Optimization Study for Multicores

Muneeb Anwar Khan
Abstract

Optimization Study for Multicores

Muneeb Anwar Khan

IT giants like Intel and AMD have set the stage for extensive use of Multicore processors in IT business and research, thereby providing much more computing power to run software for both the scientific and commercial use.

One of the problems with writing software in traditional fashion is that Software engineers and even top architects may not take into consideration architecture-based or resource-aware optimizations, simply relying on plain process based parallelism to improve performance of the target application for Multi-Core processors. True optimal performance for any given parallelized application can be achieved only when it has been carefully written to utilize the available resources with a deeper understanding of the underlying available hardware resources and processor architecture e.g. memory bandwidth, cache architecture etc.

The purpose of this study is to demonstrate performance optimization techniques for applications from various fields, parallelization techniques for these applications and the maximal performance achieved for parallelized applications over multicore processors through architecture and resource aware optimizations. We show how the simplest of code optimizations help improve application performance multiple times, speeding up the applications by a factor of more than two.
To my parents, brother and friends
Introduction .......................................................................................................................... 1
1.1 Background .................................................................................................................... 2
1.2 Caches ............................................................................................................................ 3
1.3 Data Locality Principles of Caches .............................................................................. 4
    1.3.1 Spatial Locality ..................................................................................................... 4
    1.3.2 Temporal Locality ............................................................................................... 4
1.4 Hardware Prefetching ................................................................................................. 4
1.5 Caches in Multicore Architectures ............................................................................. 5
    1.5.1 Cache Coherence ............................................................................................... 5
1.6 Cache Misses ................................................................................................................. 6
1.7 Memory Bandwidth & the Memory Wall ...................................................................... 7
Concepts in Performance Analysis ...................................................................................... 9
2.1 Metrics for Analyzing Optimization ......................................................................... 10
    2.1.1 Fetch Ratio ....................................................................................................... 10
    2.1.2 Miss Ratio ....................................................................................................... 10
    2.1.3 Fetch Utilization .............................................................................................. 11
    2.1.4 Writeback Utilization ....................................................................................... 11
    2.1.5 Hardware Prefetch Probability ......................................................................... 11
    2.1.6 Percentage of Fetches ...................................................................................... 11
    2.1.7 Percentage of Misses ...................................................................................... 12
2.2 Metrics for Analyzing Performance ............................................................................. 12
    2.2.1 Application Speedup ....................................................................................... 12
    2.2.2 Optimized code region speedup ....................................................................... 12
    2.2.3 Relative Throughput ....................................................................................... 12
    2.2.4 Parallel Speedup .............................................................................................. 13
Acumem Tools & Selected Applications ............................................................................ 14
3.1 Acumem Tools for Optimization Analysis ............................................................... 15
    3.1.1 Methodology .................................................................................................... 15
    3.1.2 Acumem SlowSpotter ...................................................................................... 15
    3.1.3 Acumem ThreadSpotter ................................................................................... 16
3.2 Applications for Case Studies .................................................................................... 16
Generalizing Techniques Used for Optimization .............................................................. 18
Case Study 1 – LBM ........................................................................................................... 27
Case Study 2 – MCF ......................................................................................................... 49
Case Study 3 – CIGAR ..................................................................................................... 69
Case Study 4 – LIBQUANTUM ....................................................................................... 91
Chapter no. 1

Introduction
1.1 Background

After exhausting opportunities to find more techniques for exploiting ILP (Instruction level parallelism), the world moved towards time-slice multithreading (or superthreading) combining ILP with thread-level parallelism in an attempt to keep the processor pipelines running. Having reached its limit on uni-processors the trend steered towards having multiple processing elements on a single chip, called multicores. Commercial efforts with regards to multicores have reached up to six cores on a single die with products like AMD Istanbul. IT giants like Intel, AMD and IBM have now set the stage for extensive use of homogeneous and heterogeneous multicore processors in IT business and research, thereby providing much more computing power to run software for both the scientific and commercial use.

IT hardware world has moved long towards parallel processing in terms of readily available multicore platforms being used everywhere in the industry down right to home users desktop. However, the software part of the IT world has not been particularly good at catching up with the growing trend of parallel processing. Software research up to now has not been able to mature software development tools and platforms making parallelism feasible in the process of software development. To this day, software development efforts and research do not consider parallelism to be incorporated naturally into softwares. Perhaps, parallelism is still considered more to be a part of performance improvement and optimization. Without parallelism in place, the true power of multicores can not be unleashed, though multicore hardware platforms can naturally provide greater resources to multi-tasking operating systems.

Having more processing power on the hardware end does not mean that it has the capability to run software much faster, since it is usually the “memory wall” that keeps softwares from reaching peak performance, and no processor can help improving software performance in case of data stalls caused by memory. Performance problems caused by the memory wall may be due to

- Inefficient data layout in memory
- Inefficient data access patterns
- Low utilization of fetched data
- Immature cache eviction of data and losing data reuse opportunities
- Prefetching problems

Improving performance for a given application in face of the “memory wall” may include having to remove any of the above mentioned “memory wall” problems. In the process of code optimization, parallelism is actually the next step to performance improvement measures and not the first. In the following study of optimization for multicore architectures we demonstrate Sanjiv Shah's assertion of parallelizing the application only after having dealt with the memory wall problems [17]. Not doing so, will cause
parallelism not to be very productive, although parallelism alone may show significant performance improvements over the serial application.

Optimizing applications for dealing with memory wall problems is nothing new, since application developers have been doing that with help of profiling applications that gather information from hardware counters. Use of dynamic profiling tools like the open source GPROF and PROF are common in code optimization efforts. However, in this study we use Acumem SlowSpotter and ThreadSpotter tools to look at individual application statistics and come up with optimization techniques that are independent of the underlying architecture.

A delicate point in understanding application optimization is that, performance improvement mostly revolves around “good use” of cache memory and “limited use” of shared resources like available Memory bandwidth. Proper use of cache memories is what most positively impacts the application performance. Even parallelism cannot deliver required performance improvements if a parallel application's threads are unable to exploit the data within their processors' cache properly. This point will become very clear as we go through the rest of the report.

1.2 Caches

Modern processors can run at clock speeds of several GHz and are able to execute several instructions per clock cycle. This means that a processor may have a peak execution speed of several instructions per nanosecond. For example, a 3 GHz processor capable of executing 3 instructions per cycle has a peak execution speed of 9 instructions per ns.

Modern RAM memories on the other hand are quite slow. An access to RAM memory takes 50 ns or more, causing the processor to stall waiting for the data to arrive. This makes RAM accesses one of the slowest operations a processor can perform. For example, a processor capable of executing 9 instructions per ns could have executed up to 450 instructions in the time it takes to perform a single RAM access with a latency of 50 ns.

The time it takes to load the data from memory is called the latency of the memory operation. It is usually measured in processor clock cycles or ns.

Since memory accesses are very common in programs and can account for more than 25% of the instructions, memory access latencies would have a devastating impact on processor execution speed if they could not be avoided in some way.

To solve this problem computer designers have introduced cache memories, which are small, but extremely fast, memories between the processor and the slow main memory. Frequently used data is automatically copied to the cache memories. This allows well written programs to make most of their memory accesses to the fast cache memory and only rarely make accesses to the slow main memory.
Often a computer does not just use a single cache, but a hierarchy of caches of increasing size and decreasing speed. For example, it may have a 64 kilobyte cache with a latency of 3 cycles for the most frequently accessed data, and a 1 megabyte cache with a latency of 15 cycles for less frequently accessed data. The caches in such a configuration are called the first level cache (the 64 kilobyte cache) and the second level cache (the 1 megabyte cache), or shorter the L1 and L2 caches. Some computers may also have an additional third cache level, the L3 cache. [Excerpt from “Motivation for Caches”: Acumem Freja Manual]

1.3 Data Locality Principles of Caches

Modern day processor caches work on principle of data locality, which is based on the assumption that the same or “similar” data will be accessed soon after. There are two types of data locality principles, described as follows

1.3.1 Spatial Locality

The *Spatial locality* principle assumes that data close or contiguous to the data currently being accessed will be used soon in future. For this reason, whenever required data is accessed from the main memory, a whole chunk of data is brought into the cache, making the data around the required location available for use in near future. The size of the chunk of data (containing the required data) pulled into the cache depends upon the cache line size and may vary from vendor to vendor and different processor models. Common cache line sizes are 32, 64 and 128 bytes.

1.3.2 Temporal Locality

The *Temporal locality* principle of caches assumes that data currently being accessed will be accessed yet again in the near future. For the same reason, once data is accessed, it remains in the cache for access again unless evicted by another entry due to capacity or conflict reasons. Caches commonly use the LRU (Least Recently Used) mechanism for managing eviction of cache entries in an attempt to remove the candidate entry that has been unused the longest. Some processors use victim caches to avoid the newly evicted cache lines to be dumped back into the main memory, and instead keep them in the victim cache until they are evicted from there too.

1.4 Hardware Prefetching

Modern processors have a built-in capability to analyze and detect simple data access patterns in the main memory, and pull data that is expected to be used by upcoming instructions into the cache, before these instructions are executed or even issued in the CPU pipeline. However, in an attempt to keep things fast and simple, such hardware prefetch utilities are not capable of detecting logically growing or shrinking strides in data access patterns. Therefore irregular data accesses in application code don't have good prefetch probabilities.
1.5 Caches in Multicore Architectures

For multicores, the cache hierarchy is much more complex compared with a uniprocessor. Often processors with multiple cores have partially or fully shared L2 and L3 caches with private L1 data caches. The details of the cache architecture may vary from vendor to vendor based on their business strategies and research interests. Nowadays, we have modern processors, like AMD Istanbul and Intel i7, with each core having private L1 and L2 caches with a larger shared L3 cache. There are also other flavors of multicore cache architectures, like the Intel E5345, with private L1 caches, and L2 caches shared by two cores on a QuadCore processor.

Along with different cache hierarchical architectures multicore platforms implement a cache coherence protocol to keep the data, in the caches of different cores, consistent. MESI cache coherence protocol is the one commonly used.

1.5.1 Cache Coherence

Multicores and other shared memory systems need to assure that all cores have a common view of the memory. This implies that caches need to be kept synchronized to some extent. The task of maintaining a common view of memory can be divided into to two separate sub-problems, memory coherence and memory consistency.

An informal description of memory coherence is that a read should return the data from the most recent write to the same location. There are however practical issues that prevent writes from being seen by all processors instantly when the write is performed. The point in time when a write is seen by other processors is defined by the consistency model.

Memory coherence is upheld using a memory coherence protocol. There exist several different protocols, the following discussion assumes a MESI protocol. For simplicity we will assume that the hardware has a single cache level and a shared bus where all memory accesses from all cores can be snooped.

A cache controller implementing the MESI protocol implements a finite state machine that is driven by requests seen on the shared bus and the requests from the local processor cores. The MESI protocol is named after the four states in this machine, Modified, Exclusive, Shared and Invalid.

The cache line states in the MESI protocol are described as follows

- **Modified**: The cache line is modified and is guaranteed to only reside in this cache. The copy in main memory is not up to date and must be updated at some later point.
- **Exclusive**: This cache contains an unmodified version of the cache line, no other cache contains a copy of the line.
- **Shared**: This cache and possible other caches contain an unmodified version of the cache line. The main memory is up to date with the cached copies.
- **Invalid**: This cache does not contain a copy of the cache line. Other caches may contain a copy.

Whenever a processor reads a line that is not available in its cache, the line enters either the exclusive or shared state depending on whether other caches hold the line at the time of the read. Other caches holding the cache line in the modified or exclusive state are downgraded to the shared state. The downgrade will force modified data to be written back to memory.

A write causes the line to enter the modified state. Writing to a line in the shared state causes invalidations to be sent on the bus to invalidate other copies of the line. Writing to a line in the exclusive state does not cause any invalidation to be sent since the local cache is known to have the only cached copy of the line.

A write to a line in the invalid state forces all other caches to invalidate the line and, somewhat unintuitively, causes the cache line to be read into the local cache. Hence, a write will normally require twice the bandwidth of a read. [Acumem Freja Manual]

### 1.6 Cache Misses

A cache miss refers to a failed attempt to read or write a piece of data in the cache, which results in a main memory access with much longer latency. There are three kinds of cache misses: instruction read miss, data read miss, and data write miss.

A cache read miss from an instruction cache generally causes the most delay, because the processor, or at least the thread of execution, has to wait (stall) until the instruction is fetched from main memory.

A cache read miss from a data cache usually causes less delay, because instructions not dependent on the cache read can be issued and continue execution until the data is returned from main memory, and the dependent instructions can resume execution.

A cache write miss to a data cache generally causes the least delay, because the write can be queued and there are few limitations on the execution of subsequent instructions. The processor can continue until the queue is full.

One significant contribution to this analysis was made by Mark Hill, who separated misses into three categories (known as the Three Cs):

*Compulsory misses* are those misses caused by the first reference to a datum. Cache size and associativity make no difference to the number of compulsory misses. Prefetching can help here, as can larger cache block sizes (which are a form of Prefetching). Compulsory misses are sometimes referred to as Cold Misses.
**Capacity misses** are those misses that occur regardless of associativity or block size, solely due to the finite size of the cache. The curve of capacity miss rate versus cache size gives some measure of the temporal locality of a particular reference stream. Note that there is no useful notion of a cache being "full" or "empty" or "near capacity": CPU caches almost always have nearly every line filled with a copy of some line in main memory, and nearly every allocation of a new line requires the eviction of an old line.

**Conflict misses** are those misses that could have been avoided, had the cache not evicted an entry earlier. Conflict misses can be further broken down into *mapping misses*, that are unavoidable given a particular amount of associativity, and *replacement misses*, which are due to the particular victim choice of the replacement policy. [Wikipedia]

Cache coherence introduces a new group of misses, communication misses. They are misses that occur because of actions taken by other cores.

**Communication misses** can be further separated in to two sub-classes, invalidation misses and upgrade misses. Invalidation misses occur whenever a local access misses because of an invalidation request from another core. Upgrade misses on the other hand occur when a line was available in the exclusive or modified state and later downgraded to the shared state prior to a write.

**False sharing** denotes the situation that two threads repeatedly write to data in the same cache line, causing the ownership of the cache line to move back and forth between the two caches of the two threads. Additionally, the two threads are accessing completely different addresses, so if these two data items could be separated more, say to two separate cache lines, the implicit communication between the caches would disappear.

False sharing is notoriously difficult to detect, but the remedy is simple. Move the different data objects further apart. However, for the detection part, Acumem tool ThreadSpotter makes the job easier for us in our case studies. [Acumem Freja Tutorial: “Communication Misses” and False Sharing]

### 1.7 Memory Bandwidth & the Memory Wall

It is not only the latency of the memory accesses that can limit the execution speed of an application. There is also a limited ratio at which data can be read from main memory, the memory bandwidth limit.

When the memory bandwidth limit is reached it is no longer the latency of the memory accesses that limits the execution speed, but the number of accesses to main memory that the application causes. The only way to improve performance is then to reduce the fetch ratio of the application, so that the number of main memory accesses is reduced.

When a program hits the memory bandwidth limit some optimizations intended to reduce the impact of memory access latencies become ineffective, or even reduce performance. For example, prefetching does not reduce the number of main memory accesses, so it
loses its effect. Instead, prefetching often leads to more main memory accesses since some data that isn't used is usually prefetched, decreasing the performance of the application.

For single-core processors it is relatively unusual that applications hit the memory bandwidth limit. Very high fetch ratios are required for this to happen. However, with the current trend towards multi-threaded and multicore processors the computation power of processors is increasing very rapidly, while the memory bandwidth is increasing much more slowly. This makes the memory bandwidth limitation one of the biggest problems for scaling application performance on multicore processors.

Applications that manage to stay below the memory bandwidth limit on single-core processors and with problems that were hidden by hardware or software prefetching, may suddenly hit the memory bandwidth limit when parallelized on multicore processors. It is not unusual that applications get no speed-up at all. Some applications even get reduced performance, since threads start throwing each other's data out of the cache, further increasing the fetch ratio. [Excerpt from “Memory Bandwidth”: Acumem Freja Manual]

The "memory wall" is the growing disparity of speed between CPU and memory outside the CPU chip. An important reason for this disparity is the limited communication bandwidth beyond chip boundaries. From 1986 to 2000, CPU speed improved at an annual rate of 55% while memory speed only improved at 10%. Given these trends, it was expected that memory latency would become an overwhelming bottleneck in computer performance.

Currently, CPU speed improvements have slowed significantly partly due to major physical barriers and partly because current CPU designs have already hit the memory wall in some sense.

However, a new solution did arrive in the Intel Nehalem processor technology, where the memory channel is replicated three times to distribute requests for memory access made by the processing cores. This new technique of providing multiple memory channels, and not necessarily a single faster one, has solved the problem to some extent by routing memory access requests from different cores to separate memory channels. [Wikipedia: Memory Wall]
Chapter no.2

Concepts in Performance Analysis & Optimization
2.1 Metrics for Analyzing Optimization

Unlike traditional methods of collecting and analyzing application's performance information based on hardware counters, we use another methodology to analyze application's behavior and how it affects its performance. With help of Acumem tools we capture various properties of an application such as fetch ratio, miss ratio and fetch utilization. Metrics used for analyzing performance bottlenecks and data reuse opportunities in the application code, are described below.

2.1.1 Fetch Ratio

In a processor without any kind of prefetching the cache miss ratio is also the ratio of memory accesses that cause a fetch from main memory. However, with prefetching the number of cache misses may be much lower than the number of accesses the processor does to main memory.

Prefetching therefore introduces a new concept, the fetch ratio. This is the ratio of memory access instructions that cause a fetch from main memory. Note that writes normally cause a fetch of a cache line even though the data in the line isn't actually used.

The fetch ratio directly reflects the memory bandwidth requirement of the application's memory read/write accesses.

2.1.2 Miss Ratio

It is hard to tell from just the number of misses if cache misses are causing performance problems in an application. The same number of misses will cause a much greater relative slowdown in a short-running application than in a long-running one.

A more useful metric is the cache miss ratio, that is, the ratio of memory accesses that cause a cache miss. From the miss ratio you can usually tell whether cache misses may be a performance problem in an application.

The cache miss ratio of an application depends on the size of the cache. A larger cache can hold more cache lines and is therefore expected to get fewer misses.

The performance impact of a cache miss depends on the latency of fetching the data from the next cache level or main memory. For example, assume that you have a processor with two cache levels. A miss in the L1 cache then causes data to be fetched from the L2 cache which has a relatively low latency, so a quite high L1 miss ratio can be acceptable. A miss in the L2 cache on the other hand will cause a long stall while fetching data from main memory, so only a much lower L2 miss ratio is acceptable.
2.1.3 Fetch Utilization

Whenever a memory operation instruction causes data access to the main memory, blocks of contiguous data are fetched into the cache; the size of this data block equals cache-line size, which is always 64 bytes in our case. How much of this fetched data is actually used by the program, determines the fetch utilization of a single memory-access instruction, in the program. Fetch utilization of all the memory accesses within the application code determine the overall fetch utilization for the entire application.

Fetch utilization is related to the metrics of Fetch and Miss Ratio, and directly affects the memory bandwidth requirements of the application. Higher fetch utilization improves the fetch/miss ratios and limits the memory bandwidth required by the application.

2.1.4 Writeback Utilization

Similar to fetch utilization, writeback utilization for a single write-statement in the application code determines the percentage of data in the cache line (pulled into the cache due to memory-write access) that is actually changed (or written) as a result of memory write-access.

Like a read access to the memory, a write access too causes a block of contiguous data to be pulled into the cache. Where fetch utilization looks at how much data is read from a given cache-line, writeback utilization looks at how much of the cache-line brought into the cache, as a result of write access to the memory, is written or changed before being dumped back to the main memory.

2.1.5 Hardware Prefetch Probability

Hardware prefetch probability is the metric expressing percentage of successful prefetching of relevant data (or data that is actually used) by the hardware prefetcher for a given instruction. It is the ratio of size of prefetched data actually used to the size of all prefetched data for a given instruction.

Replacing irregular and random accesses to the main memory with more refined and regular access patterns in the application code can help improve the prefetch probability and thereby remove potential CPU stalls caused by data misses in the cache.

2.1.6 Percentage of Fetches

Every time a memory access causes data to be fetched from the main memory, it adds to the total number of data fetches made by the entire application. An interesting tool for detecting optimization regions within application code is to monitor the percentage of memory operations resulting in main-memory fetches, for different code-regions. Code regions and loops with high percentage of fetches and low fetch utilization are potential problem areas, and fixing the problem here means effective limiting of application's bandwidth requirement.
2.1.7 Percentage of Misses

Every time a read or write access suffers a cache miss, it adds to the total number of misses suffered by the application. To detect performance degradation due to cache miss stalls in the application code, considering percentage of memory operations resulting in cache misses for different sections of code is a very effective technique. Adding intelligent software prefetching here can help reduce cache misses and miss stalls suffered by the application during its execution.

2.2 Metrics for Analyzing Performance

Having worked with the optimization of the application code, such as removing memory bottlenecks and exploiting data reuse opportunities, we need measurement tools that could gauge achieved performance gains. Although performance improvements can be expressed in terms of improvements in fetch/miss ratios and fetch utilization, it is always good and exciting to have resulting speedup performance reviewed along with it. We describe certain useful performance analysis tools and techniques for expressing application performance improvements.

2.2.1 Application Speedup

This metric is used to express as to how much faster does the application run after being optimized. Application speedup is the ratio of execution time of Original application to the execution time of the Optimized application.

\[
\text{App. Speedup} = \frac{\text{Exec. time of Original app}}{\text{Exec. time of Optimized app}}
\]

2.2.2 Optimized code region speedup

Besides the overall application speedup, it is also very interesting to see the speedup of code regions or loops that have been optimized. Methodology for measuring execution times for such candidate regions in code may vary from programmer to programmer. The most practical suggestion here would be to place time recording commands around the candidate region. Other less accurate, but still good heuristics may include measuring the time for functions (or methods) where the code has been optimized.

Optimized code region speedup is the ratio of the execution time of the original candidate regions to the execution time of the corresponding optimized code regions.

\[
\text{Code region speedup} = \frac{\text{Exec. time of original code regions}}{\text{Exec. time of optimized code regions}}
\]

2.2.3 Relative Throughput

Relative throughput is the ratio of the maximum execution time of simultaneous executing application instances to the execution time of single original instance times number of instances executed in parallel.
Relative throughput = max exec. time of all instances / exec. time for single original instance * no. of instances

Relative throughput partly depends on the ongoing competition and contention between the running instances of an application for the shared resources on a multicore architecture. For instance if we execute 4 instances of an application on a quad-core with each instance bound to a different core, the relative throughput should be 4 if there is no contention between the executing instances for the shared resources. However, in case of competition for shared resources, this figure is always below 4. In cases of extremely resource-hungry applications, the relative throughput might drop below 1, resulting in “super linear slowdown”.

Good throughput scalability for any given application is shown by growing relative throughput, keeping close to the ideal curve, with added number of instances (for equal number of cores). For optimized applications, relative throughput is an effective measure for expressing how sensitive the application performance is to shared resources.

2.2.4 Parallel Speedup

Parallel speedup is the measure for expressing how fast the parallel application, whether optimized or not, runs with respect to the original serial application. It is the ratio of the execution time for the parallel application (with varying number of threads) to the execution time of the original application.

Parallel speedup = exec. time of parallel application / exec. time of original serial application

This performance measurement is especially important for analyzing performance improvement due to optimization of “memory wall” problems in the code, through removal of memory bottlenecks and exploiting of data reuse opportunities, and performance gain achieved as a result of adding parallelism. This particular metric would answer the question as to whether parallelism is really required to improve application performance or does plain optimization dealing with memory bottlenecks give good enough results for the given application; and to what extent would the performance be affected with parallelism added after optimization. It is also interesting to study scalability of parallel applications, after having optimized them for the removal of problems posed by the memory wall.
Chapter no.3

Acumem Tools & Selected Applications
3.1 Acumem Tools for Optimization Analysis

For the given case studies in this report, we use the Acumem tools for analyzing performance optimization opportunities in various application codes. Acumem tools' runtime analysis of applications bases on their memory-access pattern and generates advice for the programmer to help improve the application’s performance.

3.1.1 Methodology

Acumem tools use a proprietary light-weight sampling technology when sampling the application, not hardware performance counters. The collected fingerprint is richer in information than what can be obtained from the hardware performance counters and allows metrics such as utilization of fetched cache lines to be calculated.

Since it is the behavior of the application and not the hardware that is sampled, the gathered data is independent of the hardware the sampling was done on. This allows the user to analyze how the application would behave on a processor with a different cache configuration, for example, to predict the performance on different processor models.

Acumem tools make use of various performance metrics such as fetch ratio, miss ratio and fetch utilization to name a few, to point out regions in application code with memory bottleneck problems and data reuse opportunities. Acumem tools also have strong heuristics employed for detecting communication bottlenecks in threaded applications, for multicore architectures. The analysis and advice provided here too is architecture independent and portable to other multicore architectures.

3.1.2 Acumem SlowSpotter

Acumem SlowSpotter analysis focuses on the memory access behavior of serial or single-threaded applications, and specifically on how the application's memory access patterns interact with the processor caches.

As described above Acumem SlowSpotter produces the analysis results in terms of meaningful metrics such as fetch ratio, miss ratio etc. as a function of cache-size, also providing a well calculated estimate of how much improvements can be made for all such performance indexes, for any given application. Measuring and expressing deficiencies in terms of such meaningful metrics helps in coming up with optimization solutions that are easily workable on hardware platforms with similar cache-architecture properties.

In addition to that, Acumem SlowSpotter supplies required advice to resolve each performance issue within the code. Simply acting on the provided advice should be good enough to resolve all issues and improve the performance up to the estimate provided by the analysis report.
Acumem SlowSpotter is programming language independent. It only looks at the binary code of the application. It can generate a report without any source code, but to generate source code references it needs access to the source code and the application must be compiled with debug information in a standard format in order to identify the affected source code lines.

### 3.1.3 Acumem ThreadSpotter

Acumem ThreadSpotter is an extension to the Acumem SlowSpotter, enabling analysis of multi-threaded applications for multicores. Acumem ThreadSpotter, besides providing advice for resolving issues related to the “memory wall”, also provides advice for resolving every single performance issue related with thread communication in a threaded application. Acumem ThreadSpotter accurately points out inefficiencies in the application caused by communication bottlenecks between threads. Issues in communication may be due to low communication utilization or false-sharing of shared data between different cores. A ground-breaking technique employed by Acumem ThreadSpotter is used to detect False-Sharing of data caused by threads operating on the same data cache-line, but in different caches.

### 3.2 Applications for Case Studies

To demonstrate the applicability and effectiveness of performance improvement code-analysis using architecture-independent metrics, as used by the Acumem tools, we have studied four different software applications from various different fields such as Scientific Computing, Algorithms, Combinatorics and Artificial Intelligence; the studied applications are used extensively for simulations of physical phenomena, solving real world problems (like traffic flow and control) and for solving problems in Bioinformatics and Biotechnology. We briefly list and describe the applications studied here

- **LBM** – LBM abbreviates “Lattice-Boltzmann Method”, a method of computational fluid dynamics (CFD), often used in solving scientific computing problems. The LBM application is an implementation of such a system applicable to simulations of behavior of Newtonian fluid. The application studied here is provided as a part of the SPEC CPU2006 suite's floating-point applications, referred to as 470.lbm. Code optimization for this application involves a combination of re-arranging memory-access pattern of the data structure used and the general blocking technique, making the application run three times as fast as the original application.

- **MCF** – The MCF application is a network simplex implementation used to solve single depot vehicle scheduling problem. An application based on combinatorial problem solving logic, it is applicable to several planning problems. This application uses it for planning of mass public transportation. The specific application studied here is provided by Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB) and is part of the SPEC CPU2006 suite, referred as 429.mcf benchmark. Through carefully isolating the data, required in the most
data intensive loops of the application code, by replicating it in a separate memory pool, we make the application run almost twice as fast.

- **CIGAR** – CIGAR is an abbreviation for “Case Injected Genetic AlgoRithm”, and is an implementation of a unique genetic algorithm method employing case-based reasoning to approach towards solution quicker. Based on techniques from machine learning and artificial intelligence, the application is directed towards use in strategy games, dynamic decision makers and evolutionary user interface design. The CIGAR application is an open-source application provided by the Evolutionary Computing Systems Lab (ECSL) at the University of Nevada, Reno. Similar to the MCF case study, the optimization steps include required data’s isolation and replication for use in the most data intensive loops of the application code. The optimizations show significant gains, speeding up the application by over a factor of thirty.

- **Libquantum** – The Libquantum application is an implementation to simulate Quantum Computers based on the principles of Quantum mechanics. The application contains an implementation of Peter Shor's algorithm for factorization of numbers. The application considered here is provided as a part of the SPEC CPU2006 suite's integer applications, referred to as 462.libquantum. Optimizations for this application include re-arrangement of memory-access pattern, loop fusion and function fusion, resulting in application speedup of factor three.

All applications studied here are written in C.

Applications are always optimized for the highest or the last level of cache. This may vary from one architecture to another, depending upon the number of cache levels in the given architecture’s cache hierarchy. Applications studied here have been optimized for level 2 caches.

Having explained the background and basic concepts we build on, we now proceed on to the real case studies and describe analysis, optimization and performance tests for every application in detail. Each case study is described in a self-contained way to allow for the studies to be read in any order.

First we proceed to describe the optimization methods and techniques used in our case studies for improving performance of the applications considered here.
Chapter no.4

Generalizing Techniques Used for Optimization
4.1 Data Splitting

Computation and data intense program loops/sections require to pull a lot of data from the main memory into the cache. An inefficient implementation of a data intense loop would require the application to stall and wait every time it wants to use some data, as it has to be pulled into the cache from the main memory. One of the reasons for such a bad implementation could be wasting of cache space, due to data that is fetched but never used and eventually evicted. Also, pulling in unwanted data results in effective waste of memory bandwidth, which may even result in late arrival of data into the cache due to bandwidth unavailability; this is very likely to be the case in data intense loops.

Programmers often use structures to group all related data. In scientific applications this might not always be very efficient as parts of the program may not use all the data in the structure, resulting in a lower fetched data utilization for that part of the application. This may cause serious slowdown of the application and waste of precious memory bandwidth, if the fetched data is not utilized properly in data intense program loops and/or sections.

One way to avoid such a scenario is not to reference (pull into cache) the data that will never be used in the near future. In the specific case of the use of structures within a data intense loop, we can avoid this by breaking up the structure into two (or perhaps more) smaller structures.

Let's look at an example here. Consider the following structure.

```c
struct mystruct
{
    int *iptr_1, *iptr_2, *iptr_3;
    long lng_1, lng_2;
    double dbl_1, dbl_2, dbl_3, dbl_4;
}
```

The size of the above structure is 72 bytes (8 bytes more than the cache line, 64 bytes, in our case). When we reference an object of the above given type, contiguous data worth two cache lines size (128 bytes) is fetched into the cache, from the main memory. The 56 bytes next to the referenced structure may be from another such structure which may never be used. Even worse is the case when such data is fetched in data intensive loops and not utilized completely.

To avoid such a condition, we break up the structure definition as follows. Here we assume that the long and the double data types are never used in certain data intensive loops (ones being optimized) of the programs. So we make a cold data structure and shift the unused datatypes in it and add a pointer to such a structure in the original 'mystruct'.
Doing so, the programmer should take care of properly initializing both of the above structures. However, it should be noted that to access the cold data through the "mscd_ptr" we still need to pull in the 24 bytes (iptr_1, iptr_2 and iptr_3) of 'mystruct'. Again, if this additional data is still not required in the program sections making use of the cold data, then it is best to completely split the structure into two separate structures (without adding any pointer to the related cold data structure) and manage related data through indexing of both structure types. This is especially easy when the application works with vectors of such data structures.

Data splitting may however incur the cost of readability of the code and may make it difficult for a programmer looking at the code for the first time. It may also cause maintainability problems when trying to extend the codebase to accommodate added functionality. Therefore, it is always advisable to go for such a solution once the application has been completed and all necessary and related functionalities have been included as well.

4.2 Data Replication

Sometimes it is not feasible or desirable to breakup a structure into more than one smaller structure that can be used by data intensive loops more efficiently, due to several reasons like complexity, maintainability etc. This can however, be avoided by carefully replicating the data required (from structure objects) in the loop into smaller data structures. This can also help avoiding complexities in grouping related data.

For instance if a loop requires to check a single component variable of all the structure objects in a tree or linked list, that variable can instead be vectorized into an array representing that particular data part for the entire data structure. This process of vectorizing would include replication of the particular data part into the array and careful indexing and updating of the data in the vector whenever that data part is updated, changed or modified in the corresponding structure object. This technique necessarily involves data redundancy of the node objects of a dynamic data structure. In the loop, the program can make use of the vector instead of having to pull in the structure object every time just for its single component variable; moreover, the vector array would be easier to fit into the cache, lowering the fetch and the miss ratio significantly. In addition to the
above this is another way to avoid the randomness problem when accessing data, especially from structures.

![Figure 1: Vectorizing useful data from structure pool](image)

However, this technique is practical only in cases where the data being replicated into a separate pool is not changed frequently, since a dataset prone to frequent changes would require the corresponding replicated data to be updated at the same time to maintain exact behavior of the program. Frequent data update in the pool of replicated data rather becomes an overhead.

### 4.3 Memory Pooling & Burst Memory Allocation

Placing structure objects non-contiguously in the main memory (such as in dynamically growing linked-list, tree or heap) can cause Randomness (in data access) problems and slow down a data intensive loop significantly.

For data structures that grow dynamically (linked-list, tree or heap), programmers often follow a naive approach of allocating memory to new nodes (structures) only when require. Normally when an application starts-up it builds data structures filling them with the initialization data. This data is then used (and modified) once the application starts running; this might as well grow or reduce the data structure at runtime.

Program logic allocating space to data structures should be written such that memory space in the initial stages of the application startup is allocated in excess of what is actually required by the dynamic data structure. This is done to lower time complexity spent on system calls later when the program runs; more importantly, later in the execution stages of the program, when the data structure requires to be grown (to add more data to it), the program can use the already available space (that was initially created in excess).

Burst memory allocation also avoids fragmentation and makes nodes in a data structure more contiguous in memory. This also helps the hardware prefetcher since mapping the
data structure in the memory does not involve unpredictable strides, caused by fragmentation due to time-to-time memory allocation (and de-allocation).

Also, when the data structure needs being reduced in terms of the number of nodes, similar measures should be taken into account to always maintain a suitable amount of memory in excess.

This reduces the memory access randomness problem, due to lower fragmentation between the data nodes of the data structure. This approach helps the application make more use of spatial locality of data in the cache.

4.4 Grouping Related Data

Programmers often arrange data in simple arrays rather than complex data structure whenever they find the opportunity to do so. Although this does make the code less intuitive, however this practice is mainly aimed at making room for performance enhancement. This gives programmers the freedom to arrange the data in memory as they find suitable.

Such simple data arrangements should carefully map related data to contiguous memory locations. Doing so helps in better cache utilization and improving the fetch and miss ratio.

Inefficient looping is a good example showing underutilization of spatially located data. Consider the following example of a loop iterating over an array, accessing or indexing the array with an inefficient stride factor.

```c
long int arr[100];
int i;

for(j = 0; j < 5; j++)
    for( i = 0; i < 10; i += 5 )
        //print arr[ i ]
```

The above piece of code will cause a cache miss every second iteration, causing the program to wait until the new cache line containing the required element is fetched into the cache.

The best way to deal with such a problem is to restructure the data placement of the entire array such that the stride is reduced to a factor where it stops missing more often in the cache, at best 1. This can be done by grouping “related data” within the array into contiguous memory locations. However, having done this, the data placement and indexing of the data structure might not be as intuitive for a new programmer as before, but shouldn't be a problem for seasoned programmers.
This is how the above code may look after grouping of data

```plaintext
max = 100 / 5;
for(j = 0; j < 5; j++)
  for(i = 0; i < max; i++)
    //print arr[j * max + i]
```

The above example can also be represented with a simple animation as shown in Figure 2. Related data in a given array is shown with identical colors. The idea here is to have the piece of data with identical colors to be contiguous in memory. This would require remapping of the related data, by the programmer.

![Figure 2: Grouping related data](image)

In the real world, problems might be more complex than this though.

### 4.5 Software Prefetching

Once remedies for bandwidth conservation have been applied to places in the code using unnecessary bandwidth, the programmer can go on to apply software prefetch within the code to reduce the miss ratio for the parts of the program having very high miss ratio. Software prefetching means touching of data expected to be used in near future, and thereby bringing it into the cache. Again care should be taken while doing so in data intense code regions in order to avoid choking of the memory highway with too much data. The prefetch should not be too early so that the data may be evicted from the cache prematurely. Prefetching should be applied to data pieces that are expected to be used a few hundred cycles later. However, it should not be late enough to stall the requesting instruction, making the prefetch instruction ineffective to produce any performance improvement.

### 4.6 Loop Fusion

Loop fusion is a kind of loop transformation technique (a compiler optimization technique) used to merge several similar loops into a single loop. Through merging, all the operations being performed by the different loops are performed by the resulting single fused or merged loop.
The two adjacent loops on the code fragment below can be fused into one loop.

```
for (i = 0; i < 300; i++)
    a[i] = a[i] + 3;

for (i = 0; i < 300; i++)
    b[i] = b[i] + 4;
```

Below is the code fragment after loop fusion.

```
for (i = 0; i < 300; i++)
{
    a[i] = a[i] + 3;
    b[i] = b[i] + 4;
}
```

The above given example is very simple and real-life applications may have loop-fusion opportunities in much different and even difficult scenarios. In certain cases fusible loops may not exist in the same function. In such cases, opportunities for merging the functions into a single function and then merging the “alike loops” should be exploited. However, this is not always easy and the solution may be very problem dependant.

Loop fusion is not commonly available in C compilers as an optimization.

Although loop fusion reduces loop overhead, it does not always improve run-time performance, and may reduce run-time performance. For example, the memory architecture may provide better performance if two arrays are initialized in separate loops, rather than initializing both arrays simultaneously in one loop.

### 4.7 Parallelizing Computation & Data Intense Program regions

Software that has already been written may not always be easy to parallelize wholly. Some applications may not be parallelizable at all. The simplest and most straightforward way of parallelizing an existing code is to profile the application with tools such as PROF, GPROF, VTune or others with good precision and accuracy and analyze program regions executing most of the time during the application's total runtime. Being able to parallelize even 40% of any application's computation intense regions effectively is bound to produce significant runtime improvements.

Parallelizing computation and data intense code regions, the programmer should consider breaking of the problem into smaller sub-problems capable of being solved almost individually by the worker threads. Being able to replace a serial algorithm with its true parallel counterpart is most effective in achieving task division into smaller sub-tasks that can be taken care of by the worker threads privately and more individually requiring least amount of inter-process communication. However, while doing so, parallel algorithms should be tweaked to run well on multicores. While distributing workloads to worker threads, fairness should be addressed so that threads don't wait for each other for long in parallel regions.
4.8 Minimizing Inter-process Communication

One of the distinct contrasts in multi processor and multicore programming is inter-process (or inter-processor) communication. Although compared to multi processors, communication in multicores isn't as expensive, however, very frequent synchronization should be avoided. Communication and computation balance should be maintained. Compared to the original code, if the computation cost of the parallel region goes down significantly, we get much room for inter-process communication and maybe some room for expensive thread synchronization as well. Taking decisions on this aspect depends very much on the programmer’s insight of the problem, as this is very problem dependent.

Minimizing communication over shared data structures and dividing it over individual task threads can be done effectively by specifying regions/bounds within the shared data structure for each worker thread to work on and treating the data structure as private to each thread. It is often the case that the component data nodes of the shared data structure are located contiguously in the memory. This might cause data inconsistency when threads try to work on their part of the data structure as they might pull in data belonging to the next thread in the memory (due to spatial locality principle of caches or prefetch by the Hardware prefetch utility); this may cause an inconsistent view of the memory at different processors. This can be avoided by interleaving neighboring “thread-regions” of the data structure with memory padding through dummy memory allocation when the data structure is being allocated memory initially; this interleaving region will never actually be used by any of the threads.

Data partitioning for allocation to threads can be done in different ways, the most simple being Horizontal and vertical partitioning of the dataset. Figure 3 depicts a dataset partitioned horizontally with unused data space.

![Figure 3: Grouping related data](image)
4.9 Resource Mapping: Avoiding unnecessary Thread Migration

For parallel applications that frequently move in and out of parallel regions, it is best to avoid unnecessary thread migration or thread rescheduling to different processor as this may cause the threads to stall over missing data in the cache; missing data that is already present in another processor's cache (from the threads previous execution there). Doing so may also require resource mapping (such as processor binding) at the time of application startup such as use of “numactl” or “taskset” commands available in Linux.

Having disabled varying thread count within parallel regions of a program (for OpenMP this can be done by setting the environment variable 'OMP_DYNAMIC' to 'false'), if the worker threads tend to make use of the same data (available from previous fetches of the same worker threads), then in such cases it is best to avoid thread migration and maintain consistency in resource allocation for thread scheduling, to make sure that the threads are scheduled over the same cores whose cache contains the thread's data from its previous execution there. Static scheduling (available in OpenMP as well) is one way of restricting worker threads to particular cores or chips.
Chapter no.5

Case Study 1 – LBM
5.1 Introduction

The LBM application is an implementation to simulate incompressible fluids in 3D using the “Lattice Boltzmann Method”. The LBM method implementation is an important part of simulations used in Materials science and Fluid dynamics.

The particular LBM implementation under consideration for this case study is provided by the SPEC CPU2006 floating-point benchmark 470.lbm. This application has a particularly bad reputation for relative throughput. We add optimizations to the code to show possible improvements in performance.

Details about the benchmark as provided by SPEC CPU2006 can be found here


This case study discusses only the issues that have been solved to improve the application performance. Other issues have been ignored.

It is recommended to have the code for SPEC CPU2006 benchmark 470.LBM when looking into this case study.

5.2 Acumem SlowSpotter's Diagnosis

Profiling the mentioned version of the LBM application with the Acumem SlowSpotter tool, the report immediately points out at areas of possible high performance gains. SlowSpotter complains about Memory Bandwidth, Memory Latency and Data Locality problems and lists them in order of estimated performance gains, placing the code with the highest estimated gain at the top.

Figure 4 shows the “Bandwidth issues”, ranked by the percentage of fetches, in the issues section of the report.
Figure 5 shows the loops in order of worst loop stats (or highest expected performance gains), in the loops section of the report.

Figure 5: Loops with highest expected gains (left pane) and loop in code (right pane)
Figure 6 shows the summary of the overall sampled application

<table>
<thead>
<tr>
<th>Issues</th>
<th>Loops</th>
<th>Summary</th>
<th>Files</th>
<th>Execution</th>
<th>About/Help</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Global statistics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miss ratio</td>
<td>8.8%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch ratio</td>
<td>10.2%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Writeback ratio</td>
<td>7.5%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Upgrade ratio</td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication ratio</td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch utilization</td>
<td>24.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-back utilization</td>
<td>32.1%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication utilization</td>
<td>100.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Analysis parameters</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor model</td>
<td>Intel(R) Pentium(R) D CPU 3.20GHz (auto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of CPUs</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of caches</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache level</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache size</td>
<td>2M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line size</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replacement policy</td>
<td>random</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 6: Summary of Application statistics*

The statistics for the LBM application show overall fetch ratio at 10.2%, miss ratio at 8.8% and fetch utilization of just 32.1%
Acumen SlowSpotter's diagnosis immediately points out at the following major problems. Several problem statements have been pointed out. We number each problem statement according to its category and deal with the group of statements accordingly.

In the following loop (in function 'LBM_performStreamCollide')

```c
SWEEP_START( 0, 0, 0, 0, 0, SIZE_Z )
if( TEST_FLAG_SWEEP( srcGrid, OBSTACLE )) {
    PROBLEM 1
    DST_C ( dstGrid ) = SRC_C ( srcGrid );
    DST_S ( dstGrid ) = SRC_N ( srcGrid );
    DST_N ( dstGrid ) = SRC_S ( srcGrid );
    PROBLEM 2
    DST_W ( dstGrid ) = SRC_E ( srcGrid );
    DST_E ( dstGrid ) = SRC_W ( srcGrid );
    DST_B ( dstGrid ) = SRC_T ( srcGrid );
    PROBLEM 2
    DST_T ( dstGrid ) = SRC_B ( srcGrid );
    PROBLEM 2
    DST_SW( dstGrid ) = SRC_NE( srcGrid );
    PROBLEM 2
    DST_SE( dstGrid ) = SRC_NW( srcGrid );
    PROBLEM 3
    DST_NW( dstGrid ) = SRC_SE( srcGrid );
    PROBLEM 2
    DST_NE( dstGrid ) = SRC_SW( srcGrid );
    PROBLEM 2
    DST_SB( dstGrid ) = SRC_NT( srcGrid );
    PROBLEM 2
    DST_ST( dstGrid ) = SRC_NB( srcGrid );
    PROBLEM 2
    DST_NB( dstGrid ) = SRC_ST( srcGrid );
    PROBLEM 2
    DST_NT( dstGrid ) = SRC_SB( srcGrid );
    PROBLEM 2
    DST_WB( dstGrid ) = SRC_ET( srcGrid );
    PROBLEM 2
    DST_WT( dstGrid ) = SRC_EB( srcGrid );
    PROBLEM 2
    DST_ET( dstGrid ) = SRC_WB( srcGrid );
    continue;
}
rho = + SRC_C ( srcGrid ) + SRC_N ( srcGrid )
     + SRC_S ( srcGrid ) + SRC_E ( srcGrid )
     + SRC_W ( srcGrid ) + SRC_T ( srcGrid )
     + SRC_B ( srcGrid ) + SRC_NE( srcGrid )
     + SRC_NW( srcGrid ) + SRC_SE( srcGrid )
     + SRC_SW( srcGrid ) + SRC_NT( srcGrid )
     + SRC_NB( srcGrid ) + SRC_ST( srcGrid )
     + SRC_SB( srcGrid ) + SRC_ET( srcGrid )
     + SRC_EB( srcGrid ) + SRC_WT( srcGrid )
     + SRC_WB( srcGrid );

/***********************
 * Skipping code here  *
 ***********************/

SWEEP_END
```

In the above code, the macros starting with ‘DST’ point to elements in the ‘dstGrid’ array whereas the ‘SRC’ macros index elements within the ‘srcGrid’ array. The macro in the first line of the code “SWEEP_START” translates to a “for” loop, whereas the macro “SWEEP_END” ends this loop. The first three lines of the code can be expanded as follows
for( i = 0; i < 26000000; i += 20 /*N_CELL_ENTRIES*/ ) {
    if( (*((unsigned int*) ((void*) (&((((srcGrid)[ 19 /*FLAGS*/+ (i) ])))))) & (OBSTACLE))) {
        dstGrid[ ( (C) + 0) + (i) ] = srcGrid[ (C) + (i) ];
    }
}

Where in the above expanded code, we have constants ‘C’ and ‘OBSTACLE’. Macro definition for “SWEEP_START” is shown in section 5.4

Problem 1:
The report shows Problem 1’s fetches to be 91.5% of all the memory fetches in this application, with a fetch utilization of only 24% for the statement in the first problem (see in Figure 15). The instruction stats show the percentage of misses to be 90.6% of all the cache misses in this application, and fetch ratio at 18.3% and miss ratio at 15.5%. Reducing the fetch and miss ratio for this statement would greatly help improve bandwidth issues. Also, hardware prefetch probability here is very low at 15%.

Why Problem 1?
The high miss and fetch ratio for the 'if' statement in this problem come from the fact that 'srcGrid' being a simple one dimensional array (logically interpreted as a 4D array) is 256 MB in size and is too big for the cache, as the ‘for’ loop touches each element of the array exhaustively. While iterating through the array with stride 20, it tests the element at an offset of 19 from 'srcGrid[i]' (against 'OBSTACLE'). The 20 factor stride pattern in the ‘for’ clause is followed by successive fetching of 3 cache lines of contiguous data causing at least two cache misses every single iteration; it is here where the Hardware prefetch utility fails, since while fetching the contiguous cache lines of data the addressing pattern of the first element-to-be-fetched keeps changing between successive iterations, lowering the Hardware fetch probability to as much as 15.5%. The fetch utilization of 24% is not well understood. It is probably because of the exchange of ‘srcGrid’ and ‘dstGrid’ pointers taking place one step above in the call stack, before a subsequent call of the same function “LBM_performStreamCollide”.

Problem 2:
We consider all the highlighted statements within the 'if' clause to be included into the second problem since all of them are related. Within this problem we restrict ourselves to the problem caused by write misses and low utilization.
All the statements show a cumulative miss and fetch ratio of almost 24% with almost no Hardware fetch probability of 0.3%. These instructions contribute 7.6% and 8.8% to the entire application's memory fetches and cache misses respectively. Improving fetch and miss ratio here can help avoid stalling due to misses or memory latency during iterations.

Why Problem 2?
Here the main problem comes from addressing the destination array 'dstGrid', which is identical in type and size to 'srcGrid'. However, the problem here is that unlike the 'srcGrid', the 'dstGrid' is indexed almost randomly and not a single element of the 'dstGrid' addressed during each iteration makes any use of spatially located data in the cache (since each one of the indexed element is placed far off in the main memory). Data fetched in previous iterations due to contagiousness in memory, is never used since successive iterations index each of the elements of the 'dstGrid' with a factor of 20 (160
bytes) thereby causing a miss every time a 'dstGrid' element is addressed. Data reuse is not an issue here, since data from both 'srcGrid' and 'dstGrid' is never reused. Also, the hardware prefetcher is unable to rightly predict the location of the upcoming element, since there is no pattern in successive indexing of the 'dstGrid'; though instruction based hardware prefetching might work well here.

Problem 3:
Miss and fetch ratio for the single statement in this problem is reported three times at almost 22%, 21% and 26% for the miss ratios and a cumulative 13.5% for the fetch ratio of the entire statement.

Why Problem 3?
Misses occur at this point due to unavailability of required data from the 'srcGrid' since it needs to be pulled in after the 'if' test. Miss and fetch ratios are reported three times probably due to the fact that the 19 successive elements addressed in 'srcGrid' take 3 cache lines and three misses occur every time this statement executes, first when 'SRC_C' is addressed, second when 'SRC_NW' is addressed and then finally when 'SRC_EB' is indexed. Actually the same sequence of read misses would occur for the operations on 'srcGrid' inside the 'if' statement, since like the statement in Problem 3, all 19 elements from 'srcGrid[i]' onwards are addressed there too.

5.3 Acumem SlowSpotter's Advice

Problem 1:
Acumem SlowSpotter advises
1. correction of inefficient loop nesting, and
2. Spatial blocking, identifying the need to reuse data and pointing out the problem of data access after its eviction from the cache.
3. improvement in fetch utilization

Problem 2:
Acumem SlowSpotter advises
1. avoiding randomness in data accesses (due to very large strides)
2. Spatial blocking for data reuse and identifying the problem of required data's immature cache eviction
3. improvement in fetch utilization

Problem 3:
Acumem SlowSpotter advises
1. correction of inefficient loop nesting
2. Spatial blocking for data reuse and identifying the problem of required data's immature cache eviction

5.4 Steps for Optimization

Note: Please refer to code in Figure 8 to see changes made in every point
1. We start by restructuring the access pattern of the 'dstGrid' in an attempt to remove the stride of 20 while going through the data structure can help improve spatial locality for the application's data in the cache. As mentioned earlier the grid array data structure actually represents a 4D array with each cell in the X, Y and Z dimensions having an array of 20 cell entries. This array of 20 cell entries for each XYZ-cell is located within the grid one after another in a successive order, i.e. 20 entries for one cell are located together, followed by the 20 cell entries for the next XYZ-cell. In an effort to improve data locality for reuse purposes, we group 'related' cell entries of all cells such that they occur in a consecutive sequence in the memory. Such as grouping of 'DST_C' for all XYZ-cells in 'dstGrid' into one group; same goes for the rest of the cell entries. Hence remapping the data layout, results in the related cell entries to be ordered into 20 groups, with each group containing the corresponding cell entry for all the cells in the XYZ plane. Prior to this, in the original version, all the cell entries were accessed with a stride of 20 elements (160 bytes) in successive iterations, in the memory. This stride is now reduced to 1.

Having made the recommended changes in step 3, this is how the Grid indexing macro (in 'lbm.h') looks like

```c
#define CALC_INDEX(x,y,z,e) ((x)+(y)*SIZE_X+(z)*SIZE_X*SIZE_Y+(e)*VOLUME_XYZ)
#define SWEEP_START(x1,y1,z1,x2,y2,z2) 
  for( i = CALC_INDEX(x1, y1, z1, 0); 
      i < CALC_INDEX(x2, y2, z2, 0); 
      i ++ ) {
```

Previously, in the original version, it looked like this

```c
#define CALC_INDEX(x,y,z,e) ((e)+N_CELL_ENTRIES*((x)+(y)*SIZE_X+(z)*SIZE_X*SIZE_Y))
#define SWEEP_START(x1,y1,z1,x2,y2,z2) 
  for( i = CALC_INDEX(x1, y1, z1, 0); 
      i < CALC_INDEX(x2, y2, z2, 0); 
      i += N_CELL_ENTRIES ) {
```

Look at how the stride in the for loop has changed from “i += N_CELL_ENTRIES” to “i++”. This simple change causes significant decrease in the overall fetch and miss ratio. Other macros and statements not shown here are changed accordingly.

2. Having changed the access pattern of the Grid data structure to a more suitable indexing, making optimal use of data locality, we go on to 'icing the cake' with yet another more general optimization and data reuse technique called 'blocking'. This might sound unnecessary in the beginning, but it shows astounding results once implemented. This optimization may as well be argued to be a necessary addition for better performance, as we still fear premature data eviction from random replacement policy (used by many of the caches out there); this due to making several accesses to
far off addresses within the 'dstGrid' before returning to the very first one in the next iteration. Since all the data streams from the 'srcGrid' and 'dstGrid', 38 in total, are accessed independently and placed far off in memory, they can be logically thought of as a two dimensional array (of dimensions 38x1.34 million) in this loop. Blocking over such a logical structure would guarantee immediate reuse of the data that has been brought into the cache. Constructing this technique for the given problem, instead of simply iterating through the entire 'srcGrid' and 'dstGrid' streams, we now logically divide the two grid arrays into blocks and iterate over the blocks instead; having completed a round of iteration over one block we repeat the iteration over the next. We divide the data streams from the 'srcGrid' and the 'dstGrid' into 4 groups (can be less or more than 4; but 4 in our case) each dealing with 5 elements, except for the first group (since we do a bit more work here). In effect, our block size would be 8x32 elements in the first case (accessing 32 elements of the first 4 streams from 'srcGrid' and corresponding 4 streams from 'dstGrid') and 10x32 (accessing 32 elements from next 5 streams from 'srcGrid' and corresponding 5 streams from 'dstGrid') in the rest, as shown in the code excerpt below. This technique would help immediate reuse of spatially and temporally located data in the cache, lowering the fetch and the miss ratios significantly.

Figure 7 animates the transformations to the code applied in the above mentioned steps.
Figure 8: Optimized code
With these changes we introduce temporal reuse of data, as our blocking technique moves back and forth over the 'srcGrid' data in the 'if' statement. Also, rho, ux, uy, uz and u2 variables are reused in blocking.

The above improvements in the code are bound to produce effective results which should be clearly visible in application's run-time, since profiling the application reveals that the function containing the above code runs for 99.8% of the total application's duration. Lowering runtime for this piece of code means lowering of the entire application's runtime.

5.5 Performance Evaluation

We evaluate the performance of the optimized and the original LBM applications using more than a single platform in an attempt to get a realistic view of how the two applications would behave in the real world on different platforms, they are deployed on.

On an Intel QuadCore E5345 (64-bit Xeon processors) running at 2.33 GHZ with L2 cache of 4MB, running the original LBM application takes 2553 seconds. Whereas, running LBM application with the optimized code takes just 772 seconds, which is almost thirty minutes faster than the original application.

These results show a runtime drop of almost 70% for the entire application.

Measuring the speedup of the application after optimization, we see that the optimized application runs more than three times as fast as the original one at 3.3X. Imitating the speedup behavior of the entire application, the optimized function runs more than three times as fast as the unoptimized function in the original application, at 3.5X.
Figure 9 shows the performance gains in application speedup and optimized function.

Profiling the two applications with the GPROF tool shows that in the original version of the application, the function ('LBM_performStreamCollide') with the unoptimized code ran 99.8% of the total application's runtime. Whereas, for the optimized code, the same function runs for 99.5% of the entire application's runtime. Runtime figures from GPROF show this function's total runtime is lowered by 71.3%, thereby lowering the entire application's runtime.

All of this speed-up is the result of the optimization steps listed in the previous sections, focusing mostly on a single loop of code.
5.6 Acumem SlowSpotter's Profiling Comparison

Comparing the profiling data collected from the Acumem SlowSpotter tool gives us the following results.

![Figure 10: Comparison of Fetch ratio for Optimized application](image)

Figure 10 compares the Fetch ratio of the LBM application before and after optimization. The image shows that the fetch ratio has been lowered significantly and minimized within the cache-memory region (4 MB in this case); starting at almost 11 percentage units at 512 KB, the improvement in the fetch ratio within the cache region is at least 8 percentage units at 4 MB. As clearly observable, the fetch ratio of the application remains stable throughout at 1.3%. Having a low fetch ratio helps speedup the application, as it avoids memory bottlenecks (especially in processors with low memory bandwidth interfaces) which cause memory latency, consequently causing the data to arrive late in the memory; instructions may stall for long, as they wait for the data to arrive in cache.
Figure 11 shows the improvement in miss ratio once the LBM application is optimized. There is an improvement of at least 8 percentage units in the cache region, beginning at 10 percentage units. The miss ratio for the optimized application diminishes and remains stable throughout at 0.9%. Something interesting seen from the miss ratio curves is that the size of the active dataset remains the same for both the optimized and the original application. However, the miss ratio for the optimized application is reduced significantly due to rearranging of the information in the dataset. Such low miss ratio is another reason for the application speedup, since the optimized application doesn't have to stall and wait (as much as the original one) for data to arrive in the cache on data requests made by the instructions.
Figure 12 compares the fetch utilization for the optimized application with the unoptimized version. The fetch utilization for the optimized version posts an improvement of at least 45 percentage units within the cache region. For the optimized application, the spatial usage of the data is always stable and consistent at 100%. It is not possible to achieve better utilization, since all the data being brought into the cache is always used before being evicted.

Keeping in mind that for this application we simply optimized the single most data (and computation) intensive loop to improve the runtime and throughput performance of the entire application, we now compare the loop statistics, before and after optimization, collected by Acumem SlowSpotter.

Figure 13 shows the Fetch and the Miss ratios of the unoptimized loop in the original LBM application. If corrected, the fetch ratio for the loop can be brought down to almost 5% within the cache region (4MB). Similarly, the miss ratio can be brought down too. Note that the Fetch and the Miss ratios follow a similar pattern with miss ratio slightly lower than the fetch all the time.

The Fetch and Miss Ratios for the optimized loop are shown in Figure 13.
Similarly, the fetch utilization (shown in Figure 14) can be improved as well. In the unoptimized loop, fetch utilization is below 25% within the cache region, reaching a maximum of 24% at the 2MB cache border.

The Fetch utilization for the optimized loop is shown in Figure 14; it is almost 40%. This is the percentage of data brought into the cache that is read and not written.
5.7 Throughput Scalability

We perform throughput scalability tests by running multiple instances of the LBM application simultaneously on equal number of cores. Each of the running application instances is bound to a predefined core. We perform throughput tests on two different processors.

Figure 15 shows the throughput comparison. The system used for generating the results is an Intel Xeon E5345 QuadCore processor running at 2.33 GHz, with each processor having L2 cache of 4MB, shared by every two processors.

![Figure 15: Relative throughput over multiple cores for Intel Xeon E5345](image)

Compared to the original LBM application, the optimized application scales well in terms of relative throughput. Running 4 simultaneous instances of both the applications (in separate runs) on a QuadCore shows that the optimized application performs much better (always at factor of almost 3X) in terms of memory bandwidth consumption when compared with the original one. When running 4 instances over 4 cores, the original application shows a slowing down trend, not achieving as much speedup, due to excessive memory bandwidth consumption. Compared to that, the optimized version keeps scaling pretty well. The rate of difference in relative throughput for the two applications keeps stable with added number of cores. The gain on one core is 3.3X, on four cores it stays almost comparable at 3X. Although, like the original version of the LBM, the optimized version shows a slowing down trend as well, but it has the capability
to scale further on a couple of added number of cores, without showing considerable performance degradation, due to its excellent memory access properties.

Running multiple instances of the two applications on a system with AMD Opteron 8384 shows a rather strange curve. The optimized application runs slower for a single instance but keeps gaining relative throughput with added number of cores and instances.

Figure 16 shows the throughput comparison when the two applications are deployed on an AMD Opteron 8384 QuadCore running at 2700 MHz with 512 KB of L2 cache per core and a 6MB L3 cache shared by all four on-chip cores.

![Figure 16: Relative throughput over multiple cores for AMD Opteron 8384](image)

For the above case of running the two applications on the AMD Opteron 8384, we make an observation that the original LBM application stops showing any improvement in relative throughput when running 4 instances on 4 cores. With added number of cores, and keeping hardware scalability consistent, this would eventually result in super-linear slowdown of the original application. On the other hand, we observe that the optimized application keeps scaling pretty well, posting even better throughput at 4 cores. This depicts a trend where the optimized application will continue to scale with added number of cores. This is essentially due to optimized and efficient data access behavior of the optimized application. The optimized application for a single instance shows a slight slowdown; this is probably because the blocking version of the code makes use of several other arrays besides the 'srcGrid' and the 'dstGrid' and moving them in and out of the cache may be one reason for this slight slowdown.
5.8 Parallelizing the LBM Application

Having successfully optimized the serial implementation of the LBM application and investigating the performance aspects of the optimized implementation, we go on to test the two versions of the application with parallelism enabled. We do so in order to make a reasonable comparison of how well the two versions would perform in terms of runtime speedup and scalability. Also, it is necessary to study the relationship between the relative throughput of the serial application and the runtime speedup of the parallel versions of the application, if there exists any. Doing so we can get the answer to how well would an optimized application scale after parallelism has been enabled, how can it be compared to its parallel unoptimized counterpart and if it really is better to parallelize the optimized application, as suggested by Sanjiv Shah [17].

As mentioned in the previous sections, the most data and computation intense code-section within the entire LBM application consists of a single function called 'LBM_performStreamCollide'. According to runtime statistics collected by GPROF this function runs for 99.8% of the total application runtime in the original LBM application, whereas for the optimized version it runs for 99.5% of total runtime. The benchmark LBM application provided by SPEC CPU2006 already has parallelism implemented, and just needs being turned on. In this implementation, the function most important to our study, 'LBM_performStreamCollide', is also completely parallelized.

The parallel implementation of this application is yet another example of an EP (embarissingly parallel) problem. Although, the application's worker threads work on shared data structures, there seems no need for synchronization amongst the threads, creating a perfectly parallel application. The communication ratio reported by Acumem ThreadSpotter for the entire application is none i.e. 0%, which is phenomenally low for a parallelized application. This small figure shows complete independence of the worker threads. Taking function runtime into consideration, this application is 100% parallel.

5.9 Performance Evaluation after Parallelizing

For performing a suitable performance evaluation of the parallel versions of the two applications, a criteria is set so that the results produced are intuitively comparable. We set the OpenMP environment variable 'OMP_SCHEDULE' to 'dynamic', thereby enabling dynamic resource allocation to tasks at runtime depending upon resource availability. Also, we disable worker thread count variation at runtime by setting 'OMP_DYNAMIC' to 'false'. Finally, we vary 'OMP_NUM_THREADS' from 1 to 4 in 4 different runs and record the runtime for measuring the speedup relative to serial original application runtime. In addition, we bind the application to equal number of processing cores as 'OMP_NUM_THREADS'.

45
Figure 17 shows the speedup of the parallelized original and optimized versions of the LBM application. The system used in this case is the Intel Xeon E5345 QuadCore processor running at 2.33 GHz, with each processor having L2 cache of 4MB, shared by every two processors.

Here, the relative speedup for the optimized application stays consistent at almost 3. However, when taking a close look at the gain factor from going onto an additional core, we observe that speedup-gain ratio for the optimized application is always considerably greater than the original one. Going from three to four cores we see a diminishing trend in the relative speedup gain for the original application, whereas the optimized version still seems to be gaining well. Going hypothetically from multicore to manycores in the current scenario, extrapolating this trend would keep showing a significant relative speedup for the optimized application. Also, something very interesting to note here is that the speedup achieved by the original application at four threads is lower than the serial version of the optimized application. In effect, indicating that you should consider optimizing the application before parallelizing it.

We study yet another evaluation for runtime speedup over a different platform. Figure 18 shows the speedup of the parallelized original and optimized versions of the LBM application. The system used here is AMD 8384 QuadCore processor running at 2700 MHz, with each core having private L2 cache of 512KB, and L3 cache of 6MB shared by all on-chip cores.
Unlike the previous case, the relative speedup of the optimized application keeps growing reaching to a maximum of 2X at four threads (scheduled over four cores). This shows a much better scalability trend in terms of parallel performance with added number of cores. Interpreting this trend for manycores indicates a significant relative speedup for the optimized application, where the original version hardly shows any improvement beyond 4 cores while the optimized application keeps gaining speedup. Notice the speedup gain for the original application when going from three to four cores; it hardly shows any improvement. So, even if the speedup gain for the optimized application beyond four threads is smaller than before, it will still post increasing relative speedup, since as expected from the curve, the original application wouldn't scale at all beyond four cores.

While working on this application, we realized a correlating relationship between the relative throughput and the relative parallel speedup of the two applications for at least two different processor architectures. Comparing figure 15 (relative throughput over multiple cores for Intel Xeon E5345 ) and figure 17 (Parallel speedup comparison over multiple cores, on Intel Xeon E5345 ) shows that the two curves are almost identical. Same is the case with AMD Opteron 8384 as seen in figure 16 and figure 18, supporting the thesis that the relative throughput for a given application over a given architecture is the upper bound/limit for the parallel speedup of that application over the same architecture; that is, the relative throughput curve roughly depicts the maximum achievable relative speedup for an application over a given architecture when it has been
perfectly parallelized. One may hardly find such an example where 100% of the runtime-application has been parallelized (or is parallelizable).

The LBM application studied here shows an extremely optimistic view of how well scientific applications may perform through adding very simple optimizations to the code and how these simple optimizations effect various properties of the application, such as relative speedup, speedup scalability, throughput scalability and parallel speedup. Also, this implementation is an example of embarrassing parallelism, with absolutely no thread communication amongst the cores. This application is perfectly parallelizable and good to study traits of parallel applications for parallel architectures, especially multicores. It was easier with this application to study the relationship between relative throughput and parallel speedup, since the application is optimizable and parallelizable up to a perfect 99.5%.
Chapter no.6

Case Study 2 – MCF
6.1 Introduction

The MCF application is an implementation of a graph optimization algorithm used to solve Minimum cost flows. The particular variant of the algorithm used in the MCF application is used to solve “Single depot vehicle scheduling problem in mass public transportation”. The network simplex algorithm implemented in the application is a specialized version of the Simplex algorithm for network flow problems.

Over the years since this application was originally developed at the “Konrad-Zuse-Zentrum Berlin (ZIB)”, several of its variants have been developed either due to updates or other reasons. In this case study we make use of the variant provided by the SPEC CPU2006 benchmark 429.mcf (successor version of the SPEC CPU2000 benchmark 181.mcf). The source code in this particular variant of the application has been changed to work mostly with simple integer and pointer arithmetic instead of performing complex mathematical calculations using the GCC math library. This has been done to improve performance and to dramatically speedup the application.

We further optimize it to show possible performance improvements, by removing memory bottlenecks, latency issues and by adding several other architecture-aware optimizations.

Further details about this application can be found here


This case study describes issues that have been solved to improve the application performance. Other issues have been ignored.

It is recommended to have the code for SPEC CPU2006 benchmark 429.MCF when looking into this case study.

6.2 Acumem SlowSpotter's Diagnosis

Profiling the mentioned version of the MCF application with the Acumem SlowSpotter tool, the report immediately points out at areas of possible high performance gains. SlowSpotter complains about Memory Bandwidth, Memory Latency and Data Locality problems and lists them in order of estimated performance gains, placing the code with the highest estimated gain at the top.

Figure 19 shows the “Bandwidth issues”, ranked by the percentage of fetches, in the issues section of the report
Figure 20 shows the loops in order of worst loop stats (or highest expected performance gains), in the loops section of the report.

Figure 20: Loops with highest expected gains (left pane) and loop in code (right pane)
Figure 21 shows the summary of the overall sampled application statistics.

<table>
<thead>
<tr>
<th>Global statistics</th>
<th>Miss/Fetch ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss ratio</td>
<td>13.3%</td>
</tr>
<tr>
<td>Fetch ratio</td>
<td>14.7%</td>
</tr>
<tr>
<td>Writeback ratio</td>
<td>2.2%</td>
</tr>
<tr>
<td>Upgrade ratio</td>
<td>0.0%</td>
</tr>
<tr>
<td>Communication ratio</td>
<td>0.0%</td>
</tr>
<tr>
<td>Fetch utilization</td>
<td>26.3%</td>
</tr>
<tr>
<td>Write-back utilization</td>
<td>17.5%</td>
</tr>
<tr>
<td>Communication utilization</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analysis parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor model</td>
<td>Intel(R) Pentium(R) D CPU 3.20GHz (auto)</td>
</tr>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of caches</td>
<td>2</td>
</tr>
<tr>
<td>Cache level</td>
<td>2</td>
</tr>
<tr>
<td>Cache size</td>
<td>2M</td>
</tr>
<tr>
<td>Line size</td>
<td>64</td>
</tr>
<tr>
<td>Replacement policy</td>
<td>random</td>
</tr>
</tbody>
</table>

Figure 21: Summary of Application statistics
Acumem SlowSpotter's diagnosis immediately points out at the following major problems. We number the problems to make the statements with issues easily identifiable. The two problem statements here are named “Problem 1” and “Problem 2” and will be identifiable through these names in this case study.

In the following loop (in function 'primal_bea_mpp')

```c
NEXT:
    /* price next group */
    arc = arcs + group_pos;
    for( ; arc < stop_arcs; arc += nr_group )
    {
        if( arc->ident > BASIC ) PROBLEM 1
        {
            /* red_cost = bea_compute_red_cost( arc ); */
            red_cost = arc->cost - arc->tail->potential + arc->head->potential; PROBLEM 2
            if( bea_is_dual_infeasible( arc, red_cost ) )
            {
                basket_size++;
                perm[basket_size]->a = arc;
                perm[basket_size]->cost = red_cost;
                perm[basket_size]->abs_cost = ABS(red_cost);
            }
        }
    }
    if( ++group_pos == nr_group )
        group_pos = 0;
    if( basket_size < B && group_pos != old_group_pos )
        goto NEXT;
```

**Problem 1:**
The report shows problem 1’s fetches to be 30.8% of all the memory fetches in this application, with a fetch utilization of 43.1% in the first highlighted statement (see Figure 1). The instruction stats show the misses to be 34% of all the cache misses in the application, and fetch and miss ratio at 21.2%. Reducing the fetch and miss ratio would greatly help improve bandwidth issues. Also, hardware prefetch probability here is 0%.

**Why Problem 1?**
The structure pointed to by the pointer 'arc' is of size 64 bytes, whereas the statement pointed out in Problem 1 uses only the member 'ident'; only 8 bytes of the entire structure. Few other members (tail, head and cost) of the structure may come into use if the statement is true. Higher probability of failing the 'if' test results in lower utilization of the data that has already been pulled into the cache. Even passing the 'if' test doesn't result in good utilization of the structure's data. Moreover, pulling the data into the cache from the main memory uses memory bandwidth and low utilization of such data results in effective waste of precious memory bandwidth. In addition to this, since the structures in the array 'arcs' are accessed in strides of factor of 'nr_group', the hardware prefetch utility
is unable to correctly guess the next structure which leads to its complete failure and a hardware prefetch probability of 0%.

Problem 2:
The report points out at poor fetch utilization for the second highlighted statement. Having an identical miss and fetch ratio of 15.4%; it has an extremely poor fetch utilization of only 12.8%. This statement too has a hardware fetch probability of 0%.

Why Problem 2?
The miss and fetch ratio are due to the 'tail' and 'head' component pointers of the 'arc' structure. These pointers point to structures of another type ('node_t'), having size of 112 bytes each. Every time 'potential' (8 bytes only) is accessed in 'head' and/or 'tail', it pulls in 56 bytes of unwanted data into the cache, for each structure. This results in extremely low utilization of the data pulled into the cache and a lot of wasted memory bandwidth. The miss ratio and fetch ratio are identical because the statement suffers a cache miss every time 'head' and 'tail' structures are accessed (due to randomness in data access), and the data has to be fetched from the main memory every time. Also, since this is a general case of the randomness problem (data accessed is not located contagiously, is never reused and/or has no predictable access pattern), the hardware prefetching fails completely for this statement too.

6.3 Acumem SlowSpotter's Advice

Problem 1:
Acumem SlowSpotter advises
1. correction of inefficient loop nesting, and
2. temporal / spatial blocking, identifying the need to reuse data and pointing out the problem of data access after its eviction from the cache.
3. improvement in fetch utilization

Problem 2:
Acumem SlowSpotter advises
1. avoiding randomness in data accesses (of 'head' and 'tail' structures pointed-to by 'arc')
2. temporal/spatial blocking for data reuse and identifying the problem of required data's immature cache eviction
3. improvement in fetch utilization

6.4 Steps for Optimization

Note: Please refer to code in Figure 22 to see changes made in every point

1. To improve fetch utilization, the structure 'arc' is broken into two parts such that part of the data that is not used frequently is shifted to another structure (structure for 'cold' data) and a pointer in the 'arc' structure points to the rest of the 'cold' data that
should actually belong to 'arc' itself.

Originally, the structure 'arc' is of type 'arc_t' which is defined as follows

```c
struct arc
{
    int ident;
    cost_t cost;
    node_p tail, head;
    flow_t flow;
    cost_t org_cost;
    arc_p nextout, nextin;
}arc_t;
```

We break the above structure into the following

```c
typedef struct arc_data
{
    arc_p nextout, nextin;
}arc_data_t;

struct arc
{
    int ident;
    cost_t cost;
    node_p tail, head;
    flow_t flow;
    cost_t org_cost;
    arc_data_t *arc_data_ptr;
};
```

Although the data moved is not much, but it still shows some improvement in data intensive loops. Moreover, this approach is quite general and can be used in other scenarios to split up a data structure into significantly smaller parts.

2. According to our analysis, the stride of 'nr_group' used to access the arc structures is virtually meaningless as it just divides the whole of the 'arc' pool into a number of 'K' candidate lists, with each member of a candidate list separated by 'nr_group' number of structures in between them. We alter the 'arc' structures access pattern to a sequential one (by reducing the stride to 1 instead of 'nr_group'), where all the structures in a candidate list are contiguous in memory. This helps the hardware prefetch utility to accurately guess the next structure to be brought into the cache, improving spatially located data's reuse as well. However, this does effect the runtime, but doesn't produce any errors in the solution.

3. Since accessing and testing 'ident' in 'arc', causes the entire 'arc' structure to be pulled into the cache and remain unused if the test fails, one way to improve fetch ratio and utilization would be to vectorize the 'ident' variable of all the 'arc' structures into an array. We do this by “correctly” replicating 'ident' for each 'arc' structure in the 'arcs' pool into an array 'arc_id'. This way, the utilization will improve for the data being pulled into the cache, since the structure 'arc' will be accessed only when the test
passes. This would dramatically improve the fetch and miss ratios for “Problem 1” involving the 'if' test.

4. In Problem 2, we have already discussed that a lot of bandwidth is wasted when accessing 'head' and 'tail' for each 'arc' since only 'potential' is used from the entire 'head' or 'tail' structure's data. Vectorizing 'potential' (like 'ident' in step 3) will improve memory bandwidth usage and fetch utilization. Also, since the cache is large enough to hold the entire array, fetch and miss ratios should be minimized by doing so.

5. The final step in the Problem 2's optimization should be the vectorizing of the 'head' and 'tail' arcs. This will relieve the logic of the above loop of working with the structures all the time and avoid thrashing the cache by pulling in 'arc' structures all the time. Vectorizing of these two (head and tail) structures can again be done by carefully replicating them in an array. Doing so will maximize the fetch utilization and at the same time minimize the fetch and miss ratios.

Having made the above recommended changes, this is how the new code looks

```c
arc = arcs + group_pos;

NEXT:
    /* price next group */
    for (stop = arc + nr_group, i = group_pos; arc < stop && arc < stop arcs, arc++, i++)
    { //
        if (arc_id[i] /arc->ident*/ > BA3[C]
            { //
                tmp_index = ((node_p) arc_tail[i]) - nodes,
                tmp_index = ((node_p) arc_head[i]) - nodes,
                red_cost = arc_cost[i] - node potential[truf_index] + node potential[tmp_index],
                if ((red_cost < 0 && arc_id[i] == AT_LOWER)
                || (red_cost > 0 && arc_id[i] == AT_UPPER))
                { //
                    basket_size++,
                    perm[basket_size]->a = arc,
                    perm[basket_size]->cost = red_cost,
                    perm[basket_size]->abs_cost = AB_S(red_cost),
                }
            }
    }

    group_pos += nr_group;

    if (arc == stop arcs)
        group_pos = 0,

    if (basket_size < B && group_pos != old_group_pos)
        goto NEXT;

Figure 22: Optimized code
```
The above improvements in the code are bound to produce effective results which should be clearly visible in application's run-time, since profiling the application reveals that the function containing the above code runs for 60% of the total application's duration. Lowering runtime for this piece of code means lowering of the entire application's runtime.

6.5 Performance Evaluation

Changing the access pattern of the arcs in the above optimized loop (mentioned in optimization step 2) to sequential rather than with a factor of stride 'nr_group' alters the execution of the program along with its runtime. With this change, the program runs 1 million simplex iterations in excess of the original version. To get an accurate comparison of the application before and after the optimization, we induce the same change in the original application i.e. reduce the stride1 to make 'arc' access sequential. Although this does improve the performance due to better hardware prefetch probability, we assume no change in program properties.

On an Intel Core2 Quad (64-bit Xeon processors) running at 2.33 GHZ with L2 cache of 4MB, running the original MCF application takes 1199 seconds. The time taken to perform exactly 7 million simplex iterations (measuring only unoptimized code function time) takes 627 seconds.

Whereas, running MCF application with the optimized code takes just 758 seconds, which is almost 450 seconds faster than the original application. The time taken for this application to perform exactly 7 million iterations (measuring only optimized code function time) is 261 seconds.

Figure 23 shows the performance gains in application speedup and Simplex iterations.

Measuring the speedup of the application after optimization, we see that the optimized application runs almost twice as fast as the original one at 1.9X. Whereas, the function containing the optimized loop runs almost three times as fast as the same function from the original application, at 2.9X.
Profiling the two applications with the GPROF tool shows that in the original version of the application, the function ('primal_bea_mpp') with the unoptimized code ran 60% of the total application's runtime. Whereas, for the optimized code, the same function runs for only 40% of the entire application's runtime; runtime figures from GPROF show this function's total runtime is lowered by 56.25%, thereby lowering the entire application's runtime.

All of this speed-up is the result of the optimization steps listed in the previous sections, focusing mostly on a single loop of code.
6.6 Acumem SlowSpotter's Profiling Comparison

Comparing the profiling data collected from the Acumem SlowSpotter tool gives us the following results.

Figure 24: Comparison of Fetch ratio for Optimized application

Figure 24 compares the Fetch ratio of the MCF application before and after optimization. The image shows that the fetch ratio has been lowered considerably within the cache-memory region (4 MB in this case); starting at 8 percentage units at 512 KB, the improvement in the fetch ratio within the cache region is at least 5 percentage units at 4 MB.
Figure 25 shows the improvement in the miss ratio once the MCF application is optimized. There is an improvement of at least 8 percentage units in the cache region, beginning at 11 percentage units. The miss ratio for the optimized application diminishes completely once the data size reaches 256 MB. So, for the optimized application the active dataset could be said to be about 256 MB. The maximum active dataset for the original MCF application is 1700 MB (for 64 bit version). The above curve shows that in the optimized application the size of the active dataset has been reduced almost seven times. Smaller data set in the optimized application helps reduce the miss ratio, thereby lowering stalls in the application caused by misses, making the application run faster.
Figure 26 compares the fetch utilization for the optimized application with the unoptimized version. The fetch utilization for the optimized version posts an improvement of at least 30 percentage units within the cache region. About twice the amount of the data brought to the cache is now used before eviction.

Keeping in mind that for this application we simply optimized the single most data (and computation) intensive loop to improve the runtime and throughput performance of the entire application, we now compare the loop statistics, before and after optimization, collected by Acumem SlowSpotter.

Figure 27 shows the Fetch and the Miss ratios of the unoptimized loop in the original MCF application. If corrected, the fetch ratio for the loop can be brought down to 5% within the cache region (4MB). Similarly, the miss ratio can be brought down too. Note that the Fetch and the Miss ratios overlap and are identical in the following figure.

The Fetch and Miss Ratios for the optimized loop are shown in Figure 27. We clearly get very close to the predicted improvement.
Similarly, the fetch utilization (shown in figure 28) can be improved as well. In the unoptimized loop, fetch utilization is below 40% within the cache region.

The Fetch utilization for the optimized loop is shown in Figure 28; it is almost 100%.

**Figure 28: Fetch utilization for unoptimized loop(above), optimized loop(below)**
6.7 Throughput Scalability

Figure 29 shows the throughput comparison. The system used for generating the results is an AMD Opteron Barcelona QuadCore (dual socket) processor with each processor having L2 cache of 512KB and L3 cache of 2MB, shared by every two processors.

Figure 29: Relative throughput over multiple cores for AMD Opteron Barcelona

Compared with the original MCF application, the optimized application is more scalable in terms of relative throughput. Running 4 simultaneous instances of both the applications (in separate runs) on a QuadCore shows that the optimized application performs much better (in excess of factor 1.6) in terms of memory bandwidth consumption when compared with the original one. When running 8 instances over 8 cores, the original application shows a dramatic slowdown due to excessive memory bandwidth consumption. Compared to that, the optimized version shows better performance. The rate of difference in relative throughput for the two applications keeps growing with added number of cores. The gain on one core is 1.5, on four cores it grows to almost 1.7 and on 8 cores it becomes almost 1.8. This comparison proves excellent scalability properties once the application has been optimized.

Figure 30 shows the throughput comparison (for up to 3 cores) when the two applications are run on an Intel Xeon E5345 QuadCore (running at 2.33GHz with a 4MB L2 cache shared by every two processors). The case for 4 cores is not included due to unavailability of required memory for running 4 simultaneous instances of the application.
Running multiple instances of the two applications on a system with better resources reveals that the original MCF application performs much worse (than expected from the curve in Figure 29) compared to the optimized application, in terms of throughput scalability. This clearly points out at inefficient memory bandwidth consumption of the original.

### 6.8 Parallelizing the MCF Application

Having successfully optimized the serial implementation of the MCF application and investigating the performance aspects of the optimized implementation, we go on to parallelize the MCF application. Now which version do we parallelize? “Both of them!” is the correct answer. By doing so we can practically get the answer to the very exciting thought of how well would an optimized application scale after parallelizing, compared to parallelizing its unoptimized counterpart and whether it would really be better to parallelize the optimized application. In doing so we have to make sure that both the applications are parallelized using identical methodology. We use OpenMP to parallelize the two versions of the MCF application.

We saw in the previous sections that the most data and computation intense code-section within the entire MCF application consists of a single nested-loop (in the function ‘primal_bea_mpp’). It consumes almost 60% of runtime in the original MCF application, whereas 40% of the entire runtime after it has been optimized. This code-section within the entire application is the most suitable area to implement the parallelism; parallelizing any other part of the application is not straightforward and requires major changes in the application design.

![Figure 30: Relative throughput over multiple cores for Intel Xeon E5345](image-url)
A preferable approach to parallelizing the application would be to perform the simplex iterations in parallel, but since this seems hard with the code structure of the MCF application, we try approaching the same effect by increasing the size of the sequential candidate list, the program goes through while performing simplex iterations. We increase the size of the sequential candidate list that is processed in the optimized loop, according to the number of threads, in an effort to reduce the total number of simplex iterations performed throughout the life of the application and to maintain task balance for the worker threads. So as the number of processing cores grow for the parallel application the total simplex iterations lower. Not doing so results in super-linear slowdown (below 1X) of the application beyond 2 cores, due to the growing overhead of thread creation and management as a result of higher simplex iterations.

This is how the parallelized code looks like in the original application

```c
NEXT:
   /* price next group */
   bs_count = 0;
   if( (stop_arcs - (arcs + group_pos)) > nr_group)
      stop_index = group_pos + nr_group;
   else
      stop_index = stop_arcs - arcs;
   chunk = ceil((stop_index-group_pos)/omp_get_num_threads());
   chunk = ceil(chunk / omp_get_num_threads());

   #pragma omp parallel private (arc, red_cost) 
      firstprivate (chunk, arcs, stop_index, group_pos)
      reduction(+:bs_count)
   {
      #pragma omp for schedule (dynamic, chunk) nowait
      for( i = group_pos ; i < stop_index; i++ )
      {
         arc = arcs + i;
         if( arc->ident > BASIC )
            {
               /* red_cost = bea_compute_red_cost( arc ); */
               red_cost = arc->cost - arc->tail->potential + arc->head->potential;
               if( bea_is_dual_infeasible( arc, red_cost ) )
               {
                  temp_basket_size++;
                  temp_perm[temp_basket_size]->_a = (arcs + i);
                  temp_perm[temp_basket_size]->_cost = red_cost;
                  temp_perm[temp_basket_size]->_abs_cost = ABS(red_cost);
               }
            }
      } /* for loop */
      bs_count += temp_basket_size;
   }

   group_pos += nr_group;

   if( (arcs + group_pos) >= stop_arcs)
      group_pos = 0;
```
if( bs_count < B && group_pos != old_group_pos )
    goto NEXT;

The buffer “temp_perm” and the variable “temp_basket_size” are private to each thread and are manageable only through their owner threads. This has been done to avoid sharing of data structures amongst the threads. The parallel solution implemented here is truly an EP (embarrassingly parallel) solution and the threads are never synchronized. However, the program in the function 'primal_bea_mpp' moves from serial to parallel regions three times for every single iteration.

This is how the parallelized code looks in the optimized application

NEXT:
    /* price next group */
    bs_count = 0;
    if( (stop_arcs - (arcs + group_pos)) > nr_group)
        stop_index = group_pos + nr_group;
    else
        stop_index = stop_arcs - arcs;

    chunk = ceil( (stop_index-group_pos)/omp_get_num_threads() );
    chunk = ceil(chunk / omp_get_num_threads());

    #pragma omp parallel private (tmp_tindex, tmp_hindex, red_cost) \
        firstprivate (chunk, arcs, nodes, stop_index, arc_head, arc_tail,\
            arc_id, arc_cost, node_potential, group_pos)\n        reduction(+:bs_count)
    {
        #pragma omp for schedule (dynamic, chunk) nowait
        for(i = group_pos ; i < stop_index; i++ )
        {
            if( arc_id[i] > BASIC )
            {
                tmp_tindex = ((node_p) arc_tail[i]) - nodes;
                tmp_hindex = ((node_p) arc_head[i]) - nodes;
                red_cost = arc_cost[i] - node_potential[tmp_tindex] + 
                            node_potential[tmp_hindex];
                if( (red_cost < 0 && & arc_id[i] == AT_LOWER) \n                    || (red_cost > 0 && arc_id[i] == AT_UPPER) )
                {
                    temp_basket_size++;
                    temp_perm[temp_basket_size]->a = (arcs + i);
                    temp_perm[temp_basket_size]->cost = red_cost;
                    temp_perm[temp_basket_size]->abs_cost = ABS(red_cost);
                }
            }
        }
        bs_count += temp_basket_size;
    }

    group_pos += nr_group;
if((arcs + group_pos) >= stop_arcs)
    group_pos = 0;

if( (bs_count) < B && group_pos != old_group_pos )
    goto NEXT;

Both of the parallel versions produce identical runs provided the for loop is scheduled as 'static'. Dynamic scheduling, however, for practical reasons is the best scheduling choice for better speedup. The results produced and discussed here are the best of recorded test runs.

The communication ratio reported by Acumem ThreadSpotter for the entire application is just 0.6% which is considerable amount of communication amongst the caches. This is due to dynamic scheduling of the worker threads. Since each of the worker thread has its own set of data which hardly changes, dynamic scheduling may cause threads to be scheduled on cores different from where they were scheduled previously, with separate caches, thereby causing the threads’ data to be communicated to the cache where the thread is executing.

6.9 Performance Evaluation after Parallelizing

Having altered the buffer size for processing the sequential candidate list changes the runtime behavior of both the serial MCF versions speeding up the optimized application by factor of 1.5.

Figure 31 shows the speedup of the parallelized original and optimized versions of the MCF application. The greatest speedup posted by the two applications is when they are run over two cores. The optimized application runs at 2.4X the original serial MCF application. Whereas the original parallel MCF application runs at a perfect 1.6X, compared to its serial version; keeping in mind that for the original MCF application, the function “primal_bea_mpp” (the one parallelized) runs for 60% of the total application's runtime.
However, both the optimized and the original applications slow down once parallelized for more than two cores. This is primarily due to increase in system time, due to greater overhead of thread management and not much decrease in the application's processing time. However, the optimized MCF application still posts better results when compared with its original counterpart for any given number of cores.

The correlation seen in the slowdown of both applications with the growing number of cores (beyond two) translates to an inherent problem in the application marking inability of the application design to adapt to parallelism greater than two worker threads. This might be due to the fact that the original application was not necessarily designed with parallel execution in mind, suggesting that not all applications can adjust well to parallel execution once they have been written. Although in this example, for the original MCF we have parallelized the function executing 60% of the total application runtime, with thread communication of 0.6% for the entire application (as reported by Acumem ThreadSpotter), still this doesn't result in good scalability of the application in terms of parallelism. Same goes with the optimized MCF, though the function 'primal_bea_mpp' here runs only 40% of the total application runtime.

However the results posted here very strongly suggest that parallelizing of an optimized application will always show better speedup compared to parallelizing an unoptimized serial application. The results shown here also strongly suggest that not every application can adapt well to parallel execution and show better runtime performance and speedup.
Chapter no.7

Case Study 3 – CIGAR
7.1 Introduction

The CIGAR application is an implementation combining genetic algorithms and case-based reasoning for the purpose of approaching solutions to problems much quickly. The application relies on a novel optimized machine learning technique, in which the system acquires knowledge by generating cases on its own, and approaches the correct solution based on the acquired knowledge from recently solved problem cases, through case-based reasoning.

The particular CIGAR implementation under consideration for this case study is developed by the ECSL (Evolutionary Computing Systems Lab) at University of Nevada, Reno. The CIGAR module studied is being used in many projects at the mentioned institution such as computer strategy games and evolving believable agents.

We add optimizations to the code to show possible improvements in performance of this application. Throughout our case study we will always uniformly consider population size of 50,000, “Closest to best” as the Injection strategy and “Hamming distance” as the distance metric.

Details about the application as provided by University of Nevada, Reno can be found here

http://ecsl.cse.unr.edu/

The CIGAR code can be downloaded from

http://www.cse.unr.edu/~sushil/class/gas/code/index.html

This case study discusses the issues that were actually solved to demonstrate improvements in application performance. Other issues have been ignored.

It is recommended to have the code for CIGAR when looking into this case study.

7.2 Acumem SlowSpotter's Diagnosis

Having profiled the CIGAR application with the Acumem SlowSpotter tool, the report immediately points out at areas of possible high performance gains. SlowSpotter complains about Memory Bandwidth, Memory Latency and Data Locality problems related with this application and lists them in order of estimated performance gains, placing the code region with the highest estimated gain at the top.

Figure 32 shows the “Bandwidth issues”, ranked by the percentage of fetches, in the issues section of the report
### Figure 32: Bandwidth Issues (left pane) and problem areas in code (right pane)

Figure 33 shows the loops in order of worst loop stats (or highest expected performance gains), in the loops section of the report.
Figure 34 shows the summary of the overall sampled CIGAR application

<table>
<thead>
<tr>
<th>Issues</th>
<th>Loops</th>
<th>Summary</th>
<th>Files</th>
<th>Execution</th>
<th>About/Help</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global statistics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miss ratio</td>
<td>11.1%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch ratio</td>
<td>16.2%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Writeback ratio</td>
<td>0.1%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Upgrade ratio</td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication ratio</td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch utilization</td>
<td>17.4%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-back utilization</td>
<td>22.1%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication utilization</td>
<td>100.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Analysis parameters</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor model</td>
<td>Intel(R) Pentium(R) D CPU 3.20GHz (auto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of CPUs</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of caches</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache level</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache size</td>
<td>2M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line size</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replacement policy</td>
<td>random</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 34: Summary of Application statistics**

The statistics for the CIGAR application show overall fetch ratio at 16.2%, miss ratio at 11.1% and fetch utilization of just 17.4% within the cache region of 2MB.
Acumem SlowSpotter's diagnosis immediately points out at the following major problems. Each statement is marked as a part of a problem. All problems are referred to with their number in the case study.

In the following loop (in function 'FindNBest')

```c
for(current = 0; current < howmany - 1 ; current++) {
    max = current;

    /* Find Next best */
    for(i = current + 1; i < size ; i++) {
        if((p + rank[i])->fitness > (p + rank[max])->fitness) {
            PROBLEM 1
            max = i;
        }
    }

    SwapInt(&rank[current], &rank[max]);
}
```

and in the following loop (in function 'Roulette')

```c
if(p->scaleFactor > 0.0) {
    rand = FRandom() * p->scaledSumFitness;
    j = -1;
    do{
        j++;
        partsum += pop[j].scaledFitness;
    } while (partsum < rand && j < popsize - 1) ;

    return j;
} else {
    /* Skipping code here */
}
```

Problem 1:
The report points to a very high accumulated percentage of fetches to be 98.2% of all the memory fetches in this application, with a cumulative fetch utilization of only 12.5% for the entire 'if' statement involving two memory reads (both memory read accesses have individual fetch utilization of 12.5%), in the first problem (see Figure 28). The instruction stats show the percentage of misses to be 54.6% of all the cache misses in this application, fetch ratio at an average of 17% and miss ratio at 12%, for the entire 'if' statement. Reducing the fetch and miss ratio for this statement would greatly help improve bandwidth issues. Also, hardware prefetch probability here is relatively low and averages at 30% for the whole loop.

Why Problem 1?
The main reason for such high fetch and miss ratios is the inappropriate memory accesses made for the acquisition of the required data (the 'fitness' property); extremely low utilization of the data available in the cache happens for the same reason. The loop
iterating repeatedly between '0' to 'size' (with starting point incrementing by a factor of '1'
every iteration in the outer loop), makes a comparison of the 'fitness' attribute of two
structures, of datatype 'INDIVIDUAL', from the structure pool. The structures being
accessed in this loop are arranged consecutively in memory just like an array of
structures. However, each structure is sized 96 bytes, of which only 8 bytes ('fitness'
attribute) are used in the given loop. Every time the index 'i' is incremented and a new
structure is pointed to access its 'fitness' attribute, at least 64 bytes of consecutive data
from the structure (and probably a part of the next structure in memory, depending upon
how the structure being accessed is 64-byte aligned relative in memory) being referred
need being brought into the cache, to fill in an entire cache-line; cache-line size in our
case is 64 bytes. Of these 64 bytes pulled into the cache, only 8 bytes of 'fitness' are ever
used resulting in extremely low fetch utilization. This is visible when we look at the fetch
utilization for the memory read accesses in the 'if' statement; both of these read
instructions post the fetch utilization to be 12.5% (8 of 64 bytes). In cases where the
variable 'size' is large enough, as the iteration approaches close to 'size', data from earlier
iterations (iterations closer to 'current+1') will be evicted from the cache, since every
access to a new structure (indexed by increasing 'rank[i]') needs to pull in a new cache-
line of data. Having completed the inner loop once, the outer-loop increments 'current' by
one and returns to the inner loop to access mostly the same data again, causing most of it
to be re-fetched in to the cache. This causes the high percentage of cache misses and
consequently very high fetch percentage and fetch/miss ratios. This pattern is repeated
until the outer loop ends, causing a majority of similar data accesses (by the inner loop)
between immediate iterations of the outer loop.

Problem 2:
Of the rest of the program (not including problem 1), the loop pointed out in this problem
causes 50% of the remaining misses and fetches. The fetch ratio for this loop is almost
33%, whereas the miss ratio is at a high of 21%. Hardware prefetch is down at 37%, only
slightly better than that of problem 1. Also, fetch utilization is extremely low at 12.5%.

Why Problem 2?
The issue in the loop pointed out in this problem is similar to that in problem 1. The only
difference is the attribute being accessed; in this case it is 'scaledFitness'. Also unlike the
loop in problem 1, we don't re-fetch previously fetched data over and over again, as we
go through the array of structures in this case just once. However, this doesn't change the
situation much since we have to fetch a new cache-line of data (64 bytes) for every
structure, while we try to access and use its 'scaledFitness' attribute. The fetch utilization
for this loop, of almost 12.5% shows that exactly only 8 bytes of the entire cache-line are
ever used, the rest is wasted data which is eventually evicted from the cache without
being touched. Since the prefetched data is not useful in this loop and the program suffers
a cache-miss whenever accessing a new structure here, the miss ratio and consequently
the fetch ratio rises dramatically. High miss ratio causes stalls in this loop frequently,
thereby increasing the runtime of this loop.
7.3 Acumem SlowSpotter's Advice

Problem 1:
Acumem SlowSpotter advises
1. Spatial blocking, identifying the need to reuse data and pointing out the problem of data access after its eviction from the cache
2. improvement in fetch utilization

Problem 2:
Acumem SlowSpotter advises
1. Spatial blocking for data reuse and identifying the problem of required data's immature cache eviction
2. improvement in fetch utilization

7.4 Steps for Optimization

Note: Please refer to code in Figure 35 to see changes made in every point

1. To improve the fetch utilization for loop in problem 1 and to avoid fetching unnecessary data into the cache, we replicate the required data from the array of structures (pointed to by 'p->oldpop' in the main scope) into a separate array containing only the relevant data needed, from each structure, in the function 'FindNBest'. For every member structure of the array pool pointed-to by 'p->oldpop' in function 'main', we copy the attribute 'fitness' to the corresponding location (index) in an array 'opop_fitness'. 'opop_fitness' is an array explicitly created for maintaining updated 'fitness' values for each structure in 'p->oldpop'. Changes are consistently and carefully maintained in other parts of the program, copying the 'fitness' attribute into the corresponding entry of 'opop_fitness' array, whenever and wherever it is changed in the original structures. Having done so, we replace the code in function 'FindNBest', accessing structures for retrieving 'fitness' attribute with a simple array access, by accessing corresponding 'fitness' values from the 'opop_fitness' array. This would avoid fetching data not needed during the execution of the loop, also avoiding stalls from cache misses every time a new structure is accessed to retrieve its 'fitness'. Also, it would be easier to fit the whole array 'opop_fitness' within the cache region, avoiding re-fetching of data fetched in previous iterations of the outer loop.

2. Since the issue pointed in problem 2 is similar in nature when compared with problem 1, we apply the same procedure as in step 1 to improve the fetch utilization and to reduce the high fetch and miss ratios for the loop. We create an array 'opop_scaled_fitness' to replicate and hold updated values of the attribute 'scaledFitness' for all the structures in the pool pointed to by 'p->oldpop' in the main function's scope. Just as in step 1, consistency in values of structures' 'scaledFitness' attributes and their corresponding values in array 'opop_scaled_fitness' is maintained carefully in all other parts of the program. In the function 'Roulette' we replace the structure access of the 'scaledFitness' attribute with simple array access, by accessing corresponding 'scaledFitness' values from the 'opop_scaled_fitness' array.
Having made the recommended changes, this is how the new code in function 'FindNBest' looks like

```c
for(current = 0; current < howmany - 1; current++) {
    max = current;

    /* Find Next best */
    for(i = current + 1, i < size, i++) {
        if(opop_fitness[rank[i]] > opop_fitness[rank[max]]) {
            max = i;
        }
    }
    Swapint(&rank[current], &rank[max]);
}
```

Code excerpt mentioned in problem 2, in function 'Roulette', looks like this

```c
if(p->scaleFactor > 0.0) {
    rand = FRandom() * p->scaledSumFitness,
    j = .1,
    do{
        j++;
        partsum += opop_scaled_fitness[j];
    } while(partsum < rand && j < psize - 1);
    return j;
} else {
    /*Skipping code here*/
}
```

**Figure 35: Optimized code**

Note how the structure access is replaced by array access in both problems with same effect. Figure 36 shows the advised data replication procedure.

**Figure 36: Required data isolation through replication**
In effect, replacing the structure access with sequential access of array is equivalent to removing a stride of a “cache line”. Also, in accessing an array in the given situation, a cache-miss would necessarily be followed by at least 7 consecutive cache-hits, effectively lowering the miss/fetch ratios and the stalls suffered at such points. Moreover, even such mandatory cache-misses are avoided in current scenario due to hardware prefetch utility being able to accurately guess the next piece of data-to-be-fetched, resulting in no cache-misses and hence no stalls. Sequential access to the arrays will also guarantee an eight-fold increase in the fetch utilization, going from 12.5% to 100% for the given loops.

The above improvements in the code are bound to produce effective results which should be clearly visible in application's run-time and relative throughput speedup, since profiling the application reveals that the two functions containing the above code run for almost 99.9% of the total application's duration. Lowering runtime for this piece of code means lowering of the entire application's runtime.

### 7.5 Performance Evaluation

We evaluate the performance of the optimized and the original CIGAR applications using more than a single platform in an attempt to get a realistic view of how the two applications would behave in the real world on different platforms they are deployed on.

On an Intel QuadCore E5345 (64-bit Xeon processors) running at 2.33 GHZ with L2 cache of 4MB, running the original single-threaded CIGAR application takes 2063 seconds. Whereas, running CIGAR application with the optimized code takes just 205 seconds, which is more than 1800 seconds faster than the original application. These results show a runtime decrease of almost 90% for the entire application.

Measuring the speedup of the application after optimization, we see that the optimized application runs several times as fast as the original one at 10X. Imitating the speedup behavior of the entire application, the optimized functions run more than ten times as fast as the unoptimized functions in the original application, at 12X. Data for optimized functions has been gathered with help of GPROF in this case study.

Figure 37 shows the performance gains in application speedup and optimized functions.
Profiling the two applications with the GPROF tool shows that in the original version of the application, the two functions ('FindNBest' and 'Roulette') with the unoptimized code run 99.9% of the total application's runtime. Whereas, for the optimized code, the same two functions run for 96% of the entire application's runtime. A major drop in runtime is seen in the function ‘FindNBest’, which runs 72% of the total application runtime after optimization; before optimization it ran for 92% of the total runtime. This, from our reasoning, is primarily due to reduction in stalls from cache-misses in the optimized loops. Runtime figures from GPROF show the total accumulated runtime for both the optimized functions is lowered by 91%, thereby significantly lowering the entire application's runtime.

All of this speed-up is the result of the optimization steps listed in the previous sections, focusing mostly on two data-intensive loops of code in the entire application.

7.6 Acumem SlowSpotter's Profiling Comparison

Comparing the profiling data collected from the Acumem SlowSpotter tool gives us the following results
Figure 38 compares the Fetch ratio of the CIGAR application before and after optimization. The curve shows that the fetch ratio has been lowered significantly and minimized to 0% within the cache-memory region (2 MB in this case); starting at almost 26 percentage units at 512 KB, the improvement in the fetch ratio within the cache region is at least 15 percentage units. As clearly observable, the fetch ratio of the application remains stable throughout at 0% from the cache border and onwards. Having such low fetch ratio helps speedup the application significantly, as it avoids memory bottlenecks (especially in processors with low memory bandwidth interfaces) which cause memory latency, consequently causing requested data to arrive late in the memory; instructions may stall for long, as they wait for the data to arrive in cache. Application's overall fetch ratio is the property that affects the relative throughput speedup; lower overall fetch ratio results in significant improvement in relative throughput gain.
Figure 39 shows the improvement in miss ratio once the CIGAR application is optimized. There is an improvement of at least 10 percentage units within the cache region, beginning at 20 percentage units. The miss ratio for the optimized application diminishes and remains stable throughout at 0%. The miss ratio curve shows that the active dataset has shrunk 8 times for the optimized application, compared to the original application. Such low miss ratio is another reason for the application speedup, since the optimized application doesn't have to stall and wait (as much as the original one) for data to arrive in the cache due to cache-misses when data requests are made by the instructions.
Figure 40 compares the fetch utilization for the optimized application with the unoptimized version. The fetch utilization for the optimized version posts an improvement of at least 40 percentage units within the cache region. For the optimized parts of the code, the spatial usage of the data is almost always stable and consistent at 90%. The curve above might be slightly misleading as it represents the utilization of the data for the entire application. Still the optimized application always posts considerably much better fetch utilization, as the optimized application uses twice as much data from the cache before its eviction.

Keeping in mind that for this application we simply optimized the two most data intensive loops to improve the runtime and throughput performance of the entire application, we now compare the loop statistics, before and after optimization, collected by Acumem SlowSpotter.

Figure 41 shows the Fetch and the Miss ratios of the unoptimized loop mentioned in 'Problem 1'. Both the fetch and miss ratios for this loop are enormously high. If corrected, the fetch ratio for the loop can be brought down to almost 0% within the cache region (2MB). Similarly, the miss ratio can be brought down too. Note that the Fetch and the Miss ratios follow a similar pattern with miss ratio lower than the fetch all the time.
For the optimized loop, the fetch and the miss ratios are never reported by the Acumem tools; our reasoning is that the fetch/miss ratios for optimized code in problem 1 are 0%. The Fetch and Miss ratios for the second loop (in function 'Roulette') are shown in Figure 42. Here too, both the fetch and miss ratios are considerably high.

The fetch and miss ratios for the optimized loop from problem 2 are shown in figure 42; the curve is the same as the predicted optimized curve.

![Figure 42: Fetch/Miss ratio curves for unoptimized loop(above), optimized loop(below), in problem 2](image)

Similarly, the fetch utilization for the loop in problem 1 (shown in figure 43) can be improved as well. In the unoptimized loop, fetch utilization is below 20% within the cache region, staying consistently low all the way.

The fetch utilization is not reported for the optimized loop by the Acumem tool indicating that no problems persist in the statement in Problem 1.

![Figure 43: Fetch utilization for unoptimized loop](image)
The Fetch utilization for the unoptimized loop in problem 2 is shown in Figure 44; it is under 20%.

The fetch utilization for the optimized loop in problem 2 is shown in Figure 44; it is always at 100%.

**Figure 44: Fetch utilization for unoptimized loop(above), optimized loop(below)**

### 7.7 Throughput Scalability

We perform throughput scalability tests by running multiple instances of the CIGAR application simultaneously on equal number of cores. Each of the running application instances is bound to a predefined core. We perform throughput tests on two different processors to have a better view of the optimized application's behavior on hardware platforms with varying properties.

Figure 45 shows the throughput comparison. The system used for generating the results consisted of an Intel Xeon E5345 QuadCore processor running at 2.33 GHz, with each processor having L2 cache of 4MB, shared by every two processors.
Compared with the original CIGAR application, the optimized application scales well in terms of relative throughput. Running 4 simultaneous instances of both the applications (in separate runs) on a QuadCore shows that the optimized application performs way better in terms of memory bandwidth consumption when compared with the original one. When running 4 instances over 4 cores, the original application shows a slowing down trend, not achieving as much speedup, due to excessive memory bandwidth consumption. On four cores the absolute throughput speedup for the original application remains limited at 1.2X. Compared to that, the optimized version keeps scaling phenomenally well, with absolute throughput at 40X the original application. The rate of difference in relative throughput for the two applications keeps growing with added number of cores. The relative gain on one core is 10X, on four cores it grows more than thirty folds at 33X. Unlike the original version of CIGAR, the optimized version does not show a slowing down trend at all, but keeps scaling at the same rate which shows that it has the capability to scale further on a couple of added number of cores, without showing considerable performance degradation, due to its excellent memory access properties.

Running multiple instances of the two applications on a system with AMD Istanbul shows a similar curve, though with lesser speedup gain for the optimized version. However, the optimized application here too runs multiple times faster for a single instance and keeps gaining relative throughput with added number of cores and instances.
Figure 46 shows the throughput comparison when the two applications are deployed on an AMD Opteron Istanbul running at 2.2 GHz with 512 KB of L2 cache per core and a 6MB L3 cache shared by all six on-chip cores.

For the above case of running the two applications on the AMD Opteron Istanbul, we make an observation that the original CIGAR application stops showing much improvement in absolute throughput gain when running multiple instances beyond 3, each bound to a separate core. The original CIGAR application posts a maximum absolute throughput gain of 2X at 5 and 6 instances. With added number of cores, and keeping hardware scalability consistent, this would eventually result in super-linear slow down of the original application. On the other hand, we observe that the optimized application keeps scaling pretty well, posting even better throughput at 6 cores; 33X in absolute terms. The relative throughput speedup for the optimized application (relative to original) starts at 6X when running a single instance on single core, and grows to a maximum of 17.4X for six instances. Moving to an added core, the absolute throughput gain for the optimized application stays consistent at 1.5, whereas the relative throughput gain shows a constantly increasing trend as the original application tends to slow down beyond 3 cores. This depicts a trend where the optimized application will continue to scale with added number of cores. This is essentially due to optimized and efficient data access behavior of the optimized application. The optimized application shows relative throughput speedup with a consistent rate.
7.8 Parallelizing the CIGAR Application

Having successfully optimized the serial implementation of the CIGAR application and investigating the performance aspects of the optimized implementation, we go on to test the two versions of the application with parallelism enabled. We do so in order to make a reasonable comparison of how well the two versions would perform in terms of runtime speedup and scalability, having been parallelized. Also, it is necessary to study the relationship between the relative throughput of the serial application and the runtime speedup of the parallel versions of the application, if there exists any. Doing so we can get the answer to how well would an optimized application scale after parallelism has been enabled, how can it be compared to its parallel unoptimized counterpart and if it really is better to parallelize the optimized application.

As mentioned in the previous sections that the most data and computation intense code-section within the entire CIGAR application consists of two functions called 'FindNBest' and 'Roulette'. According to runtime statistics collected by GPROF these two functions run for 99.9% of the total application runtime in the original CIGAR application, whereas for the optimized version it runs for almost 96% of total runtime. Here it should be noted that according to GPROF, the function 'FindNBest' is where the original application spends almost 87% of its total runtime. So here we focus on parallelizing the loops within this function only.

We rather follow a simple strategy for parallelizing the nested loop in the function 'FindNBest' which looks as follows in the original application.

```c
for(current = 0; current < howmany - 1 ; current++) {
    max = current;
    /* Find Next best */
    for(i = current + 1; i < size ; i++) {
        if((p + rank[i])->fitness > (p + rank[max])->fitness) {
            max = i;
        }
    }
    SwapInt(&rank[current], &rank[max]);
}
```

Instead of parallelizing the outer loop we focus on parallelizing the inner, this primarily due to the complexity of parallelizing the iterations of the outer loop. Since each iteration of the outer loop works on a common data structure 'rank' and every subsequent iteration depends upon the same data structure resulting from changes induced by the previous iteration, hence, maintaining consistency amongst parallel iterations of the outer loop becomes an extremely hectic task and might require making major changes to the application. Doing so would kill the whole point of making a reasonable comparison of the original and optimized applications after introducing identical parallelism in them.
So we simply parallelize the inner loop as follows

```c
for(current = 0; current < howmany - 1; current++) {
    max = current;
    my_max = max;

    /* Find Next best */
    #pragma omp parallel private (i) firstprivate(my_max)
    {
        #pragma omp for schedule (static) nowait
        for(i = current + 1; i < size; i++) {
            if((p + rank[i])->fitness > (p + rank[max])->fitness) {
                my_max = i;
            }
        }
        #pragma omp critical
        if(my_max > max)
            max = my_max;
    }
    SwapInt(&rank[current], &rank[max]);
}
```

We use static scheduling for the worker threads in order to avoid thread migration, since the data is reused by the worker threads once they return to the inner loop, having incremented 'current' in the outer, dynamic scheduling of the inner loop will result in unpredictable runtime allocation of the mapped or available resources. This may cause worker threads to be scheduled onto different cores, for the next iteration of the inner loop (due to runtime resource availability), which may result in thread migration, causing a lot of data relevant to a given worker thread to be communicated and fetched from neighboring cores' caches, where that particular thread had been scheduled during the previous iteration of the outer loop.

We follow exactly the same steps for parallelizing the optimized application in an effort to make both the versions comparable after being parallelized. Notice that there is critical block for sorting out the real 'max' value after the threads finish with the inner loop, thereby causing the applied parallel section to be synchronized in the end and hence avoiding an EP solution to the parallelization problem, which would have been very interesting to investigate.

### 7.9 Performance Evaluation after Parallelizing

For performing a suitable performance evaluation of the parallel versions of the two applications, a criteria is set so that the results produced are intuitively comparable. The scheduling is permanently set to 'static' within the code, so environment variable 'OMP_SCHEDULE' is not required to be set explicitly. We disable worker thread count variation at runtime by setting environment variable 'OMP_DYNAMIC' to 'false'. Finally, we vary 'OMP_NUM_THREADS' from 1 to 4 (as many cores as are available per chip) in 4 different runs and record the runtime for measuring the speedup relative to serial
original application runtime. In addition, we bind the application to equal number of processing cores as the value of 'OMP_NUM_THREADS'.

Figure 47 shows the speedup of the parallelized original and optimized versions of the CIGAR application. The system used in this case is the Intel Xeon E5345 QuadCore processor running at 2.33 GHz, with each processor having L2 cache of 4MB, shared by every two processors.

![Figure 47: Parallel speedup comparison over multiple cores, on Intel Xeon E5345](image)

Here, the relative speedup for the optimized application reports to be 7X for three threads and 6.2X for four threads showing trends of stabilizing around the factor of 6. Although for growing thread count the absolute speedup for the optimized application is considerable, the speedup trend relative to the original parallelized application seems to keep decreasing beyond the thread count of 2. Going hypothetically from multicores to manycores in the current scenario, extrapolating this trend would not show any significant relative speedup for the optimized application and is expected to stay around 6X. The growing thread count also shows diminishing rate of absolute speedup gain for the optimized application, strongly suggesting that on the given platform the application will not scale well beyond 4 cores. Going from three to four cores the rate of absolute speedup gain for the optimized application is 1.1, where this factor is 2 when going from serial execution to two threads. Also, something very interesting to note here is that the speedup achieved by the original application at four threads is lower than the serial version of the optimized application; the original CIGAR application is still almost twice
as slow. In effect, indicating that you should consider optimizing the application before parallelizing it, asserting Sanjiv Shah's advice on parallelism.

We study yet another evaluation for runtime speedup over a different platform. Figure 48 shows the speedup of the parallelized original and optimized versions of the LBM application. The system used here is AMD Opteron Istanbul SixCore processor running at 2.2 GHz, with each core having private L2 cache of 512KB, and L3 cache of 6MB shared by all six on-chip cores.

![Parallel speedup comparison over multiple cores, on AMD Opteron Istanbul](image)

Figure 48: Parallel speedup comparison over multiple cores, on AMD Opteron Istanbul

Somewhat like the previous case, the absolute speedup of the optimized application keeps growing considerably reaching to a maximum of 16X at six threads (scheduled over six cores). However, when taking into account the relative speedup, compared with the original parallelized application, we see a decreasing speedup trend, starting at 5.3X speedup relative to original parallel application at 2 threads down to 4.2X at six. Also the absolute speedup gain for the optimized application is maximal when going from serial execution to two threads, where the speedup gain for the optimized application is almost 1.8. However, beyond three threads, the rate of absolute speedup gain is decreased to slightly more than 1 at almost 1.1 and stays consistent up to six threads. The original parallel application also posts the best absolute speedup gain of factor 2 when going from serial execution to two threads and starts showing slowing down trend beyond thread
count of three. Again, the absolute speedup for the serial optimized application is much better than the original parallel application at six threads; the optimized serial application is 1.6 times as fast as the original application running with six worker threads.

The CIGAR application studied here is a good example of how well scientific applications may perform through adding very simple optimizations to the code. It helps demonstrate how these simple optimizations affect various properties of the application, such as absolute speedup, relative speedup, speedup scalability, throughput scalability and parallel speedup. This application is well parallelizable and good to study traits of parallel applications for parallel architectures, especially multicores. It was not feasible with this application to study the relationship between parallel speedup and relative throughput, since only a part of the code had been parallelized to study parallel speedup gain trends and application behavior when parallelized for multicore architectures. A problem faced in finding a good solution to the parallelism part was not being able to approach an EP solution. This clearly has a role to play in the considerable slowing down of the optimized application beyond three threads, where threads executing on more than three cores have to synchronize frequently in each iteration of the outer loop.
Chapter no.8

Case Study 4 – LIBQUANTUM
8.1 Introduction

The LIBQUANTUM application is an implementation simulating quantum computers based on the principles of Quantum mechanics. The application contains an implementation of Peter Shor's algorithm for factorization of numbers, discovered by him in 1994, having polynomial time complexity, much better than prime factorization's exponential time complexity. Libquantum offers to simulate quantum algorithms and develop quantum error correction algorithms.

The specific implementation taken into consideration in this case study is provided as part of the SPEC CPU2006 integer application benchmark suite, called 462.libquantum. We add optimizations to the code to show possible improvements in performance of this application.

Details about the application as provided by SPEC can be found here


This case study describes the issues that have been resolved to demonstrate improvements in application performance. Other issues have been ignored.

It is recommended to have the code for 462.libquantum benchmark from SPEC CPU2006 when looking into this case study.

8.2 Acumem SlowSpotter's Diagnosis

Having profiled the Libquantum application with the Acumem SlowSpotter tool, the report immediately points out at areas of possible high performance gains. SlowSpotter complains about Memory Bandwidth, Memory Latency and Data Locality problems related with this application and lists them in order of estimated performance gains, placing the code region with the highest estimated gain at the top.

Figure 49 shows the “Bandwidth issues”, ranked by the percentage of fetches, in the issues section of the report.
Figure 50 shows the loops in order of worst loop stats (or highest expected performance gains), in the loops section of the report.

![Statistics for instructions of this issue](image)

**Figure 49: Bandwidth Issues (left pane) and problem areas in code (right pane)**

**Figure 50: Loops with highest expected gains (left pane) and loop in code (right pane)**
Figure 51 shows the summary of the overall sampled Libquantum application

<table>
<thead>
<tr>
<th>Issues</th>
<th>Loops</th>
<th>Summary</th>
<th>Files</th>
<th>Execution</th>
<th>About/Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global statistics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miss ratio</td>
<td>0.9%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch ratio</td>
<td>21.2%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Writeback ratio</td>
<td>8.9%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Upgrade ratio</td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication ratio</td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch utilization</td>
<td>49.6%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-back utilization</td>
<td>37.3%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication utilization</td>
<td>100.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysis parameters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor model</td>
<td>Intel(R) Pentium(R) D CPU 3.20GHz (auto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of CPUs</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of caches</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache level</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache size</td>
<td>2M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line size</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replacement policy</td>
<td>random</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The statistics for the Libquantum application show overall fetch ratio at 21.2%, miss ratio at 0.9% and fetch utilization of just 49.6% within the cache region of 2MB.
Acumem SlowSpotter's diagnosis immediately points out at the following major problems

In the following loop in function 'quantum_toffoli' (We refer to it as “problem 1” in the case study)

```c
for( i = 0; i < reg->size; i++)
{
    /* Flip the target bit of a basis state if both control bits are set */
    if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control1)) PROBLEM 1
    {
        if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control2))
        {
            reg->node[i].state ^= ((MAX_UNSIGNED) 1 << target);
        }
    }
}
```

and in the following loop in function 'quantum_sigma_x' (referred as “problem 2” in case study)

```c
for( i = 0 ; i < reg->size; i++ )
{
    /* Flip the target bit of each basis state */
    reg->node[i].state ^= ((MAX_UNSIGNED) 1 << target); PROBLEM 2
}
```

and in the following loop in function 'quantum_cnot' (referred as “problem 3” in case study)

```c
for( i = 0; i < reg->size; i++ )
{
    /* Flip the target bit of a basis state if the control bit is set */
    if((reg->node[i].state & ((MAX_UNSIGNED) 1 << control))) PROBLEM 3
    reg->node[i].state ^= ((MAX_UNSIGNED) 1 << target);
}
```

Problem 1:
For the loop in this problem, the report points to a very high accumulated percentage of fetches of almost 54% of all the memory fetches in this application, with a cumulative fetch utilization of almost 50% for the entire 'if' statement involving two memory reads, where one of them is a structure access. The instruction stats show accumulated percentage of misses greater than 60% of all the cache misses of this application, fetch ratio at an average of 29% and miss ratio at 1.3%, for the entire 'if' statement in the 'for' loop. Reducing the fetch and miss ratio for this statement would greatly help improve bandwidth issues.
Why Problem 1?
The reason for having an extremely high fetch ratio for the loop in this problem, is only partial utilization of the data being pulled into the cache due to the memory access being made as a result of access to the struct 'node' (which in turn is part of 'reg' struct). Every single instance of the struct 'node' is 16 bytes in size. The given loop goes through an array of type struct 'node', indexing the array with 'i'. Every time a miss occurs while trying to access a given instance of the struct, 'node[i]', 64 bytes (cache-line size) of contiguously located data are pulled into the cache including node[i] up to node[i+3]. This means that the loop will not suffer a cache miss until reaching 'node[i+4]'. When it does miss in cache, it again pulls in 64 bytes of contiguous data, with required data inclusive. Also, if hardware prefetching utility is good at detecting such consistent strides in data locality, cache misses can be further avoided. However, the problem here in this loop is the fetch ratio and the major reason for the problem is partial use of data being fetched into the cache. Every time an access to the struct 'node' is made, only half of the data (8 bytes for variable 'state') is ever used, whereas the other half goes wasted as it is never accessed and eventually evicted from the cache without ever being used. Pulling unwanted data into the cache not only lowers the overall fetched data utilization for the given loop, but also adds to the loop's fetch ratio, resulting in wasted bandwidth and shared cache space. The relatively low miss ratio is because of good hardware prefetching probability which is 97% in this case.

Problem 2:
The loop pointed out in this problem causes 21.5% of the total fetches, and 3.6% of the total misses. The fetch ratio for this loop is very high, at almost 25%, whereas the miss ratio is at 0.3%. Similar to the case in problem 1, the fetch utilization is almost 50% for the loop in this problem. Also, the writeback utilization is 50%.

Why Problem 2?
The issue in the loop pointed out in this problem is similar to that in problem 1. However, in this case the data being pulled into the cache for the struct 'node' is also being written to. Still only 8 out of the 16 bytes of the struct data are ever used (or written to); the other half of the data available in the cache remains unused and is eventually evicted from the cache. This tends to not only lower the fetch utilization to 50%, but also the writeback utilization to the same i.e. 50%, since only half of the data pulled into the cache (in this loop) is changed, whereas the other half of the cache line remains unchanged. Here too, the same movement of unwanted data in and out of the cache adds to the fetch ratio, lowering the use of data fetched and the writeback utilization as well. However, here the wasted bandwidth is even greater, since the data needs to be written back to the memory; and since only half of every cache line is changed, all of it needs being re-written to the memory, thereby requiring transporting unchanged data as well. Miss ratio remains low due to a good hardware prefetching probability of almost 99%.

Problem 3:
The loop pointed out in this problem is similar to the loop in problem 1 and causes 8.6% of the total fetches. The fetch ratio for the loop is at a high of 25.6%, whereas the miss
ratio is almost negligible at 0.2%. Similar to problems 1 and 2, the fetch utilization here too is 50%.

Why Problem 3?
The problem here too is similar to problem 1. Partial utilization of the data pulled into the cache results in low utilization of fetched data and higher fetch ratio, along with significant wasted bandwidth dedicated to moving the unwanted data from the memory into the cache. Miss ratio is low due to very high hardware fetch probability of 99%.

8.3 Acumem SlowSpotter's Advice

Problem 1:
Acumem SlowSpotter advises
1. Improvement in fetch utilization

Problem 2:
Acumem SlowSpotter advises
1. Improvement in fetch utilization
2. Spatial blocking for data reuse and identifying the problem of required data's immature cache eviction
3. Loop fusion with another loop involving similar operations

Problem 3:
Acumem SlowSpotter advises
1. Improvement in fetch utilization

8.4 Steps for Optimization

Note: Please refer to code in Figure 52 and end of section 4.7 to see changes made in every point

1. To improve the fetch utilization for loop in problem 1 and to avoid fetching unnecessary data into the cache, we remove the struct 'node' from the struct 'reg' completely, and replace it with pointers to arrays 'node_state' and 'node_amplitude'. Keep in mind that struct 'node' has only two component variables, namely 'state' and 'amplitude', which we replace with independent arrays in struct 'reg' containing corresponding values (of 'state' and 'amplitude') for all the instances of array 'node' in struct 'reg'. Now instead of accessing 'state' through 'node' as 'node[i].state', we access the array 'node_state' in 'reg' indexing it with 'i' as 'node_state[i]'. Throughout the code of the application, we make consistent changes replacing the access to the struct 'node' with array access of its component variables. Doing so we completely discard the struct 'node', since all its component variables can now be accessed through independent arrays in struct 'reg', thereby removing the need of having to keep the struct 'node'. Doing so we guarantee that all the data being pulled into the cache in the loop in this problem, due to the accesses made to 'node_state' array, will be used, thereby resulting in full utilization of data fetched into the cache.
2. In addition to the above step of replacing struct access with simple array access, we create new functions by fusing functions often called subsequently (one after the other) within the code. In functions 'test_sum', 'muxfa' and 'muxfa_inv', the functions 'quantum_toffoli', 'quantum_cnot' and 'quantum_sigma_x' are frequently called subsequently in pairs. The loops in these functions are quite similar and from the stack trace (provided in the Acumem report) we can see what points in the code they are called from in pair. We can then fuse the related functions first and then fuse the loops in them. As a result we come up with five such new fused functions namely

- fused_quantum_sigma_x_toffoli, (combining functions 'quantum_toffoli' and 'quantum_sigma_x')
- fused_quantum_toffoli_sigma_x'
- fused_quantum_cnot_toffoli
- fused_quantum_toffoli_cnot
- fused_six_quantum_toffoli

Having made the recommended changes in the above two steps, this is how the new code in function 'quantum_toffoli' looks like

```c
for(i=0; i<reg->size; i++)
{
    /* Flip the target bit of a basis state if both control bits are set */
    if(reg->node_state[i] & ((MAX_UNSIGNED) 1 << control1)){
        if(reg->node_state[i] & ((MAX_UNSIGNED) 1 << control2))
        {
            reg->node_state[i] ^= ((MAX_UNSIGNED) 1 << target);
        }
    }
}
```

Code excerpt mentioned in problem 2, in function 'quantum_sigma_x', looks like this

```c
for(i=0; i<reg->size; i++)
{
    /* Flip the target bit of each basis state */
    reg->node_state[i] ^= ((MAX_UNSIGNED) 1 << target);
}
```

and, code excerpt mentioned in problem 3, in function 'quantum_cnot', looks like this

```c
for(i=0; i<reg->size; i++)
{
    /* Flip the target bit of a basis state if the control bit is set */
    if((reg->node_state[i] & ((MAX_UNSIGNED) 1 << control)))
    {
        reg->node_state[i] ^= ((MAX_UNSIGNED) 1 << target);
    }
}```
Same changes are reflected in the new 'fused functions', where the struct access of variable 'state' (as in 'node[i].state') is changed to simple array access (as 'node_state[i]'). As an example to the fusion process described briefly in the second step, the new fused function 'fused_quantum_sigma_x_toffoli' is shown in Figure 52.

```c
void fused_quantum_sigma_x_toffoli
    (int control1, int control2, int target_t, int target_s, quantum_reg *reg)
{
    int i;
    int qec;
    quantum_qec_get_status(&qec, NULL);
    
    // Determine if a fusion is possible
    if((quantum_read_status()) ||
        // The need to call decohere prevents fusion
        qec ||
        // Should call the_ft functions
        quantum_obj_code_put(SIGMA_X, target_s) ||
        // Should return from second function
        quantum_obj_code_put(TOFFOLI, control1, control2, target_t) // Should return from second function
    )
    {
        // Call functions one by one
        printf("NOT FUSED\n");
        quantum_sigma_x(target_s, reg);
        quantum_toffoli(control1, control2, target_t, reg);
    }
    
    disc
    {
    // Fuse the main loops
    // quantum_sigma_x(target_s, reg); // FUSE THESE!
    // quantum_toffoli(control1, control2, target_t, reg); // FUSE THESE!

    for(i=0; i<reg->size; i++)
    {
        // SIGMA Flip the target bit of each basis state
        reg->node_state[i] ^= (MAX_UNSIGNED) 1 << target_s;
        
        // TOFFOLI Flip the target bit of a basis state if both control bits set
        if(reg->node_state[i] & (MAX_UNSIGNED) 1 << control1)
        {
            if(reg->node_state[i] & (MAX_UNSIGNED) 1 << control2)
            {
                reg->node_state[i] ^= (MAX_UNSIGNED) 1 << target_t;
            }
        }
    }
    }
```

*Figure 52: Optimized code*
caused a natural fragmentation in sets of meaningful data and a stride of 8 bytes. However, since this stride was consistent (8 bytes), it was predictable by the hardware prefetch utility, resulting in a hardware prefetch probability of 97% or higher. Also, in accessing an array in the given situation (after optimization), a cache-miss will necessarily be followed by at least 7 consecutive cache-hits, effectively lowering the miss/fetch ratios and the stalls suffered at such points. Moreover, even such mandatory cache-misses are avoided in current scenario due to hardware prefetch utility being able to accurately guess the next piece of data-to-be-fetched, resulting in no cache-misses and hence no stalls. Sequential access to the arrays will also guarantee a two-fold increase in the fetch utilization, going from 50% to 100% for the given loops. In addition to this, fusion of functions and loops will guarantee further improvements in runtime, firstly due to lesser function calls, and secondly due to merging related operations into a single loop, instead of having to iterate over them separately.

The above improvements in the code are bound to produce effective results which should be clearly visible in application's run-time and relative throughput speedup, since profiling the application reveals that the three functions containing the above code run for almost 97% of the total application's duration. Lowering runtime for this piece of code means lowering of the entire application's runtime.

8.5 Performance Evaluation

We evaluate the performance of the optimized and the original Libquantum applications using more than a single platform in an attempt to get a realistic view of how the two applications would behave in the real world on different platforms, they are deployed on.

On an Intel QuadCore E5345 (64-bit Xeon processors) running at 2.33 GHZ with L2 cache of 4MB, running the original Libquantum application takes 1401 seconds. Whereas, running Libquantum application with the optimized code takes just 618 seconds, which is almost thirteen minutes faster than the original application. These results show a runtime decrease of almost 56% for the entire application.

Measuring the speedup of the application after optimization, we see that the optimized application runs more than twice as fast as the original one at 2.3X. Imitating the speedup behavior of the entire application, the optimized functions run more than four times as fast as the unoptimized functions in the original application, at almost 5X. Data for optimized functions has been gathered with help of GPROF in this case study.

Figure 53 shows the performance gains in application speedup and optimized functions.
Profiling the two applications with the GPROF tool shows that in the original version of the application, the three functions ('quantum_toffoli', 'quantum_sigma_x' and 'quantum_cnot') with the unoptimized code run 97% of the total application's runtime. Whereas, for the optimized code, the same three functions run for 39.5% of the entire application's runtime. This, from our reasoning, is primarily due to reduction in stalls from cache-misses in the optimized loops. Runtime figures from GPROF show the total accumulated runtime for the optimized functions is lowered by 79%, thereby significantly lowering the entire application's runtime.

All of this speed-up is the result of the optimization steps listed in the previous sections, focusing mostly on three data-intensive loops of code in the entire application.
8.6 Acumem SlowSpotter's Profiling Comparison

Comparing the profiling data collected from the Acumem SlowSpotter tool gives us the following results.

Figure 54: Comparison of Fetch ratio for Optimized application

Figure 54 compares the Fetch ratio of the Libquantum application before and after optimization. The curve shows that the fetch ratio has been lowered significantly and minimized to 7.4% within the cache-memory region (2 MB in this case); starting at almost 14 percentage units, the improvement in the fetch ratio within the cache region remains consistent at the same. As clearly observable, the fetch ratio of the application remains stable throughout at 7.4% within the cache region and minimizing to 0% earlier than the original application's fetch ratio, showing the active dataset in the optimized application to have shrunk to half the active dataset of the original application. Having such low fetch ratio helps speedup the application significantly, as it avoids memory bottlenecks (especially in processors with low memory bandwidth interfaces) which cause memory latency, consequently causing requested data to arrive late from the main memory; instructions may stall for long, as they wait for the data to arrive in cache.

Application's overall fetch ratio is the property that affects the relative throughput speedup; lower overall fetch ratio results in significant improvement in relative throughput gain.
Figure 55 compares the fetch utilization for the optimized application with the unoptimized version. The fetch utilization for the optimized version posts an improvement of at least 45 percentage units within the cache region. For the optimized parts of the code, the spatial usage of the data is almost always stable and consistent at 100%. This means that all the data being brought into the cache is used before being evicted. Comparing the miss ratio curve to the fetch utilization curve here, we see that here it is the most data intensive parts of the application that dominate this curve, the portions that have been optimized to use 100% of data fetched into the cache.

Keeping in mind that for this application we simply optimized three most data-intensive loops to improve the runtime and throughput performance of the entire application, we now compare the loop statistics, before and after optimization, collected by Acumem SlowSpotter.

Figure 56 shows the Fetch and the Miss ratios of the unoptimized loop mentioned in 'Problem 1'. The fetch ratio for this loop is enormously high at almost 30%. If corrected, the fetch ratio for the loop can be brought down by a factor of 2 to less than 15% within the cache region (2MB). Miss ratio in this case is minimal at 1.3%

After optimizing the code, the fetch ratio curve for the optimized loop in problem 1 look as shown in figure 56. The fetch ratio is lowered to 13.6%.
The Fetch and Miss ratios for the loop in problem 2 are shown in Figure 57. Here too, the fetch ratio is considerably high.

The fetch and miss ratio curves for the loop in problem 2 are shown in figure 57. Fetch ratio is minimized to predicted 16%.

Figure 56: Fetch ratio curve for unoptimized loop(above), optimized loop(below) in problem 1

Figure 57: Fetch ratio curve for unoptimized loop(above), optimized loop(below) in problem 2
The Fetch ratio for the loop pointed out in problem 3 is shown in Figure 58. The fetch ratio is almost 26%.

For the loop in problem 3, figure 58 shows the fetch ratio curve. Here fetch ratio is down to 8%. The optimized fetch ratio curve is very close to the predicted improvement.

Similarly, the fetch utilization for the loop in problem 1 (shown in figure 59) can be improved as well. In the unoptimized loop, fetch utilization is almost 50% within the cache region, staying consistent all the way, showing that half of the data pulled into the cache, during the execution of this loop, goes unused.

After having optimized the code, the fetch utilization for all the three loops improves to almost 100% throughout. Fetch utilization for loop in problem 1 is shown below in figure 59.
The Fetch utilization for the unoptimized loop in problem 2 is shown in Figure 60; it is again almost 50%.

The Fetch utilization for the optimized loop in problem 2 is expected to be 100%, since the tool doesn’t complain about it at all.

The fetch utilization for the unoptimized loop in problem 3 is shown in Figure 61; it is always at 50%.

Fetch utilization for loop in problem 3, shown in figure 61; here too it is close to 100%
8.7 Throughput Scalability

We perform throughput scalability tests by running multiple instances of the Libquantum application simultaneously on equal number of cores. Each of the running application instances is bound to a predefined core. We perform throughput tests on three different processors to have a better view of the optimized application's behavior on hardware platforms with varying properties.

Figure 62 shows the throughput comparison. The system used for generating the results consisted of an Intel Xeon E5345 QuadCore processor running at 2.33 GHz, with each processor having L2 cache of 4MB, shared by every two processors.
Compared to the original application, the optimized version of the Libquantum application seems well scalable. For a single instance, the relative speedup for the optimized application is 2.3X. For 3 and 4 cores, the relative throughput speedup for the optimized application peaks at almost 3X. This speedup is primarily due to better bandwidth usage properties of the optimized application when compared with the original application. Beyond 2 cores, the original application hardly shows any throughput gain, whereas the optimized application keeps gaining relatively well. However, when going from 3 to 4 cores, for the optimized application we see that the absolute gain in throughput speedup isn't considerable, indicating that even the optimized application will not scale well theoretically beyond four cores. Beyond 2 cores, the original application hardly posts any improvement at all, indicating a slowing down trend beyond 4 cores.

Running multiple instances of the two applications on a system with AMD Istanbul shows a similar curve, though with comparatively better speedup gain for both the optimized and the unoptimized versions. However, the optimized application here too runs multiple times faster for a single instance and keeps gaining relative throughput with added number of cores and instances.

Figure 63 shows the throughput comparison when the two applications are deployed on an AMD Opteron Istanbul running at 2.2 GHz with 512 KB of L2 cache per core and a 6MB L3 cache shared by all six on-chip cores.
For the above case of running the two applications on the AMD Opteron Istanbul, we see that both the versions of the Libquantum application gain throughput speedup with added number of cores. Beginning at 2.1X for a single instance, the relative throughput for the optimized application peaks at almost 3X for 5 and 6 cores. With each added core we see a significant throughput gain for the optimized application up to 3 cores. However, beyond three cores, the gain is not as much significant, and going form 5 to 6 cores the achieved gain is minimal, indicating that extrapolating the throughput speedup curve beyond 6 cores will not be scalable. For the original application the situation is even worse, where there is hardly any gain beyond three cores, showing a slowing down trend beyond 6 cores. So, for AMD Istanbul, although the throughput gain is considerably high, none of the either versions of the application will scale well beyond 6 cores.

Finally we look at the curve for the relative throughput for the two applications on the Intel i7 QuadCore processor, running at 2.66 GHz with 8 MB of shared cache and 3 channels of DDR3 1066 MHz memory. Figure 64 shows the throughput curve.
Beginning with a speedup of 1.3X, the optimized application approaches 1.7X throughput speedup relative to the original application's speedup. Keep in mind that this processor has greater than 3 times the bandwidth (due to three memory channels) compared to any normal processor, and a considerable speedup of almost 2X shows that despite the enormous available bandwidth the optimized application can still post significant speedups, while the original application still remains slower due to its inappropriate bandwidth properties, despite the availability of all three hi-speed memory channels. Extrapolating the speedup trend shows that unlike other processors with single memory channel, keeping the hardware resources consistent, the optimized application will keep on scaling extremely well on this new cutting-edge architecture.

8.8 Parallelizing the Libquantum Application

Our study of the Libquantum application indicates that adding parallelism to the code is not feasible without making major changes in it. The reason of parallelizing the application here was to find existing opportunities of introducing thread parallelism and then study the impact of performance optimization in serial application on the parallelized application. That purpose did not seem being served feasibly as the Libquantum application required some very strong changes to make it well-parallelizable. Hence the idea to do so and study effects on parallelism was dropped.
References:

2. Memory Profiling using Hardware Counters - Marty Itzkowitz, Brian J. N. Wylie, Christopher Aoki and Nicolai Kosche, Sun Microsystems Inc, Menlo Park, California
3. MCF application - http://www.zib.de/Optimization/Software/Mcf
10. CIGAR - http://ecsl.cse.unr.edu/
15. HPCWire - http://www.hpcwire.com/
17. HPCWire - HPC@Intel: When to Say No to Parallelism by Sanjiv Shah