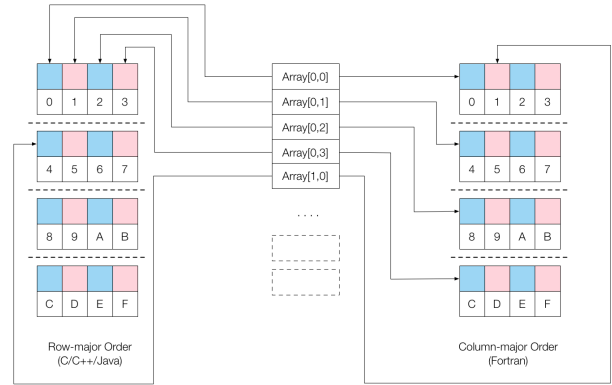
5.1.2 Context

Figure 8: How arrays are stored in memory

The way arrays are stored in memory depends on the programming language that you are using. In C/C++/C#, arrays are stored in *row-major* order; in Fortran, arrays are stored in *column-major* order. Figure 8 shows how a two-dimensional array is stored in memory for both configurations. In particular, pay attention to how the layout fits with the concept of the cache line introduced in Section 3. To work *with* the cache, the way our program accesses the array elements has to take advantage of spatial locality — once a cache line is in the cache, we need to take full advantage of it.

In Figure 3, our C program fails to take advantage of spatial locality. The loop brings in an entire cache line but only uses one element of it. Notice how the program accesses the the B array i.e. `B[k][j]` on line 17. The `k` counter changes in the inner loop; on each loop iteration we are bringing in an entire cache line but only using a single element from it!

5.1.3 Forces

In Figure 3, the program only accesses arrays A and B in a similar order. In some cases, there might be arrays that you are accessing in different orders. You then have to made a tradeoff between which array accesses to prioritize for cache performance. For instance, in Figure 9, there is no immediate answer on which array to prioritize; without a proper benchmark to profile the code it is impossible determine which array access has a bigger performance impact.

The loops in Figure 3 are also simple: there are no loop-carried dependencies between the loop iterations. When they are loop-carried dependencies, the analysis becomes more complicated and it is no longer just an issue of cache performance but also *program correctness*. Handling loop-carried dependencies is beyond the scope of this paper but this issue has been touched upon by various compiler books[15][16].

Version	Code	Cache Misses (`L2_LINES_IN.SELF.ANY`)
Before	**for** (**int** j = 0; j < B_cols; j++) **for** (**int** k = 0; k < A_cols; k++) C[i][j] += A[i][k] * B[k][j];	375,488
After	**for** (**int** k = 0; k < A_cols; k++) **for** (**int** j = 0; j < B_cols; j++) C[i][j] += A[i][k] * B[k][j];	**34,166**

Table 1: Cache miss events for Figure 3 under VTune

```
1  for(int i=0; i < I_INDEX; i++)
2    for(int j=0; j < J_INDEX; j++)
3      A[i] = A[i] + B[j,i] + C[i];
```

Figure 9: A nested loop accessing multiple arrays

```
11 // A, B = Input matrices; C = Output matrix
12 void matrix_matrix_multiplication(int A_rows,
13 int B_cols, int A_cols, double** A, double** B,
14 double** C) {
15   for (int i = 0; i < A_rows; i++)
16     for (int k = 0; k < A_cols; k++) {
17       for (int j = 0; j < B_cols; j++) {
18         C[i][j] += A[i][k] * B[k][j];
19       }
20     }
21 }
```

Figure 10: Matrix matrix multiplication with loop interchange

5.1.4 Solution

Exchange the orders of the nested loops to help improve cache locality. As a rule of thumb, in C, you would want the innermost loop to vary over the last dimension of your array i.e. if you have `Array[i][j]` then the outermost loop should iterate over i and your innermost loop should iterate over j. Figure 10 shows how we optimize the matrix matrix multiplication example using LOOP INTERCHANGE.

Table 1 shows the cache miss rate for the loops as profiled under VTune. Notice that after performing LOOP INTERCHANGE, the number of cache misses has been significantly reduced. This reduction accounts for the performance boost described in Section 4.

LOOP INTERCHANGE not only improves cache locality but also makes the code more amenable for vectorization. Optimizing compilers can usually auto-vectorize the code once the it is in a *suitable* form.

5.1.5 Known Uses

LOOP INTERCHANGE is used together with other loop transformations in BLAS (Basic Linear Algebra Subprograms)[1]. Some modern optimizing compilers like the Intel C/C++ Compiler[2] are able to detect certain opportunities for LOOP INTERCHANGE automatically. However, such detection is far from perfect and the programmer still has to perform the transformation by hand in most cases.

5.2 Data Padding

5.2.1 Problem

You have a data structure that does not fit nicely into the cache line of the machine. When you parallelize your code, your performance suffers because of *false sharing*. How do you transform your code to reduce false sharing and improve overall performance?

5.2.2 Context

On a shared-memory multi-processor machine, each processor may load the same cache line into its own local cache. To maintain *cache coherency*, whenever one processor modifies the value in its copy of the cache, it has to notify the other processors to *invalidate* their copy since the value in their cache lines is now outdated. As the number of processors increase, this notification mechanism becomes more costly since each write to the local cache on processor needs to notify every other processor that shares that cache line.

Worse, since cache lines are usually 8 - 512 bytes wide, multiple data elements might fit into a single cache line. When a processor modifies just a single byte in the cache line, the entire cache line has to be invalidated even though the other bytes in the cache line have not been modified. This phenomenon is known as *false sharing*; the way the cache line works gives the illusion that all the data in it are being shared by other processors even though it is usually just a few bytes from the entire line. False sharing is a repercussion of the way hardware works.

Figure 11 shows what happens in Figure 6. The processors are trying to access different memory cells. However, because those cells reside on the same cache line, only one processor can modify that cache line at a time.
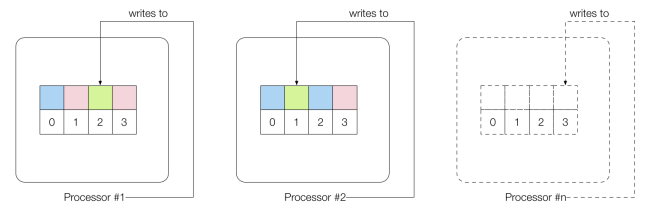

Figure 11: Processors accessing different memory cells on the same cache line

5.2.3 Forces

The main issue is the size of the cache line. If the machine has a small cache line then your data structure is likely to

Version	Code	Cache Misses (`L2_LINES_IN.SELF.ANY`)	Modified Data Sharing (`EXT_SNOOP.ALL_AGENTS.HITM`)
Before	**typedef struct {** MyData∗ data = new MyData[NUM_ELEMS]; } MyData;	4,672	4,932
After	**typedef struct {** **double** value; } __attribute__ ((__aligned__ (CACHE_BOUNDARY))) MyData;	**124**	**48**

Table 2: Cache misses and data sharing events for figure 6 under VTune

occupy the entire line. Since no other data can occupy that line, you have effectively avoided the problem of false sharing. However, the size of the cache line varies from one machine to another and has to be taken into account when implementing a solution.

On the other hand, there is the problem of generality and portability. By optimizing for a particular machine's cache line, your solution might not work as well for other machines.

A solution that reduces false sharing should also maintain the original memory consumption of the program whenever possible. There is the tradeoff here between execution time and the amount of memory necessary. Reducing false sharing at the expense of extra memory might work but could introduce other memory bottlenecks if the program utilizes more memory bandwidth.

5.2.4 Solution

Pad your data structure with extraneous "padding" data. Padding data do not contribute to the functionality of your program; they only expand the size of your data structure so that it fits into an entire cache line. In effect, this grants your data exclusivity to the entire cache line.

Figure 12 shows what happens after adding data padding. The blue element now occupies the entire cache line; and the pink element occupies the other cache line. Our program only writes to the dark blue and dark red cells; the other cells are just padding data.
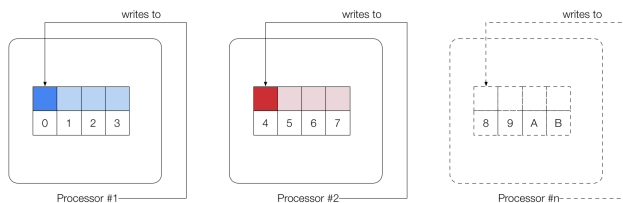

Figure 12: Padded data structure uses an entire cache line

You can pad data manually by creating a byte array that is initialized to the number of bytes to pad i.e. `byte pad[NUM_OF_BYTES_TO_PAD]`. You can also use compiler attributes to pad the data structure automatically for you. Figure 13 shows how to do this in GCC. On line 20, we use the `__attribute__` specifier with the `__align__` attribute. We use the value of `CACHE_BOUNDARY` which is defined on lines 12 - 16. We compile our program with `gcc -O2 -fopenmp -DLI_CACHELINE=$(getconf LEVEL1_DCACHE_LINESIZE)`

```
12 #if defined L1_CACHELINE
13     #define CACHE_BOUNDARY L1_CACHELINE
14 #else
15     #define CACHE_BOUNDARY 64
16 #endif
17
18 typedef struct {
19     double value;
20 } __attribute__((__aligned__(CACHE_BOUNDARY)))
21   MyData;
```

Figure 13: Data Padding with GCC

`getconf` lets us query the cache line size of the machine dynamically before compilation so we can tune our program for that particular machine. If the size of the cache line cannot be determined, the program falls back on the default size of 64 which is what most machines have today.

Table 2 shows the cache misses and modified data sharing for the program as profiled under VTune. The cache misses measure the number of times our program brings in a new cache line from memory — every time a cache line is invalidated, the processor needs to bring in an updated version. Modified data sharing measures the amount of internal *snooping* communication that goes on between different processors for maintaining cache coherence. It is a measure of how much data sharing occurs between different processors. After performing DATA PADDING, the number of cache misses and modified data sharing were significantly reduced. This reduction accounts for the performance boost described in Section 4.

DATA PADDING reduces false sharing at the expense of an increase in memory consumption. In some cases, this is an acceptable tradeoff for the programmer.

5.2.5 Known Uses

DATA PADDING is also applicable for languages such as Java and .NET that run on a virtual machine. The Barnes-Hut implementation in Deterministic Parallel Java[6] used padding to reduce the effects of false sharing; the Successive-Over-Relaxation implementation in the Java Grande Parallel Benchmark Suite[19] uses it as well.

The problem of false sharing occurs frequently enough that Intel's Threading Building Block[3] includes a `cache_aligned_allocator` that programmers can elect to use if they believe that false sharing is occurring in their

Version	Code	Splits Loads (L1D_SPLIT.LOADS)	Split Stores (L1D_SPLIT.STORES)
Before	MyData* data = new MyData[NUM_ELEMS];	0	0
After	**void** *data_v; posix_memalign(&data_v, __alignof__(MyData), NUM_ELEMS * **sizeof**(MyData)); MyData* data = (MyData*)data_v;	**0**	**0**

Table 3: Split load and store events for Figure 15 under VTune

code. However, the TBB guidelines, also advocate profiling the code to test for false sharing before using the `cache_aligned_allocator` since using it may increase memory usage for many small objects.

5.3 Data Alignment

5.3.1 Problem

You have a data structure that *could* fit within a cache line but has been allocated such that it crosses the boundary of a cache line. Every time this data structure is read, its remaining chunk has to be fetched from the next cache line incurring an additional overhead. How do you transform your code to reduce this performance impact?

5.3.2 Context

Figure 14 illustrates the problem of a 4-byte data structure that spans two cache lines. We say that this data structure is *misaligned*[3]: 3 bytes of the data structure is on the first cache line; the other byte is on the next cache line. When our program reads this data, the processor has to read from both cache lines, shift the data to the proper locations and combine them. These three additional steps incur an overhead; reading from an additional cache line being the most expensive.
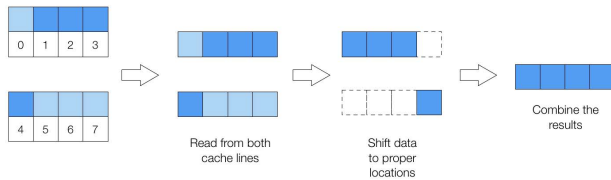

Figure 14: Misaligned memory access penalty

The problem of misalignment is also more general than spanning a cache line. All primitive data types (`int`, `char`, `float`, `double`) have a *natural* alignment within memory. For instance, on a 32-bit x86 Linux system a `double` needs to be aligned on a 4-byte boundary. On a 64-bit x86_64 Linux system, a `double` needs to be aligned on a 8-byte boundary. If the `double` is not aligned properly, our program incurs

---

[3]Readers familiar with SIMD programming would realize that data alignment is also essential for vectorization to work. Since we are concerned with cache optimizations in this paper, we will not be discussing the details of SIMD here.

some overhead as well. Fortunately, compilers are smart enough to handle such cases for us. However, compilers are not always able to determine the best alignment for user-defined data structures.

5.3.3 Forces

As with DATA PADDING there is the tension between the generality and the portability of a solution. By optimizing for a particular machine's cache line, your solution might not work as well for other machines. This is a tradeoff that you have to make.

Similarly, a solution that reduces the penalties for misaligned data access should also maintain the original memory consumption of the program whenever possible. There is the tradeoff here between execution time and the amount of memory necessary.

5.3.4 Solution

Align your data structure so that it starts on a cache line boundary. To do so you need to use an *attribute specifier* on your data structure to tell to the compiler how you want it to align your data. For GCC, you would use the `__attribute__` specifier with the `__align__` attribute that we have seen for DATA PADDING. Similar to DATA PADDING, you should set the alignment to the size of the cache line. The fringe benefit of having your data structures aligned at cache lines boundaries is that it automatically avoid problems with false sharing.

By using the `__align__` attribute on your data structure, all data of that types allocated globally, on the stack or in arrays will be aligned properly. However, dynamically allocated objects will still not be properly aligned. For that, you need to use a alignment-aware allocator. In Linux, there is `posix_memalign`. Figure 15 shows how to use it for our embarrassingly parallel example.

Our original embarrassingly parallel example used `My-Data* data = new MyData[NUM_ELEMS]`. The `new` operator only allocates memory on 16 byte boundaries. For our 64 byte cache line, this could align our data structure on four (=64/16) different boundaries in a cache line as shown in Figure 16. Three of the four possibilities lead to misaligned accesses. To avoid misaligned cache accesses, we convert the code to use `posix_memalign` instead of `new`.

Table 3 shows the split loads and stores for the cache for our embarrassingly parallel program. A split load occurs when loading data that spans two cache lines; a split store occurs when writing data that spans two cache lines.

Even though the values are zero in both cases, we intentionally show this table for two main reasons:

```
17
18  typedef struct {
19    double value;
20  } __attribute__((__aligned__(CACHE_BOUNDARY)))
21  MyData;
30  int main(int argc, char* argv[]) {
31
32    void *data_v, *values_v;
33    posix_memalign(&data_v, __alignof__(MyData),
34      NUM_ELEMS * sizeof(MyData));
```

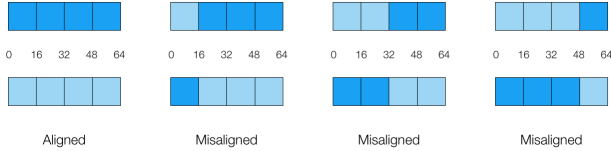**Figure 15: Data Alignment with `posix_memalign`**

Figure 16: Possible configurations of 16 byte alignment on a 64 byte cache line

1. It was a mistake to assume that performing DATA ALIGNMENT would improve performance significantly. Misaligned accesses were not the main bottleneck of the program in the first place; it was false sharing that we fixed using DATA PADDING.

2. The values of `L1D_SPLIT.LOADS` and `L1D_SPLIT.STORES` were not exactly zero for every part of the program. There were non-zero values for other modules that our program used such as *libc* and *vmlinux*. However, since we have no control over those modules there is nothing that we can do to lower the split loads and stores. Technically, our program should also contribute to the split loads and stores count but because VTune uses statistical sampling, the impact from our program was minimal compared to other modules.

Both versions of the code take the same time to run i.e. 8,200 ms. So DATA ALIGNMENT was not a necessary pattern for this program. It might be a suitable pattern for other programs such as image processing and video encoding that require reading a lot of misaligned data. Again it is important to profile the code with a representative benchmark before attempting any optimizations.

Figure 17 shows a different micro-benchmark to calculate the performance effects of split loads and stores. The program allocates a contiguous buffer (i.e. `void* data`) of a certain size (i.e.`uint32_t size`). The function `process` then reads a `uint64_t` value from the buffer, negates it and writes it back to memory. It then proceeds to the next `uint64_t` value until it reaches the end of the buffer.

Table 4 shows what happens when we run this micro-benchmark with aligned and unaligned buffers. In the *aligned* version, we used malloc to allocate a buffer. By default, `malloc` allocates memory that is 8-byte aligned for its return type i.e. a `void*` pointer. When we profiled the program under VTune, there were very few split loads and store events. In the *unaligned* version, we used malloc to allocate

```
16  void process(void *data, uint32_t size) {
17    uint64_t *begin = (uint64_t*) data;
18    // Divide by sizeof(uint64_t) == 8
19    uint64_t *end = begin + (size >> 3);
20
21    while(begin != end) {
22      *begin = -(*begin);
23      begin++;
24    }
25  }
```

Figure 17: Data Alignment micro-benchmark to show effects of split loads and stores

the buffer, but we intentionally shift the buffer alignment by 1 byte. When we profiled this version of the program, we noticed an increase in the number of split load and store events.

Because this is a micro-benchmark that runs for less than a second, it is infeasible to compare the *wall clock* running time of both versions. Instead, we read the time stamp counter (see page 355 of [5]) directly from the processor. In the *aligned* version, the `process` function took **483630** ticks; in the *unaligned* version, it took **2533174** ticks. That is about a **5x** difference.

5.3.5 Known Uses

DATA ALIGNMENT has been used in various high-performance libraries such as ATLAS[22] and FFTW[12]. To ensure high performance, these libraries require programmers to use their own internal version of the memory allocator. Intel's Threading Building Block also provides its own version of a `cache_aligned_allocator`.

6. RELATED WORK

Catanzaro's work at ParaPLoP '09[7] focuses on the importance of structuring data structures so that they are more amenable for parallel execution. Our work builds upon his by extracting smaller, well-defined patterns that programmers can apply to improve the performance on their code. In addition, we illustrate the importance of such patterns with actual programs to convince programmers who might be skeptical of the performance gains from applying such patterns.

Tasharofi's work at ParaPLoP '10[20] focuses on patterns for loop optimization. Her PARTITIONED LOOP and TILED LOOP patterns are for optimizing loops but can also be regarded as forms of cache optimizations. Both those patterns take advantage of spatial and temporal locality in the caches to improve the performance of loops. Her work complements and augments the patterns that we have introduced here.

Chilimbi and Larus's work discusses the importance of cache-conscious structure definition[8] and layout[9]. They identified several transformations that could be performed on user-defined data types to improve their performance. They also contributed several semi-automated tools that assist programmers with the transformation process. Their transformations go beyond the patterns discussed here; they suggest high-level transformations such as *structure splitting* and *field reordering*. As future work, we would incorporate their transformations into our catalog of cache optimization

Version	Code	Splits Loads (`L1D_SPLIT.LOADS`)	Split Stores (`L1D_SPLIT.STORES`)
Aligned	**void** ∗data = malloc(SIZE ∗ **sizeof**(**char**));	4	1
Unaligned	**void** ∗data = malloc(SIZE ∗ **sizeof**(**char**)); **char**∗ misaligned_data = (**char**∗)data; misaligned_data = misaligned_data + 1; data = (**void**∗)misaligned_data;	**1028**	**629**

Table 4: Split load and store events for Figure 17 under VTune

patterns.

Frigo et al. proposed the idea of *cache-oblivious algorithms*[13] which take advantage of the CPU cache without having the size of the cache, length of the cache-line, etc as an explicit parameter. Cache-oblivious algorithms typically work using a recursive divide-and-conquer approach with each successive recursive step bringing the necessary data into the memory hierarchy. Regardless of the memory hierarchy and cache levels of the target machine, eventually the recursive step reaches a small enough subproblem where the data fits into cache comfortably. Such algorithms exist for many different problems including sorting[17], matrix transposition[17] and Fast-Fourier Transform[12]. Yotov et al. [24] conducted some experiments to compare the performance of cache-oblivious and cache-conscious programs and found that while cache-oblivious algorithms achieve *good* performance, they do not come close to the *peak* performance of similar cache-conscious algorithms. Nonetheless, these algorithms are worth documenting as alternative patterns for memory optimizations.

Intel provides optimization manuals[5] to help programmers write efficient code for their platforms. However, our work with patterns is different from an optimization manual. Optimization manuals serve as guides to compiler writers and library/framework creators who are targeting a particular platform to make the most of the underlying hardware features. Our patterns focus on design decisions that the programmer has to consciously make; issues that the compiler cannot decide for us.

Various auto-tuning research projects have attempted to alleviate some of these optimization designs from the programmer. ATLAS and OSKI[21] are two examples of auto-tuned libraries for linear algebra. We agree that whenever possible, any domain-specific optimization should be done through the compiler or through optimized libraries. However, there will **always** be new domains where specialized libraries do not exist. In those cases, the programmer has to be able to reason about what to optimize.

Dig's work on interactive refactorings for parallelism[10] shows how to transform existing sequential code into parallel code. *Concurrencer*, allows the programmer to select a recursive function and convert it to use the `ForkJoinTask` framework in Java. *Relooper* allows a programmer to select a loop and convert it to use the `ParallelArray` framework in Java. The tool is interactive: if a particular transformation cannot be applied, the tool lets the programmer know *why* and gives the programmer the opportunity to restructure parts of the code to meet the requirements of the transformation.

We see the potential of expressing some of these patterns are interactive refactorings for performance. By relating our patterns with actual results from a profiler such as VTune, we are able to notify the programmer on cache bottlenecks in their code and assist them in transforming their code.

7. CONCLUSION AND FUTURE WORK

Parallel programming is about performance. Getting programmers to think in parallel with the Berkeley *Our Pattern Language* is the first step *toward* writing effective parallel programs. But before programmers can actually write effective parallel programs, they need to have a basic understanding of how the architecture works underneath. This knowledge is important: it allows the programmer to write code that works *with* the machine and not *against* it. Once programmers have this basic understanding of how machines work, it is easier for them to reason about performance.

We have presented a small selection of patterns that demonstrate the importance of cache optimizations. More importantly, we showed the importance of a well-defined process for optimization that relies on profiling tools that provide better insight into the performance impact of such problems. Our pattern catalog is far from complete; Section 6 discusses several other cache optimization techniques that have been used in different settings. There are many other such optimization patterns to be discovered and we hope that the patterns community would consider contributing to the effort of documenting them.

8. ACKNOWLEDGEMENTS

9. REFERENCES

[1] BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/.
[2] Intel Compilers. http://software.intel.com/en-us/intel-compilers/.
[3] Intel Threading Building Blocks. http://www.threadingbuildingblocks.org/.
[4] The OpenMP API Specification for Parallel Programming. http://openmp.org/wp/.
[5] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Press, 2009.

[6] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. *SIGPLAN Not.*, 44(10):97–116, 2009.

[7] B. Catanzaro. Parallel Execution Aware Data Structures. In *1st Annual Conference on Parallel Programming Patterns (ParaPLoP)*, 2009.

[8] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious Structure Definition. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 13–24, New York, NY, USA, 1999. ACM.

[9] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious Structure Layout. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 1999. ACM.

[10] D. Dig, J. Marrero, and M. D. Ernst. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 397–407, Washington, DC, USA, 2009. IEEE Computer Society.

[11] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[12] M. Frigo and S. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, Feb. 2005.

[13] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious Algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297, 1999.

[14] R. Gerber, A. Bik, K. Smith, and X. Tian. *The Software Optimization Cookbook.* Intel Press, 2nd edition, 2005.

[15] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[16] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[17] H. Prokop. Cache-Oblivious Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

[18] J. Reindeers. *Vtune Performance Analyzer Essentials.* Intel Press, 2005.

[19] L. A. Smith, J. M. Bull, and J. Obdržálek. A Parallel Java Grande Benchmark Suite. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 8–8, New York, NY, USA, 2001. ACM.

[20] S. Tasharofi and R. Johnson. Patterns of Optimized Loops. In *2nd Annual Conference on Parallel Programming Patterns (ParaPLoP)*, 2010.

[21] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI:A Library of Automatically Tuned Sparse Matrix Kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[22] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3 – 35, 2001.

[23] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[24] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. An Experimental Comparison of Cache-Oblivious and Cache-Conscious Programs. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 93–104, New York, NY, USA, 2007. ACM.