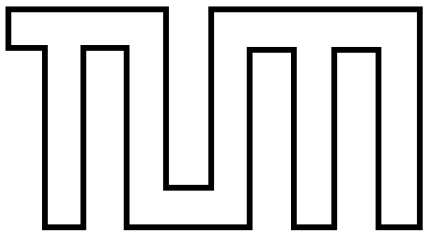

Technische Universitaet Muenchen
Fakultaet fuer Informatik

Diplomarbeit in Informatik

Exploiting Multi-core Processors for Memory-bound Numerical Codes by using Prefetching Techniques

Anagnostopoulou Vlasia

Aufgabensteller: Prof. Dr. Arndt Bode
Betreuer: Dr. Josef Weidendorfer
Abgabedatum: 16. Oktober 2006

Ich versichere, dass ich diese Diplomarbeit selbstaendig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Muenchen, den 16. Oktober 2006 Anagnostopoulou Vlasia

Abstract

Numerical codes occur in almost all science and engineering disciplines; e.g., computational fluid dynamics take effect on applications such as aeronautics, automotive, power generation, chemical and petrochemical process. They consist of small kernels, each carrying out a large number of Floating Point operations on proportionally big data sets. Moreover, such codes introduce significant portion of iterations. If we suppose that the data sets resided entirely in the main memory and every time that data were needed for computation were fetched from the main memory, then numerical codes would be confounded to very poor execution times, because of the existing barrier for quick memory accesses. Happily, memory hierarchy and optimizing techniques that can be applied to numerical codes as well have existed since a long time, bringing the execution times of numerical codes up to satisfactory levels.

From the existing optimization techniques, the technique of prefetching data could be proved as particularly beneficial when speaking of numerical codes that act on large data portions, especially if a multi-core processor would be utilized as a prefetch engine. The potential of improvement of the execution times from the utilization of this technique and the further investigation of such a possible gain comprise the subject of this Thesis.

Table of contents

Technische Universitaet Muenchen 1

Table of contents 5

1 7

1.1 Motivation 7
1.2 Structure 9

2 11

2.1 CPU Designs for Enhancing Performance 11
2.1.1 Instruction Level Parallelism 12
2.1.2 Thread Level Parallelism 13
2.2 Memory Organization for Greater Performance 14
2.2.1 Memory Hierarchy 14
2.2.2 Aspects of Caches 16
2.3 Other Optimization Techniques 17
2.3.1 Loop Optimizations 18
2.3.2 Data Prefetch 20

3 23

3.1 Implementation Schema 23
3.2 Decisions over the Implementation Schema 24
3.2.1 Loop-Blocking Optimization 24
3.2.2 Prefetch Optimization 25
3.2.3 In UNIX: Threads over Processes 25
3.2.4 Effective Scheduling for Multi-core Processors 26
3.2.5 Inter-thread Communication 27
3.2.6 Schematic Depiction of the Inter-thread Communication 30
3.3 Implementation of the Prefetch Thread 31

4 33

4.1 Addition and Multiplication Benchmarks 34
4.1.1 Naive Code 34
4.1.2 Code with blocked loops 37
4.1.3 Code including the Prefetch Thread 41
4.1.4 Prefetch Interleaved into the Code 44
4.2 Jacobi Method Benchmark 48
4.3 Computational Fluid Dynamics 51

5 53

5.1 Addition Benchmark 54
5.2 Multiplication Benchmark 58
5.3 Jacobi Benchmark 62

6 67

Appendix A 71

Appendix B 73

Chapter 1

Introduction

1.1 Motivation

The gap between CPU and memory speeds widens as processor speeds are increasing at a rate greater than the one of memory speeds [5]. What is more, this performance gap is estimated to continue increasing. Besides, there is a bandwidth problem with memory. Thus, there is a barrier to performance at memory. This barrier is a major obstacle to computer performance and even more to performance of numerical codes. It worsens the overall execution time and generally holds us back from taking full advantage of the potentials of current CPUs. It is clear that this memory barrier has to be overcome.

Present day computers have a hierarchy of memory as a way of reducing this memory barrier. The hierarchy of memory introduces small fast memory, that is termed cache, at the top of the hierarchy closest to the CPU, getting larger and slower down the hierarchy, further away from the CPU. Usually there are two levels of caches included: a fast, small, level 1 (L1) cache and a relatively slower, larger, level 2 (L2) cache. L2 cache is faster than main memory access. Data that do not reside in L2 cache have to be fetched from the main memory. Given that the cache is faster than the main memory in fulfilling a CPU's memory requests, one clearly needs to maximaze the number of cache hits which are the accesses that can be satisfied from the cache.

When the data set to be computed of a numerical application is large such that it no longer fits in the cache, clearly the program is condemned to poor performance. Therefore, the code should be implemented in a cache aware manner, meaning that it should be optimized so as to maximize the number of the accesses that can be satisfied from the L2 cache. In this case, the cache hierarchy can be a good solution to the memory barrier.

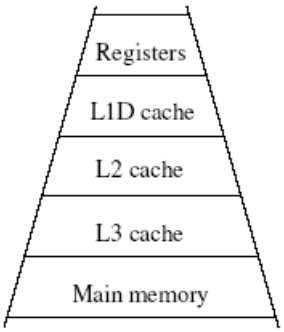

Figure 1.1. Cache and main memory hierarchy

New generation computers and servers since a few years already adopt the multi-core architecture, by including multiple cores on a single chip. One core is only able to handle one executional stream at a time. In a multi-core chip where multiple cores exist, each core can process one executional stream at a time. A difference of a multi-core system comparing to a single core is then that the microprocessor can handle multiple executional streams simultaneously and these executional streams can be handled independently to each other. Moreover, two cores might share some resources of the system; e.g. the L2 cache, the memory and the bus controller. We will exploit the case where two cores and therefore the tasks that are run on them share the same L2 cache, for the reasons explained below.

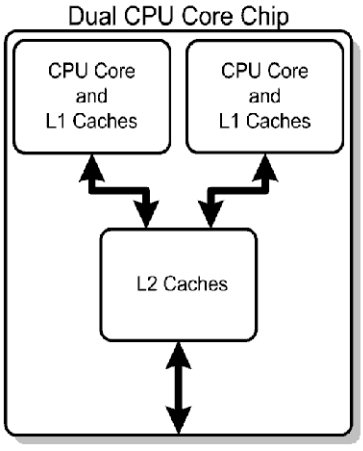

Figure 1.2. Double-core's architecture

As long as a code can be divided in autonomous executional streams, an approach as for the utilization of a second core is to use the core in order to run each executional stream on a seperate core in parallel and therefore improve the CPU speed. Such an approach however, would not improve the memory speed at all. On the other hand, by improving the CPU speed, it would contribute in the amplification of the gap between CPU and memory speeds. In another approach, the code is not divided in executional streams, but remains as a single stream of instructions. A new task is introduced to run on the second core. This task shall be responsible for fetching from the main memory into the cache data that are to be utilized by the main code (data prefetch). Therefore, at the time of execution, the code will not have to fetch the data that reside in the cache already and therefore, the memory latency could be reduced. Since numerical codes act on large data sets, a fact that memorywise introduces great memory latency, it is clear from the above that there can be a potential for signifficant gain for such codes with the employment of the second approach. Moreover, this is the reason why in this thesis we experiment on multicore machines that by two share a common L2 cache, because we want to realize the above approach.

We implement three benchmark applications in order to measure such a possible performance gain. As far as the first benchmark is concerned, it is an application that goes through the elements of an array multiple times performing the summing operation. The second benchmark is an implementation of the matrix multiplication procedure. Lastly, as third benchmark application we implement the Jacobi method. We will compare the performance of the above benchmarks for a naive implementation, for an implementation where the blocking technique is included, for an implementation the prefetch of data by a second core is introduced and for an implementation where prefetch instructions are interleaved within the main code.

1.2 Structure

This thesis follows the following structure; in the first chapter we discuss the basic concepts for enhancing the performance of the CPU and the memory. In the second chapter we explain the schemas that we utilized in order to implement the the benchmarks, while we defend our choice over each particular schema. In the third chapter we explain exactly how the benchmarks work and we present the codes of the different versions for each benchmark. Afterwards comes the results chapter, where we give the graphical interpretation of the measurements that we conducted. In the fifth and last chapter we present our conlusions.

Chapter 2

Concepts for Enhancing Performance

Numerical schemes over the past years have focused on iterative methods [1]. Such schemes are characterized by computational kernels based on loop nests, applying Floating Operations operations on large data sets. Therefore, there is a growing demand for continiously enhancing CPU computational power/capabilities. Throughout the past years there have been many methodologies for the design of CPUs and the memory system introduced. In this chapter, we will discuss beneficial concerning the performance methodologies that we have used in our benchmarks for the deployment of the CPU, and afterwards aspects of the optimal organization of the memory.

2.1 CPU Designs for Enhancing Performance

Attempts to achieve better performance have resulted in a variety of design methodologies that cause the CPU to behave less serial and more in parallel. When referring to parallelism in CPUs, two terms are generally used to classify these design techniques; instruction level parallelism and thread level parallelism. We explain both in the following subsections.

2.1.1 Instruction Level Parallelism

Instruction Level Parallelism (ILP) seeks to increase the rate at which instructions are executed within a CPU. One of the simplest methods to accomplish increased parallelism is to begin the first steps of instruction fetching and decoding before the prior instruction finishes executing. This is the simplest form of Pipelining, a technique that is utilized in almost all modern general-purpose CPUs. Pipelining allows more than one instruction to be executed at any given time by breaking down the execution pathway into discrete stages. This separation can be compared to an assembly line, in which an instruction is made more complete at each stage until it exits the execution pipeline and is retired.

Pipelining does, however, introduce the possibility for a situation where the result of the previous operation is needed to complete the next operation; a condition often termed data dependency conflict. To cope with this, additional care must be taken to check for these sorts of conditions and delay a portion of the instruction pipeline if this occurs. Naturally, accomplishing this requires additional circuitry. A pipelined processor can become very nearly scalar, inhibited only by pipeline stalls (an instruction spending more than one clock cycle in a stage).

Designs that are said to be superscalar include a long instruction pipeline and multiple identical execution units. In a superscalar pipeline, multiple instructions are read and passed to a dispatcher, which decides whether or not the instructions can be executed in parallel (simultaneously). If so they are dispatched to available execution units, resulting in the ability for several instructions to be executed simultaneously. In general, the more instructions a superscalar CPU is able to dispatch simultaneously to waiting execution units, the more instructions will be completed in a given cycle.

Most of the difficulty in the design of a superscalar CPU architecture lies in creating an effective dispatcher. The dispatcher needs to be able to quickly and correctly determine whether instructions can be executed in parallel, as well as dispatch them in such a way as to keep as many execution units busy as possible. This requires that the instruction pipeline is filled as often as possible and gives rise to the need in superscalar architectures for significant amounts of CPU cache. It also makes hazard-avoiding techniques like branch prediction, speculative execution, and out-of-order execution crucial to maintaining high levels of performance. By attempting to predict which branch a conditional instruction will take (branch prediction), the CPU can minimize the number of times that the entire pipeline must wait until a conditional instruction is completed. Specula-

tive execution often provides modest performance increases by executing portions of code that may or may not be needed after a conditional operation completes. Out-of-order execution somewhat rearranges the order in which instructions are executed to reduce delays due to data dependencies.

Both simple pipelining and superscalar design increase a CPU's ILP by allowing a single processor to complete execution of instructions at rates surpassing one Instruction Per Cycle (IPC) (according to the Best-case scenario). Most modern CPU designs are superscalar, and nearly all general purpose CPUs designed in the last decade are superscalar.

In later years some of the emphasis in designing high-ILP computers has been moved out of the CPU's hardware and into its software interface. A set of instructions that are issued in a given clock cycle as one large instruction with multiple operations is termed as Very Long Instruction Word (VLIW) and the respective strategy causes some ILP to become implied directly by the software, reducing the amount of work the CPU must perform to boost ILP and thereby reducing the design's complexity.

2.1.2 Thread Level Parallelism

The other commonly used strategy to increase parallelism of CPUs is to include the ability to run multiple threads at the same time, named Thread Level Parallelism (TLP) strategy. A thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. In the context of single processor design, the two main methodologies used to accomplish TLP are chip-level multiprocessing and simultaneous multithreading. While using very different means, both techniques accomplish the same goal: increasing the number of threads that the CPU(s) can run in parallel.

Simultaneous multithreading (SMT) is a topic out of the scope of this thesis and for this reason it is not going to be discussed here. In the case of chip-level multiprocessing (CMP), also known as Multicore, multiple processor cores are included, each executing one thread independently. A multi-core microprocessor is one which combines two or more independent processors into a single package, often a single integrated circuit. Multiple processor cores typically share a common second level cache (L2) and interconnect, while they may share a common third level cache (L3) as well. An architectural schema of a typical double-core processor can be seen in the picture below.

The sharing of cache(s) among the cores is of particular importance for this thesis; having the ability to handle a second thread running independently and simultaneously from the main, we are going to have this thread utilized by the main thread, in order to get support in the "costly" task of having to retrieve from the main memory the huge amount of data that typically underlie in numerical applications.

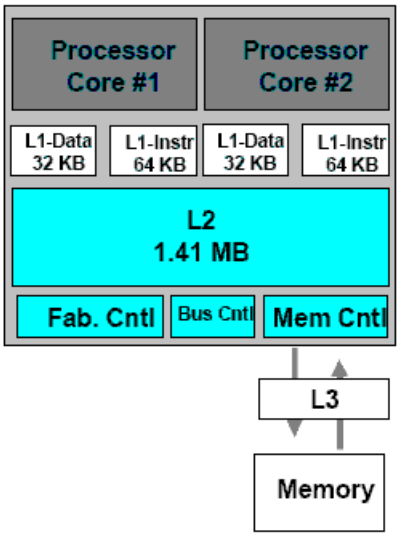

Figure 2.1. Architecture of the POWER4 double-core processor [11]

2.2 Memory Organization for Greater Performance

The impact of the constantly increasing gap between main memory performance and theoretically available CPU performance on the execution speed of an application is referred to as the memory wall. This is interpreted as the relatively low CPU to main memory bandwidth as well as the high latency of main memory accesses. Present day computers adopt a hierarchy of memory as a way of reducing this memory barrier and providing a reasonable amount of fast memory that programmers desire. Moreover, once the code respects the underlying hierarchical memory architecture, efficient execution can be expected [1]. This event is particularly utile for numerical applications that demand fast execution times. We will describe the hierarchical design of memory in the following section.

2.2.1 Memory Hierarchy

The foundation stone of the memory hierarchy is the cache. The cache is a small, fast memory which stores copies of the data from the most frequently used main memory locations. Because of their limited size, caches can only hold copies of recently used data

or code. Typically, when new data is loaded into the cache, other data has to be replaced. Caches improve performance only if data which has already been loaded is reused before being replaced. The reason why caches can substantially reduce program execution time is the principle of locality of references [2]. Locality can be subdivided into temporal locality and spatial locality. A sequence of references exhibits temporal locality if recently accessed data will be accessed again in the near future. A sequence of references exposes spatial locality if data located close together in address space tends to be referenced within a short period of time. The code optimization techniques we will discuss in the following Chapters aim at enhancing both temporal and spatial locality.

The hierarchy of memory consists of small fast memory at the top of the hierarchy which is closest to the processor, getting larger and slower down the hierarchy further away from the processor. There is a small and very fast memory sitting on top of the hierarchy which is usually integrated within the processor chip to provide data with low latency and high bandwidth; i.e., the CPU registers. The memory components which are located between the processor core and the main memory are called caches. They are intended to contain copies of memory blocks to speed accesses to frequently needed data.

The first level or L1 is integrated within the processor core itself (on-chip) and therefore, it is very fast. The L1 is often split into two separate parts; one for keeping data, the other for keeping instrcutions. The size of on-chip L1 cache is bounded to very small sizes, so as the signals from a very fast CPU do not to take too long. The next level of cache, called L2 or secondary cache, can reside inside or outside the core (off-chip). Typically, L2 caches provide data with lower bandwidth and higher access latency than L1 caches. In case that L2 is on-chip, a level three or L3 cache, typically off-chip, can be added. The size of this cache is significantly larger than the size of L2 cache and the bandwidth lower than the one of L2. The next lower level of the memory hierarchy is the main memory which is large but also comparably low. External memory such as hard disk drives or remote memory components in a distributed computing environment represent the lower end of any common hierarchical memory design.

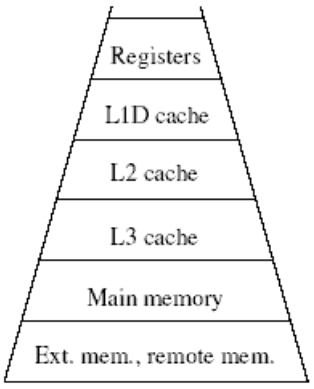

Figure 2.2. A typical memory hierarchy

2.2.2 Aspects of Caches

For the reference to the hierarchy of memory to be complete, we will have at this point to briefly report aspects such as the organization and the replacement policies of caches.

Data within the cache are stored in (cache) lines. Caches have a certain organization which describes in what way the lines are organized within the cache. Direct mapped is a simple and efficient organization [3]. The memory address of the incoming cache line controls which cache location is going to be used. In a direct mapped organization, the replacement policy is built-in because cache line replacement is controlled by the memory address. In many cases this design works well, but, because the candidate location is controlled by the memory address and not the usage, this policy has the potential downside of replacing a cache line that still contains information needed shortly afterwards. Any line with the same address modulo the cache size, will map onto the same cache location.

The fully associative cache design solves the potential problem of thrashing with a direct-mapped cache. The replacement policy is no longer a function of the memory address, but considers usage instead. With this design, typically the oldest cache line is evicted from the cache. This policy is called least recently used (see below as well). The downside of a fully associative design is cost, as additional logic is required to track usage of lines. The larger the cache, the higher the cost. Therefore, it is difficult to scale this technology to very large (data) caches. Luckily, a good alternative exists.

A set-associative cache design uses several direct-mapped caches. Each cache is often referred to as a set. On an incoming request, the cache controller decides which set the line will go into. Within the set, a direct-mapped scheme is used to allocate a slot in the cache. The name reflects the number of direct-mapped caches. For example, in a 2-way set associative design two direct mapped caches are used.

When the cache wishes to access a datum, it first checks the cache. The situation when the datum is found in the cache is known as a cache hit. The alternative situation, when the cache is consulted and found not to contain a desired datum, is known as a cache miss.

Since the cache has limited storage, it may have to eject some other entry in order to make room when a cache miss occurs. The heuristic used to select the entry to eject is known as the replacement policy. One popular replacement policy, the least recently used (LRU) policy, replaces the least recently used entry, respecting this way the temporal locality. More efficient caches compute the frequency of utilization against the size of the stored contents, as well as the latencies and throughputs for both the cache and the memory. Other replacement strategies are the least frequently used (LFU) and the first in, first out (FIFO). The LFU refers to the replacement of the datum in the cache which has least frequenlty been used, while the latter stands for the replacement of the datum which has been residing in the cache for the longest time. Another approach is the random replacement policy. Eventually, the optimal replacement strategy replaces the datum which will not be accessed for the longest time. It can be possible to implement this strategy in a real cache since it requires information about future cache references, yet, the strategy is only of theoretical value [4] .

When a datum is written to the cache, it must at some point be written to the memory as well. The timing of this write is controlled by what is known as the write policy. In a write-through cache, every write to the cache causes a write to the memory. Alternatively, in a write-back cache, writes are not immediately mirrored to the store. Instead, the cache tracks which of its locations have been written over. The data in these locations is written back to the memory when that data is evicted from the cache. For this reason, a miss in a write-back cache will often require two memory accesses to service: one to retrieve the needed datum, and one to write replaced data from the cache to the store.

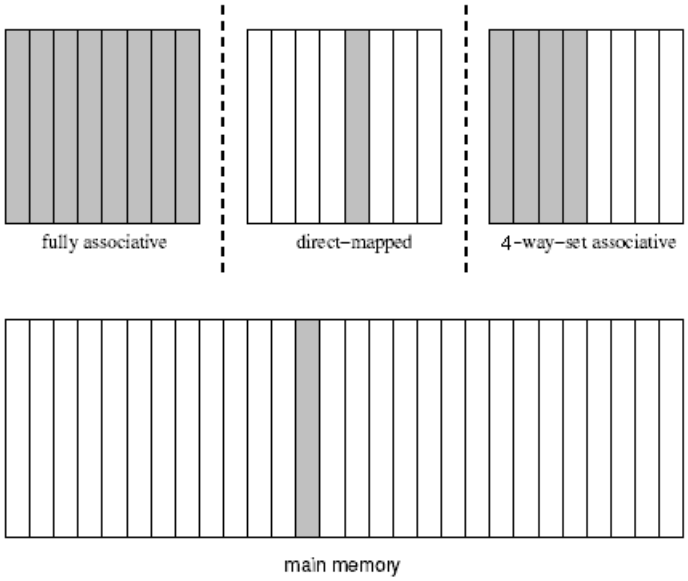

Figure 2.3. Cache organizations

2.3 Other Optimization Techniques

In order to reduce the memory barrier, the cache residence of memory of accesses should be improved. There are several techniques that are used in order to achieve this goal. For the scope of this thesis, we will focus on three basic loop optimizations and the data prefetch. A loop optimization acts on the statements which make up a loop, such as a **for** loop. Loop optimizations can have a significant impact for numerical codes, because as we have already pointed out that such codes spend a large percentage of their time within (computational) loops. On the other hand, with data prefetch data is moved into the cache prior to usage. Taking into account the largeness of the data sets under-

lying in numerical applications, data prefetch can have a significant benefit in the execution times of these applications. We will refer with more detail to both of these techniques in the following sections.

2.3.1 Loop Optimizations

A lot of compiler analysis and optimization techniques have been developed so as to make the execution of loops faster, since loop transformations play an important role in improving cache performance and in the effective use of parallel processing capabilities. We will focus on three common loop transformations which we will later apply to the synthetic benchmarks developed for this thesis; i.e. loop interchange, loop blocking and loop fusion.

Loop interchange is a technique in which the nesting of loops are interchanged so that access of data from memory is in order that they are stored. One major purpose of loop interchange is to improve the cache performance for accessing array elements. Cache misses occur if the contiguously accessed array elements within the loop come from a different cache line; loop interchange can help prevent this. For example, in the code fragment:

```
for j = 1 to n do
   for i = 1 to n do
      sum += a[i, j] ;
   end for
end for
```

Table 2.1.

loop interchange would result in:

```
for i = 1 to n do
   for j = 1 to n do
      sum+ = a[i, j] ;
   end for
end for
```

Table 2.2.

The benefit of this method is that when taking into account the programming language, it improves the stride. An array with stride 1 has elements which are contiguous in memory. Such strides are generally more efficient than non-unit stride arrays, due to the effects of caches. However, the programming language makes a difference. In the C programming language, the array elements from the same row are stored consecutively, namely in row-major order. On the other hand, FORTRAN programs store array elements from the same column together, called column-major. Thus the order of two loops in the first example is suitable for C program while the second example is better for FORTRAN.

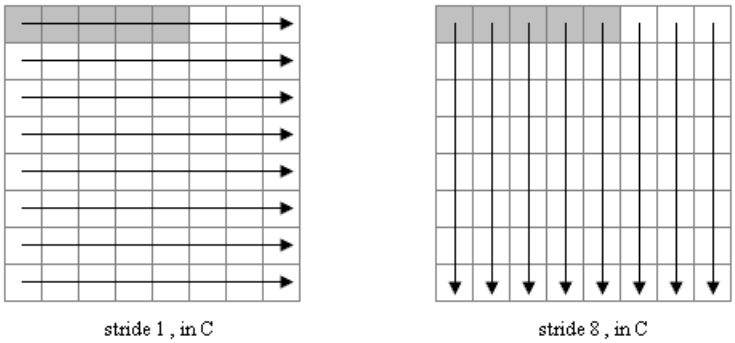

Figure 2.4. Access stride

Loop blocking is a technique that improves the temporal locality of data. Instead of operating on entire rows or columns of a matrix which may be too big to fit in the cache, blocked algorithms operate on sub-matrices or, data blocks. Then, the data that is being fetched into the cache can be used repeatedly before being swapped out. In the synthetic benchmarks, we will actually combine this technique with data prefetch. The following code fragment:

```
for i = 1 to n do
    for j = 1 to n do
        a[i, j] = b[i, j] ;
    end for
end for
```

Table 2.3.

after the loops are blocked becomes:

```
for ii = 1 to n by B do
    for jj = 1 to n by B do
        for i = ii to min(ii + B - 1, n) do
            for j = jj to min(jj + B - 1, n) do
                a[i, j] = b[i, j] ;
            end for
        end for
    end for
end for
```

Table 2.4.

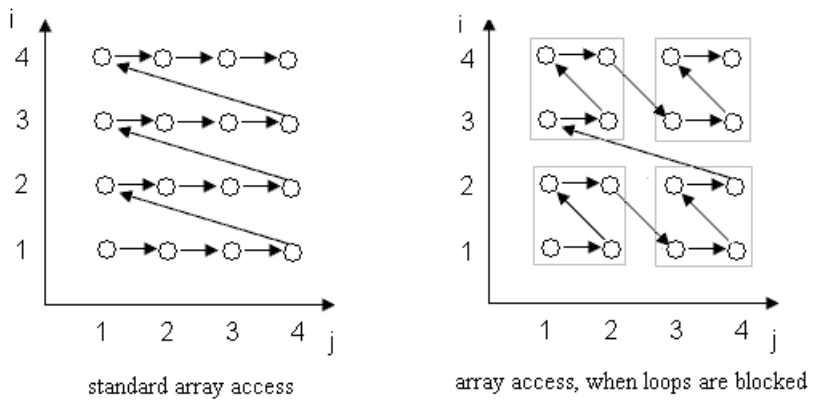

Figure 2.5. Array access patterns with and without loop blocking

Loop fusion is a another method to improve the temporal locality. When the same data is used in a task during seperate sections of code, it is better to bring them close to the same sections in the code. Then the data that is being fetched into the cache can be used repeatedly before being swapped out. The following code:

for $i = 1$ **to** n **do**
    a[i] += 1 ;
end for
for $i = 1$ **to** n **do**
    b[i] += 2 ;
end for

Table 2.5.

when the loops are fused will be transformed as:

for $i = 1$ **to** n **do**
    a[i] += 1 ;
    b[i] += 2 ;
end for

Table 2.6.

2.3.2 Data Prefetch

Prefetching is fetching data into cache before the processor actually needs it. Hopefully, it is then available by the time the processor needs it. Even if it has not arrived yet, it will help in reducing the processor stall time [5]. Therefore, thanks to prefetch, the memory latency can be hidden to a certain extent. Prefetching can be accomplished with compiler flags, via programmer intervention, or by hardware [6]. Software-based prefetching requires a special processor instruction which can be used by the compiler or programmer to issue the load from main memory. Having cache-lines longer than one word is an example of hardware-based prefetching: data brought into cache is accompanied by surrounding data; if the data exhibits spatial locality, then the surrounding data has usefully been prefetched. One of the main difficulties with prefetching is its lack of portability due to high machine dependence. Although long cache-lines support successful prefetching when the data exhibits spatial locality, long cache-lines also tend to increase conflict misses due to associativity problems. Also, support from the programming languages for application programmers to effectively use the hardware prefetch instruction is limited or non-existent [7]. Thus prefetching, though beneficial, has its share of concerns.

To hide the latency, the processor must perform sufficient other activities to allow time for the actual prefetching to occur. These activities may not be present in the application, or there are not enough other resources (for example, registers) available while the prefetch operation is in progress. Despite these potential drawbacks, prefetch is a powerful technique to improve application performance, and is worth considering as part of tuning the performance of an application [13].

Chapter 3

Implementation Infrastructure

3.1 Implementation Schema

Fetching data from main memory generally costs a lot. When speaking of numerical applications which utilize large data sets, data have to be fetched from the main memory continuously during execution, as typically the proportion of these data sets signifficantly exceeds the capacity of the cache. Therefore, in this thesis we consider the prefetching technique as a means of reducing the latency, by avoiding or reducing the time that the processor is waiting for data to arrive in the registers. There are many methodologies to implement the prefetching technique, normally depending on the architecture of the computer.

Since numerical applications order millions of millions Floating Operations per second (teraFLOPS), a trade-off in the use of the CPU for running a prefetching task would do more harm than good. It is obvious that for the prefetching to be beneficial, the lattest must not burden the calculations. A numerical application can be realized as a thread running on one core (which from now on will be referred to as main thread), while another thread running on another core will be considered for prefetching to the L2 cache the data needed during the calculations of the main thread. The latter will be referenced to as prefetch thread. Moreover, since the two cores share a common Level 2

cache, the latency could be reduced from CPU-memory levels to CPU-cache levels. In the following, we will discuss important aspects of the above schema.

3.2 Decisions over the Implementation Schema

The basic aspects of the implementation schema are the loop-blocking optimization, the prefetch optimization, the employment of threads instead of processes, the effectice scheduling in the case of multicore systems with more than two cores and the use of a ring buffer in the common address space of the two threads. We will discuss each aspect in detail in its respective subsection.

3.2.1 Loop-Blocking Optimization

In the benchmarks used in this thesis, prefetch will be software implemented as a thread (prefetch thread). This thread expects to be notified by the main thread in order to initiate the movement of data from memory towards the processor. In order to further analyze the implementation of the prefetch, we introduce a trivial computational schema that will be included in our benchmarks as well.

Let's assume an iterative loop that sums up the elements of an array. In order to be able to talk about prefetch, we have to apply the loop-blocking technique first (see figure below [13]). When the size of the array is larger than the size of the cache, data have to be fetched from main memory during the iterations of the whole array, resulting in main memory bandwidth. With the loop-blocking method, once the subarrays of data are smaller than the cache size, only the first time that the data are requested, will they be fetched from the memory. During all other iterations, the data will reside already in the cache. The bigger the number of iterations, the more the bandwidth will tend to cache bandwidth. The following figure, taken from [13], illutrates the above:

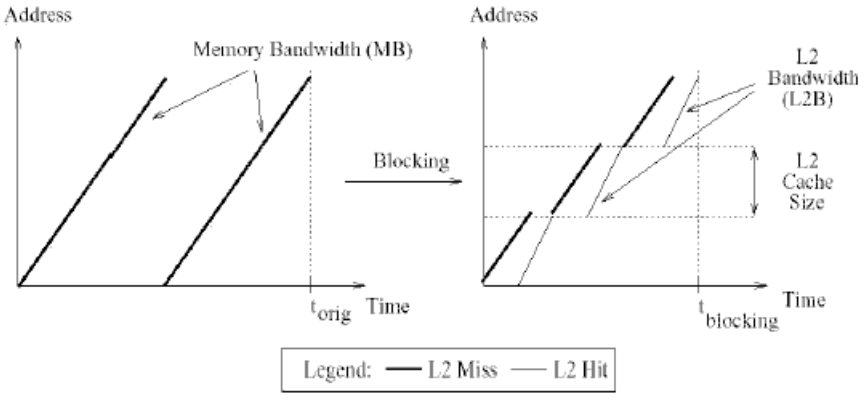

Figure 3.1. [13]: 1D Blocking Optimization

3.2.2 Prefetch Optimization

If we add prefetch to the schema, we avert the latencies of the first data subarray fetch from memory for all the subarrays except for the first one (see figure [13]). In iterative numerical codes that act on large data sets, this optimization can be very beneficial.

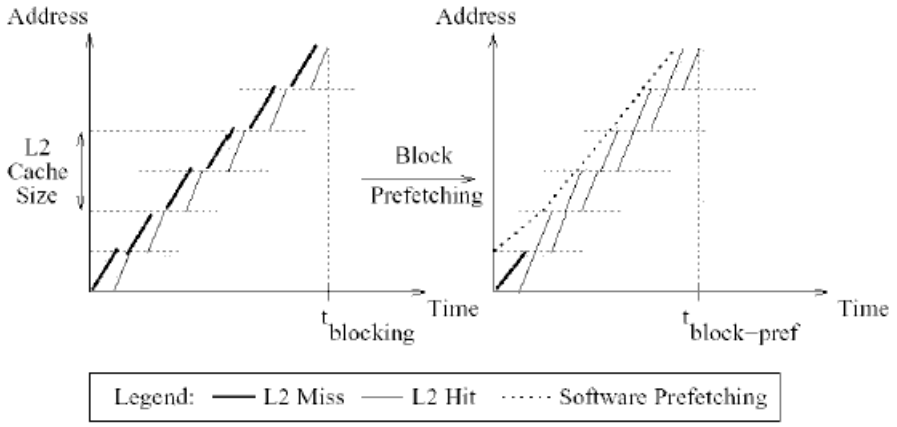

Figure 3.2. [13]: Prefetch Optimization

3.2.3 In UNIX: Threads over Processes

We mentioned that responsible for the prefetch of data in our benchmarks will be a prefetching thread. We will use the UNIX implementation of threads, which is called POSIX threads or Pthreads and adhere to the IEEE POSIX 1003.1c standard. In this section, we defend this choice over other implementation options.

Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system [14]. A process in UNIX is created by the operating system, and requires a fair amount of overhead. On the other hand, a thread in the UNIX environment exists within a process, while it has its own independent flow of control. It duplicates only the essential resources it needs to be independently schedulable. A thread is referred to as "lightweight process" because most of the overhead has already been accomplished through the creation of its process [14]. We can conclude therefore, that for the creation of a new task, in this case the prefetch task, a thread will cause less overhead than a process will.

Moreover, managing threads requires fewer system resources than managing processes. All threads within a process share the same address space, a property that we are using for the communication between the main and the prefetch thread in order to coordinate the flow of the prefetch. We are going to discuss thoroughly over this thema in a following section of this chapter. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication [14]. An utilization of shared memory among processes is more difficult to set up than among threads. From the above, it is obvious that thread management is easier, and most importanly more time efficient than process management.

3.2.4 Effective Scheduling for Multi-core Processors

In multi-core processors arises the issue of scheduling multiple threads or processes on multiple processors in a valid, but also time efficient manner. We will discuss over two known scheduling methods for resolving such situations; the gang scheduling algorithm and the processor affinity.

Gang scheduling is a scheduling algorithm that schedules related threads or processes to run simultaneously on different processors. Usually these will be threads all belonging to the same process, but they may also be from different processes. Gang scheduling is used so that if two threads or processes communicate with each other, they will all be ready to communicate at the same time. If they were not gang-scheduled, then one could wait to send or receive a message to another while it is sleeping, and vice-versa.

The ability to bind one or more processes to one or more processors is called CPU affinity. This feature, that has been long provided by operating systems such as Windows NT, has been lately included in the UNIX/Linux operating system as well [15].

The first benefit of CPU affinity is optimizing cache performance. Multiprocessing computers go through a lot of trouble to keep the processor caches valid. Data can be kept in only one processor's shared cache at a time. Otherwise, the processor's cache may grow out of synchonization, leading to the question, who has the data that is the most up-to-date copy of the main memory. Consequently, whenever a processor adds a line of data to its local cache, when a write occurs, all the other processors in the system also caching it must invalidate that data. This invalidation is costly. But the real problem comes into play when processes bounce between processors (effect known as the Ping-Pong Effect, see figure below); they constantly cause cache invalidations, and the data they want is never in the cache when they need it. Thus, cache miss rates grow very large. CPU affinity protects against this and improves cache performance [15].

	Time 1	Time 2	Time 3	Time 4
Process A	CPU 0	CPU 1	CPU 0	CPU 1
Process B	CPU 1	CPU 0	CPU 1	CPU 0

Figure 3.3. The Ping-Pong Effect

A second benefit of CPU affinity is a corollary to the first. If multiple threads are accessing the same data, it might make sense to bind them all to the same processor. Doing so guarantees that the threads do not contend over data and cause cache misses [15], property of high importance for multi-core processors with shared caches.

Commonly, in a dual-processor system, the specialized application is bound to one processor, and all other processes are bound to the other processor. This ensures that the specialized application receives the full attention of the processor [15]. In our case, this event ensures that the main thread will be run on one processor, while the prefetching thread will be run on a second processor. This way the independent and simultaneous execution of the two threads is guaranteed.

3.2.5 Inter-thread Communication

We mentioned before that the communication between the main and the prefetching thread will take place within their common address space. This is the fastest way in order to have two threads communicating, as other communicating methods such as the utilization of UNIX signals or the use of semaphores require the trafficing through the operating system and thus, they introduce a time overhead. For this reason, we will define in the common address space a ring buffer. A ring or circular buffer is a method of using memory, the features of which will be mentioned later in this section.

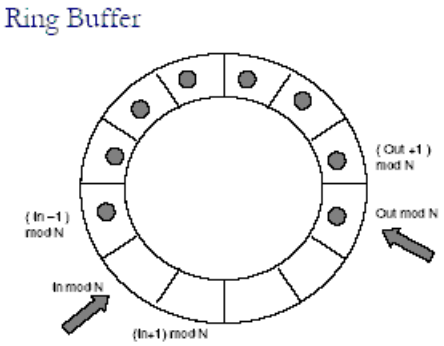

Figure 3.4. Ring buffer

The schema of the inter-thread communication that we used is explained in the following; the prefetch thread checks the ring buffer continuously for prefetch requests. The main thread will put a prefetch request in the ring buffer whenever it desires that a specific block of data must be retrieved from memory and at that moment the prefetch thread will extract and execute the request. Since we run the main and the prefetch threads on different cores, according to what we have demonstrated so far, the response of the prefetch thread to the "discovery" of a prefetch request in the ring buffer has to be spontaneous.

Since the data storage for the buffer should be large enough to hold as much data as the data source will produce while the user of the data is either working with the data or is otherwise not reading it, in our case a buffer able to hold one unit of data should be enough. However, because we want to be completely sure that the main thread will not waste a moment waiting for the prefetch thread to extract the prefetch request, we define a buffer of 10 units. Each unit of the buffer will have the size of a prefetch request, that from now on will be referred to as command.

We will discuss the fields that each command should consist of. Appart from the prefetch command, we want to define a command that will be able to kill the prefetch thread, intended to be used after all blocks have been prefetched. Therefore, because we want to have two types of commands, we include a field to indicate the type of the command (as integer variable). A prefetch command has to know the address of the first datum that is supposed to prefetch and moreover, since our benchmarks intend to examine schemes used in numerical applications, we will define a pointer to double as our starting-address field. Last but not least, we must include the length of the desired block to be prefetched, for the prefetch thread to know when to stop retrieving data from memory.

```
struct command {
        int type ;
        double *firstAddress ;
        int length ;
}
```

Table 3.1. Structure of a command (in C)

For access to the ring buffer by the two threads, we will introduce a writing and a reading pointer. The two pointers indicate where in memory the next data should be read or written. Each time data is read, the read pointer moves forward, going back to the beginning after reading the last thing in the buffer. Each time data is written, the write pointer moves forward, going back to the beginning after writing to the last position in the buffer.

If the read and write pointers point to the same spot in the buffer, that might mean that there is no data in the buffer (nothing has yet been written to the spot we next want to read from) or that the buffer is full (the next place we want to write already has in it data we have not yet read). As a way to make the distinction, we maintain a separate indication of whether there is data in the buffer, the number of elements in the buffer (number of commands).

```
struct commonAddressSpace {
        int readPointer, writePointer ;
        int numberofCommands ;
        struct command ringBuffer[10] ;
}
```

Table 3.2. Structure of the ring buffer (in C)

Putting it all together, the writing of a command in the ring buffer will be implemented according to the code fragment of the following table:

```
if ( numberofCommands < 10 ) {
        ringBuffer[writePointer] = command ;
        writePointer = (writePointer+1)%10 ;    //...modulo10 ;
        ++numberofCommands ;
}
```

Table 3.3. Writing a command in the ring buffer (in C)

The prefetch command will be formatted as:

```
void WritePrefetchCommand ( double *firstAddress , int length ) {
        /* define a new command */
        struct command anewCommand ;

        /* set the type of the command to "prefetch" type  and */
        /* assign the argument values to the respective fields of the command */
        anewCommand.type = "prefetch type" ;
        anewCommand.firstAddress = firstAddress ;
        anewCommand.lengt = length ;

       /*write the command to the ring buffer */
       /* code from Table 1.3. here */
}
```

Table 3.4. Structure of a prefetch command

On the other hand, the command intended to kill the prefetch thread will have the form:

```
void WriteKillCommand () {
      struct command anewCommand ;

      /* set the type of the command to "kill prefetch thread" type */
      anewCommand.type = "kill prefetch thread type" ;
      /* the filling of the rest of the fields is not required */

      /*write the command to the ring buffer */
      /* code from Table 1.3. here */
}
```

Table 3.5. Structure of command to kill the prefetch thead

3.2.6 Schematic Depiction of the Inter-thread Communication

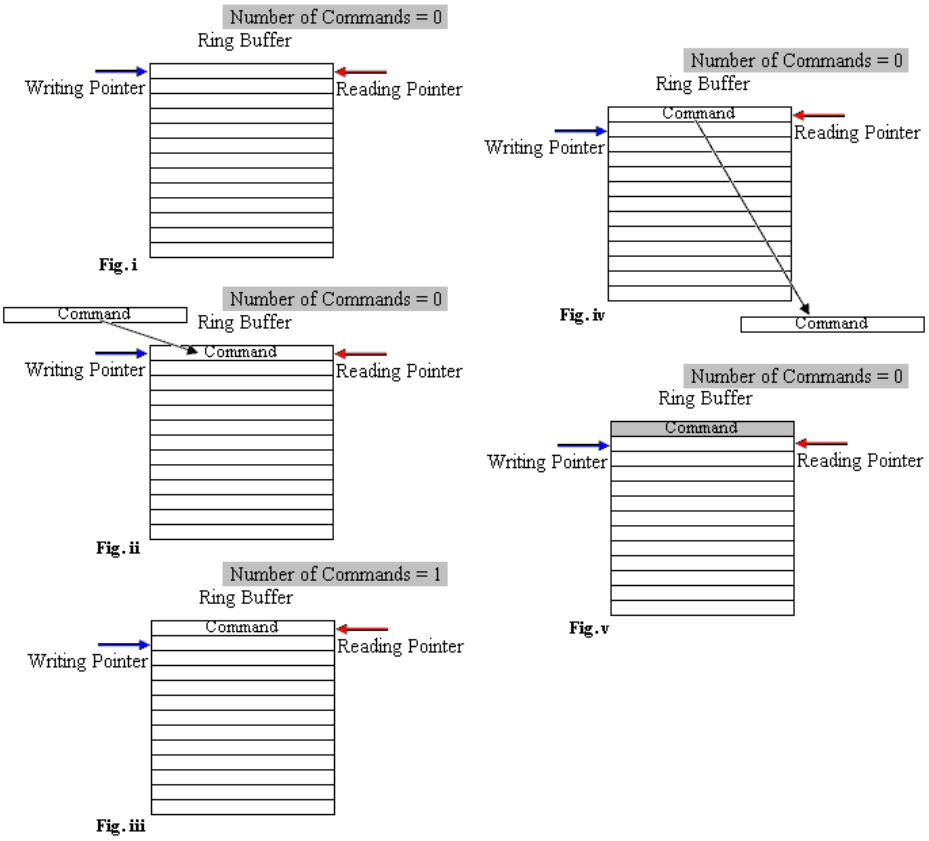

Figure 3.5. Inter-thread Communication

In the first figure the ring buffer is empty, i.e. no command has been written in the buffer yet. Moving to the following figure (Figure 1.5.ii), the main thread decides to prefetch a block of data from the memory or just kill the prefetch thread. Therefore, it creates an appropriate command and this command writes to the ring buffer either via the **void** WritePrefetchCommand (**double** *firstAddress , **int** length) function, or via the WriteKillCommand() command, respectively. Both functions move the write pointer to the next position of the buffer, while they increase the flag of the number of commands in the ring buffer (see code in Table 1.3, Figure 1.5.iii). The command is detected immediately, since the number of commands is now changed and the prefetch thread has been checking on this variable continuously. The reaction of the prefetch thread is to directly retrieve the command (Figure 1.5.iv). In the last figure (Figure 1.5.v), the read pointer is moved to the next position of the ring buffer and the retrieved command can be now overwritten.

3.3 Implementation of the Prefetch Thread

In order to have the whole inter-thread communication issue completely demonstrated, we give the code for the prefetch thread. At this point we have to mention that the same code is used in all the benchmarks, therefore, it is independent from the implementation of the main thread. The code for the prefetch thread is maintened in a seperate file that could be imported by future applications as well.

```
while (1) {

    if ( numberofCommands ) {

        --numberofCommands ;
        retrievedCommand = ringBuffer[readPointer] ;
        readPointer = (readPointer+1)%10 ;

        switch (type) {
          case "prefetch type": {
              firstAddress = retrievedCommand.firstAddress ;
              lastAddress = firstAddress + length ;
              for ( i=firstAddress ; i<lastAddress ; i+=8 )
                  dummySum += *i ;
              break ;
          }
          case "kill prefetch thread type": {
              pthread-exit(NULL) ;
              break ;
          }
        }
    }
}
```

Table 3.6. Prefetch Thread Code

It is obvious from the combination of the while-if condition that the prefetch thread will constantly check whether the number of commands is increased, event that means that a command has been written to the ring buffer. First action taken in case a command has been indeed written to the ring buffer is to reduce by one the number of commands, so that the main thread is be able to write another command, with simultaneous shifting of the read pointer by one position. Within the switch loop there takes place the actual reading of the command; at first is the type of the command retrieved in order that the correct case body is choosen. In case the command is "prefetch type", then the contents of the data contained in the addresses between the address of the first datum of the block to the last datum by 8, are summed up in a dummy variable. The reason why data are prefetched 8 by 8 doubles (or 64 by 64 byte) is that data are fetched from memory in lines and the double-core processor that we have used doing our experiments has 64 byte cache line size.

Moreover, for the data to be retrieved it is not enough that we load a variable with their value and later overwrite this variable, as this action will be removed by the compiler optimization as useless for the program. This is the reason why we finally sum up the contents of the data in a variable and indeed a global one; because we do want that this action actually takes place and it is not removed by the compiler.

Chapter 4

Synthetic Benchmarks

Most of the CFD codes on the market today feature a powerful algebraic solver based on multigrid methods. Multigrid methods in numerical analysis are a group of iterative algorithms for solving differential equations using a hierarchy of discretizations. Multigrid methods are among the most attractive algorithms for the solution of large sparse systems of equations that arise in the solution of elliptic partial differential equations [5]. They can be among the fastest solution techniques known today and as expected, the performance of CFD codes based on such methods could also be significantly improved.

For the scope of this thesis, we develop three synthetic benchmarks. Two are

intended to perform basic operations encountered in real iterative solvers, like solvers based on multigrid methods; i.e. addition and multiplication. They consist of small computational kernels operating on large 1D or 2D arrays, which are structures typically used in iterative solvers. The actual codes of real iterative solvers can be very complex, while the benchmarks simulate basic operations from iterative solvers but are relatively simple. Therefore, with these benchmarks we are comfortably allowed to investigate a potential gain of the the prefetch technique on a multicore system, while we could (approximately) scale this gain for real applications as well. The third benchmark is an implementation of a real iterative method, the Jacobi method and hence, a potential performance gain can be valued directly. All three benchmarks are thoroughly explained given their codes and in figures, in the three following sections. In the last section we discuss generally over the applications of CFD.

4.1 Addition and Multiplication Benchmarks

The first benchmark is a program that iteratively sums up the elements of a large array of doubles. The iterative summation of a large portion of elements is the foundation stone of every typical numerical code. For example, let us consider the red-black Gauss-Seidel as a smoother in the multigrid method. The standard Gauss-Seidel repeatedly performs one complete sweep through the grid from bottom to top updating all the red nodes and then performs another complete sweep updating all the black nodes. A red node can be updated for the i^{th} time by getting the mean average of its neighbouring blacks that have been updated for the $(i-1)^{\text{th}}$ time; a black can be updated for the i^{th} time by getting the mean average value of its neighbouring reds that have been updated for the $(i-1)^{\text{th}}$ time [5].

If we consider thus, that the updating of a single node for a single a moment requires four addition operations, the volume of the nodes and the iterativeness of the procedure, it is clear that a huge number of additions will take place at a full execution of the Gauss-Seidel. Therefore, it makes sense that we develop as first benchmark, a program that repeatedly adds the elements of a large array of data. In the following, we will give the code in stages, at each stage having optimized the code for the extraction of more performance.

4.1.1 Naive Code

In the naive implementation of the addition benchmark, the elements of a 1D array of doubles are summed iteratively, as illustrated in the Table below.

```
for iter=1 to ITERATIONS do
   for i=1 to SIZE do
       sum+=a[i] ;
    end for
end for
```

Table 4.1.

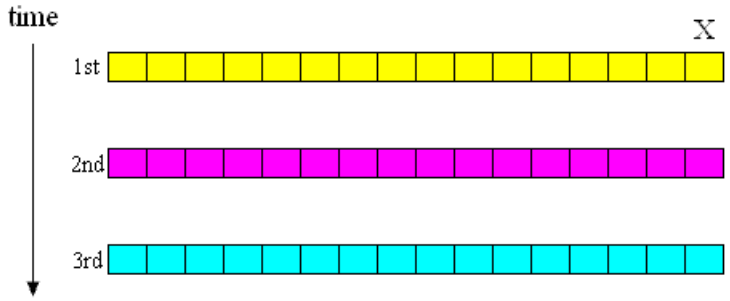

Figure 4.1. Naive Addition Benchmark

In the naive implementation of the second benchmark, two 2D arrays are multiplied in a simple way; i.e. the first row of the multiplicand array is consecutively multiplied with the columns of the multiplier array, the second row is multiplied with the columns of the multiplier array etc. The code is given below:

```
double a[M,N], b[M,K], c[K,N] ;

 for i=1 to M do
   for j=1 to N do
      for k=1 to K do
         a[i,j]+=b[i,k]*c[k,j] ;
      end for
   end for
end for
```

Table 4.2.

We mentioned before that the smallest block of data moved in and out of a cache is a cache-line. A cache-line holds data that is stored contiguously in memory. Typically,

when a datum is to be retrieved from the memory, the hardware fetches the whole line into the cache. Although many hardware systems prefetch the neighbouring cacheline(s) when a particular cache-line is fetched as well [5], for this thesis we will accept that only a single cache-line is fetched. Thus, it is advantageous to lay contiguously in memory the data that we are going to use in succession, because then an improvement of the spatial locality will be ensured. For this reason, we are going to map all the 2D matrices into 1D memory addresses, as illustrated below:

```
double *a, *b, *c ;

/* space assignment with malloc */

 for i = 1 to M do
   for j = 1 to N do
      for k = 1 to K do
         a[i*N+j] += b[i*K+k]*c[k*N+j] ;
      end for
   end for
end for
```

Table 4.3.

To further improve performance, the programmer can lay the data in memory according to the access pattern, technique known as data laying. Then, when a cache-line is brought into the cache, it also effectively prefetches the data that is to be needed in the near future. If elements in a data block are laid by access, spatial locality is improved. In this case, since we use k as innermost loop and the program is written in C language where array elements are stored in row-major order, the access stride for array b will be 1, while the access stride for array c will be equal to the cache-line size (that is 64), as illustrated in the following figure:

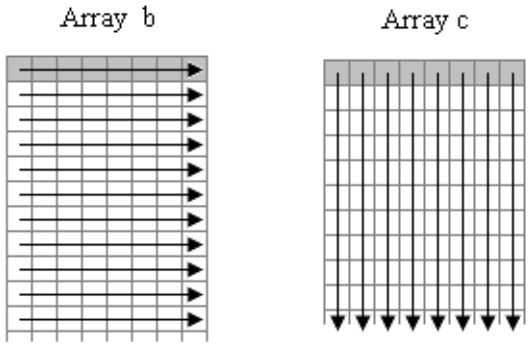

Figure 4.2. Stride of Naive Matrix Multiplication

Clearly, while array b is accessed in an optimal way, as far as exploiting the hardware cache-line prefetch is concerned, the way array c is accessed is not benefiting at all from the hardware prefetch. In case that array c is large such as not to fit in the cache, then the access pattern of array c will "cost" a lot at the execution of the code. We are going to change the access pattern of array c so as to benefit from the hardware cache-line prefetch, without spoiling the stride of array b in the following section.

4.1.2 Code with blocked loops

Assuming that the array is too large to fit in the cache, the data of the lower part of the array are no longer in the cache when the elements of the upper part of the array are summed up, because they been replaced by the latter. Hence, the data must be reloaded from the slower main memory into the cache again. In this process newly accessed elements replace the elements of the upper part of the array in the cache, and as a consequence they have to be loaded from the main memory once more at the time of their next update. The performance degrades further when the array size increases. This performance bottleneck at memory is also seen in most other scientific applications [5].

In general, the cost of floating point operations is rapidly decreasing. Moving data is what makes computing expensive. Although there are different ways of organizing on-chip and off-chip memory, programmers will have to learn that reasonable performance of a system could only be expected with programs being aware of the reduced performance of the off-chip memory access [5]. In the following, we attempt to avert this bottleneck by employing loop optimizations and the prefetch optimization mentioned in the previous chapter.

Wherever the loop-blocking technique is included, the code operates on a submatrix called data block, in contrast with the operation on a whole matrix that may too big too fit into the cache. As discussed in the above section, such case is responsible for a memory performance bottleneck. In this version of the benchmarks, instead of operating on the entire array iteratively, the code operates iteratively on subarrays (array blocks) that are able to fit into the cache. Therefore, each data subarray is thoroughly used before it is evicted from the cache. From memory's point of view and in comparison to the naive case, operating on small subarrays means that each subarray is fetched from the main memory once and then thoroughly used while it resides in the cache, while in the naive case, a portion of data has to be fetched to the cache as many times as the iterations. The respective code for the addition benchmark is given in the table:

```
for ii=1 to SIZE by ablock
  for iter=1 to ITERATIONS
    for i=1 to min(ii+ablock, SIZE)
      sum +=a[i] ;
    end for
  end for
end for
```

Table 4.4.

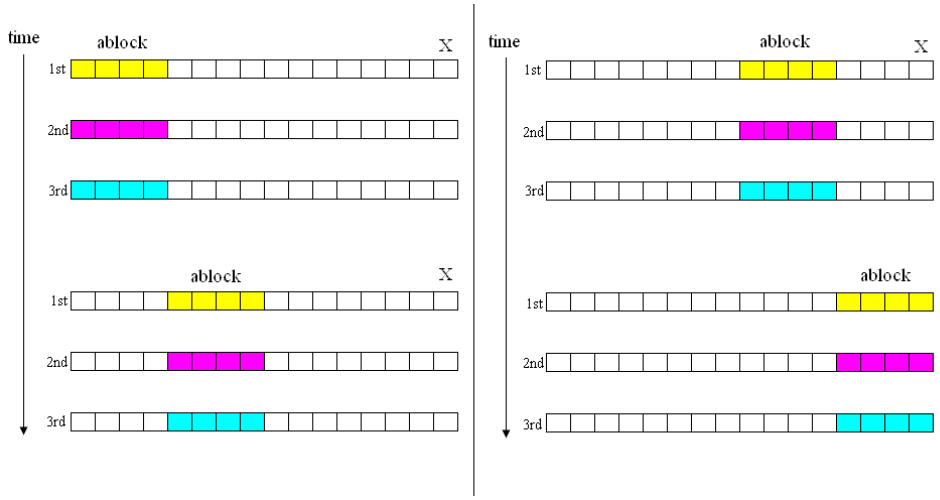

Figure 4.3. Loop-blocked Addition Benchmark

As for the multiplication benchmark, we introduce the division of the M and K dimensions into blocks of iblock and kblock sizes in respect. This potential is the only feasible according to the loop-blocking technique, as our purpose is to have index j as innnermost loop, so as to change the "bad" stride of array c to unit stride. The respective code is:

```
for ii = 1 to M by iblock do
  for kk = 1 to K by kblock do
    for i = ii to min(ii+iblock, M) do
      for k = kk to min(kk+kblock, K) do
        for j = 1 to N do
          a[i * N + j] += b[i * K + k] * c[k * N + j] ;
        end for
      end for
    end for
  end for
end for
```

Table 4.5.

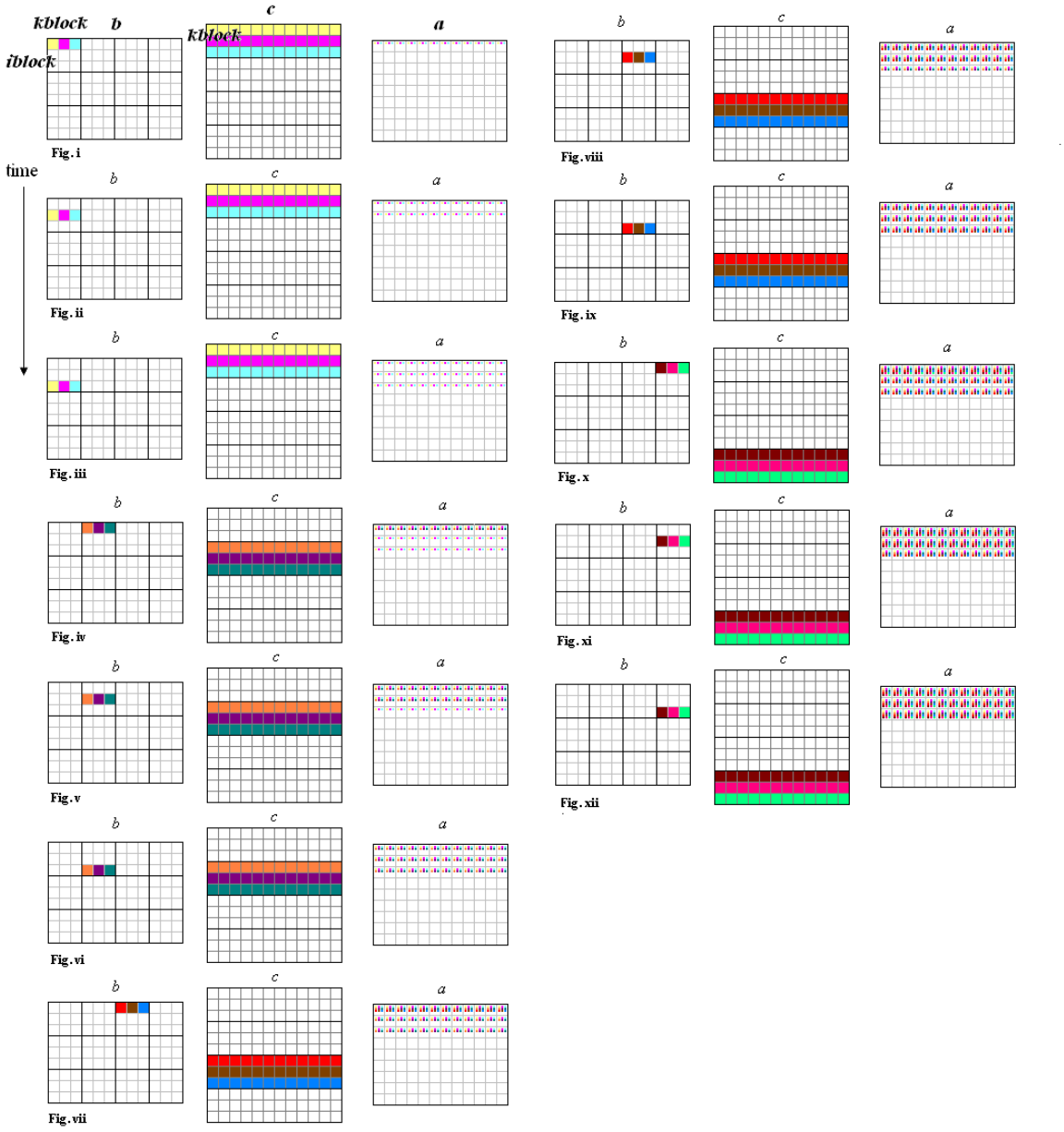

Figure 4.4. Loop-blocked Matrix Multiplication

Indeed the stride of array c will be changed to unit stride, while the stride of array b will be different than before but still the elements will be accessed in a column-major way, which is the most efficient way to access data from C programs that are stored consecutively in memory. There follows the illustration of the stride for the loop-blocked implementation of the matrix multiplication benchmark:

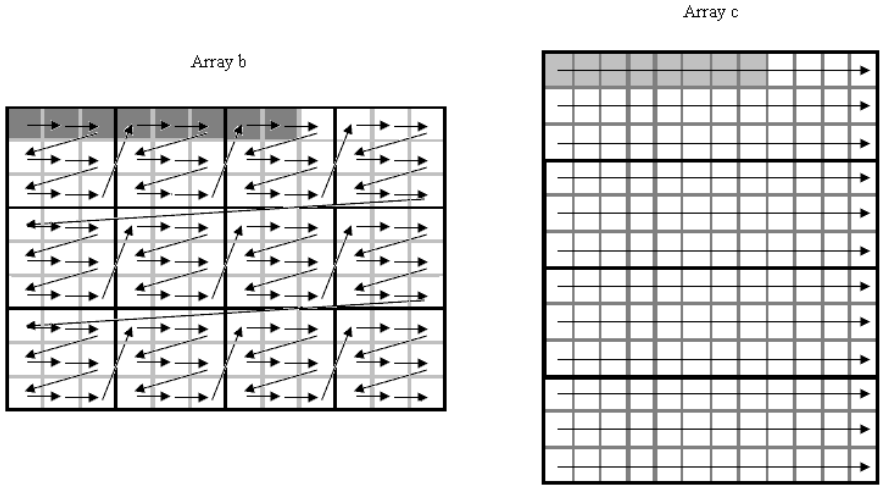

Figure 4.5. Access Pattern of Loop-Blocked Matrix Multiplication

4.1.3 Code including the Prefetch Thread

In this implementation we try to hide the latency caused by the fetching of an array block from the main memory. For this purpose we are going to use a prefetch thread handled by a second code. We already demonstrated that when two cores are available, a thread created by a process will be binded to the other core from the one that itself is run. Considering the main code as a process bound to one core, a prefetch thread created by this process will then be bound to the second core. Therefore, when a multicore machine is used, there is no burden in the main procedure by the handling of a prefetch thread.

The idea is that the prefetch thread fetches each time the future array block to be used into the cache, so that the main process (or main thread) at the time of execution finds this block in the cache and therefore, does not need to fetch it itself from the main memory. Only the first array block is fetched from the main memory by the main process itself. Moreover, the more the subarrays, the higher the benefit from the prefetch. The code for the addition benchmark can be illustrated as:

```
for ii=1 to SIZE by ablock
  begin

    if ii < SIZE-ablock
       prefetch from a[ii+ablock] to a[ii+2*ablock] ;
    end if

    for iter=1 to ITERATIONS
      for i=1 to min(ii+ablock, SIZE)
        sum+=a[i] ;
      end for
    end for

  end begin
end for
```

Table 4.6.

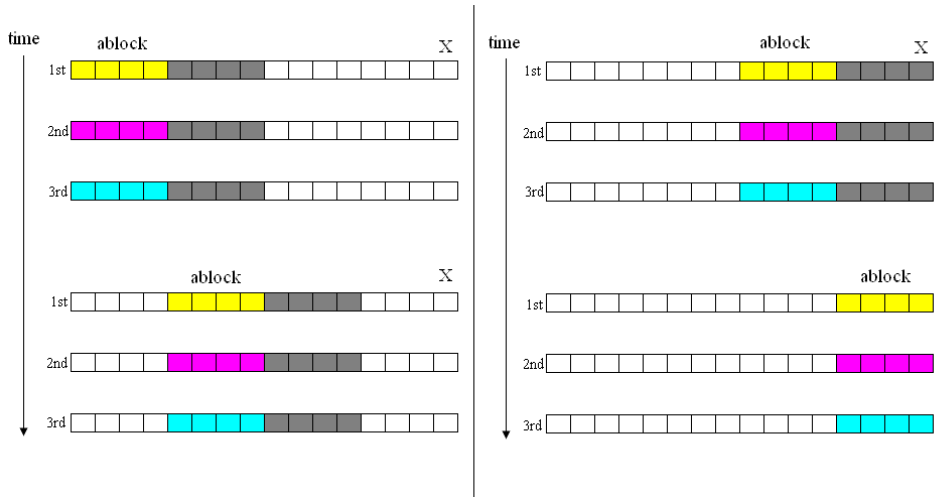

Figure 4.6. Loop-blocked with Thread Prefetch Addition Benchmark

On the other hand, the code for the matrix multiplication benchmark will be:

```
for ii = 1 to M by iblock do
  for kk = 1 to K by kblock do
    begin

      if kk < K − kblock
          prefetch from c[N ∗ (kk+kblock)] to c[N ∗ (kk+2 ∗ kblock)] ;
      else if kk = K − kbclock and ii < M − iblock
              prefetch from c[0] to c[N ∗ kblock] ;
          end if
      end if

      for i = ii to min(ii+iblock, M) do
         for k = kk to min(kk+kblock, K) do
            for j = 1 to N do
               a[i ∗ N + j] += b[i ∗ K + k] ∗ c[k ∗ N + j] ;
            end for
         end for
      end for

    end begin
  end for
end for
```

Table 4.7.

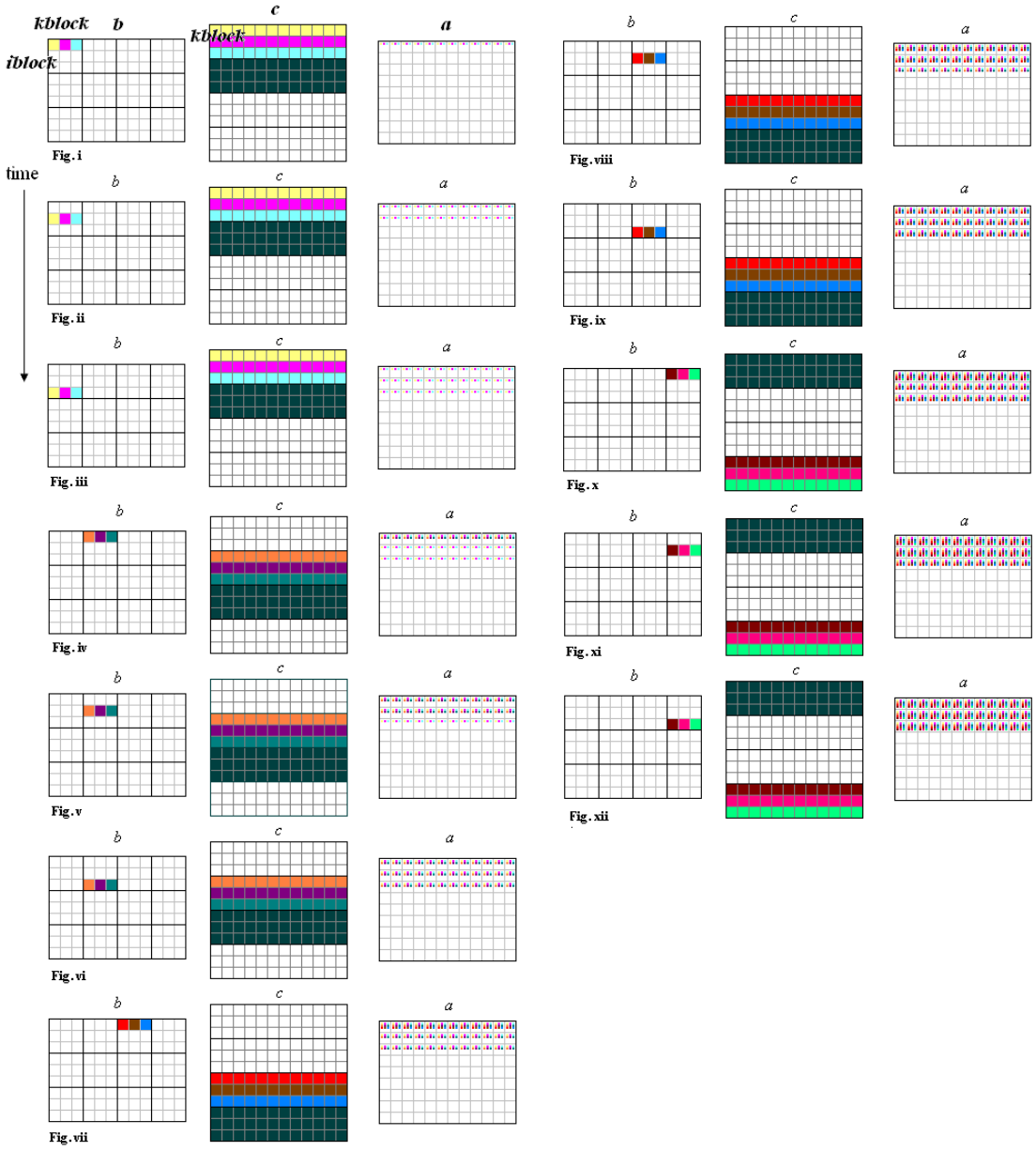

Table 4.8. Loop-blocked with Prefetch Thread Matrix Multiplication

4.1.4 Prefetch Interleaved into the Code

In this section we introduce prefetch embedded in the code. There is neither going to be a second thread to do data prefetch anymore, nor a second core. The idea is to "plant" in the main process or thread prefetch instructions. Then, at the time that a block is calculated, the data that constitute the next block stored in memory, the block that will be brought up afterwards (we have already discussed how we store our data in the memory in a consecutive way, so as to improve the spatial locality) are prefetched into the cache. Naturally, this type of prefetching causes an overhead in the main prcess.

In order to keep low this overhead, we take advantage of the cache-line hardware prefetch (when a datum is to be fetched to the cache, the whole line that it belongs is fetched as well) and prefetch a datum or array element every 64 byte, which is the cache-line size. Then, the overhead should be a lot less (around 8 times less, if the arrays consist of doubles, like in our case) comparing to the overhead that would be caused in case we prefetched all the data of the consecutive block. Moreover, the smaller the chunk of data that we prefetch each time that we come along a prefetch instruction in the code, the less the overhead, as this would mean more overlap for the main process. Last but not least, the prefetch instructions should be "planted" in the code in a sparse way, again so that the main process is not often distracted by the prefetch instructions and therefore, the overhead is not too high.

We present the code for the Addition Benchmark that includes embedded prefetch instructions. In this benchmark, at the time that a block is summed up repeatedly there take place ablock $*$ ITERATIONS operations. Moreover, since each block (subarray) is of size ablock and we explained in the above that we only need to prefetch a double every 64 byte or 8 doubles, it is enough that we prefetch in total $\frac{\text{ablock (doubles)}}{8\,\text{(doubles)}}$ or (if we simplify the doubles) $\frac{\text{ablock}}{8}$ doubles. Therefore, in the Addition Benchmark, a double must be prefetched every $\frac{\text{ablock} * \text{ITERATIONS}}{\frac{\text{ablock}}{8}}$ or by simplifying the ablock, ITERATIONS $* 8$.

Preftstep is the rate at which we should prefetch a double, namely the result ITERA-

TIONS ∗ 8. Moreover, we define a count variable that will be used as a counter for counting the summing operations that intermediate between two operations at which we prefetch a fraction of data . Last but not least, the prefetched doubles are stored into a dummy variable, which is defined as a global one, or else it is removed by the optimization of the compiler as useless. In this case, no prefetching would take place.

```
prefstep = ITER.*8 ;

   for( ii=0 ; ii<X ; ii+=ablock ) {
        offset= 0 ;

        for( iter=0 ; iter<ITER. ; iter++ ) {
            mini= min(ii+ablock, X) ;

            for ( i=ii ; i<mini ; i+=prefstep ) {
                  if ( ii<X-ablock)
                        dummySum= *(a+ii+ablock+offset) ;
                  for ( count=i ; count<i+prefstep ; count++ )
                        doubleSum += a[i] ;
             }
         }
    }
```

Table 4.9.

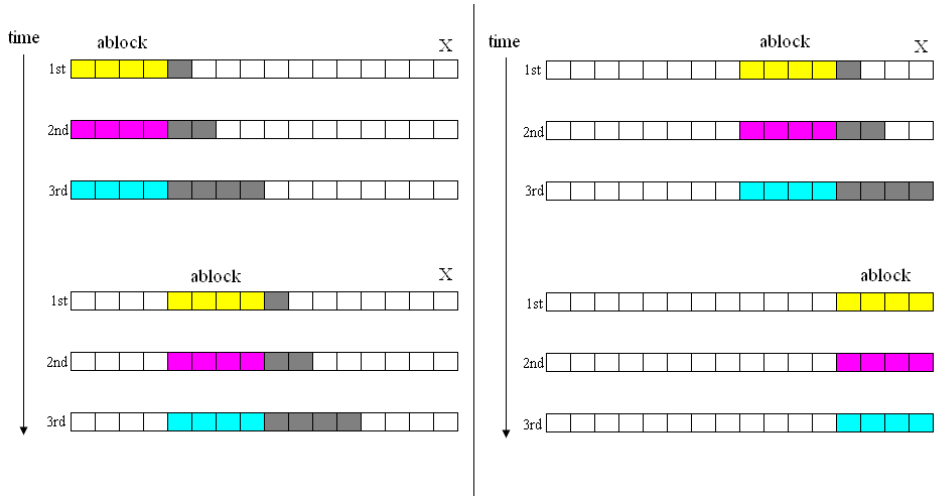

Figure 4.7. Addition Benchmark with Interleaved Prefetch

We give the interleaved code for the Matrix Multiplication Benchmark. First of all, for every multiplication between blocks we have iblock $*$ kblock $*$ N operations or iblock $*$ kblock operations including N operations each. The second notation holds if we consider that each element of an iblock $*$ kblock block of array b is multiplied with a whole row of array c. Since one row of array c contains N elements, the total number of operations will be iblock $*$ kblock $*$ N. On the other hand, the number of doubles that we have to prefetch, taking into account the fact that we only need to prefetch a double every 64 byte or 8 doubles, is kblock $*$ N / 8. If we make the simplifications, we notice that we have to prefetch N / (iblock $*$ 8) doubles every time an element of an iblock $*$ kblock is multiplied with a row of array c.

Besides the offset that will go over the data of the data block to be prefetched 8 by 8 (doubles), we define the sublineOffset that will be our flag of how many doubles to prefetch at every moment. Therefore, we are going to prefetch data as long as the offset of the prefetched data is smaller than this sublineOffset.

```
for ( ii=0 ; ii < M ; ii+=iblock )
   for ( kk=0 ; kk < K ; kk+=kblock) {
      offset = 0 ;
      sublineOffset = (N/iblock) ;

      for ( i=ii ; i < min(ii+iblock, M) ; i++ )
          for ( k=kk ; k < min(kk+kblock, K) ; k++) {

              if ( kk < K-kblock ) {
                    dummySum += *( c+N*(kk+kblock)+offset ) ;
                    offset += 8 ;
                    while( offset <= sublineOffset ) {
                              dummySum += *( c+N*(kk+kblock)+offset ) ;
                              offset += 8 ;
                     }
                     sublineOffset += (N/iblock) ;
               }
              else if ( kk==K-kblock && ii < M-iblock ) {
                    dummySum += *c ;
                    offset += 8 ;
                    while ( offset <=sublineOffset) {
                              dummySum += *( c+offset ) ;
                              offset += 8 ;
                    }
                    sublineOffset += (N/iblock) ;
              }
              for ( j=0 ; j < N ; j++)
                   a[i*N + j] += b[i*K + k]*c[k*N + j] ;
      }
  }
```

Table 4.10.

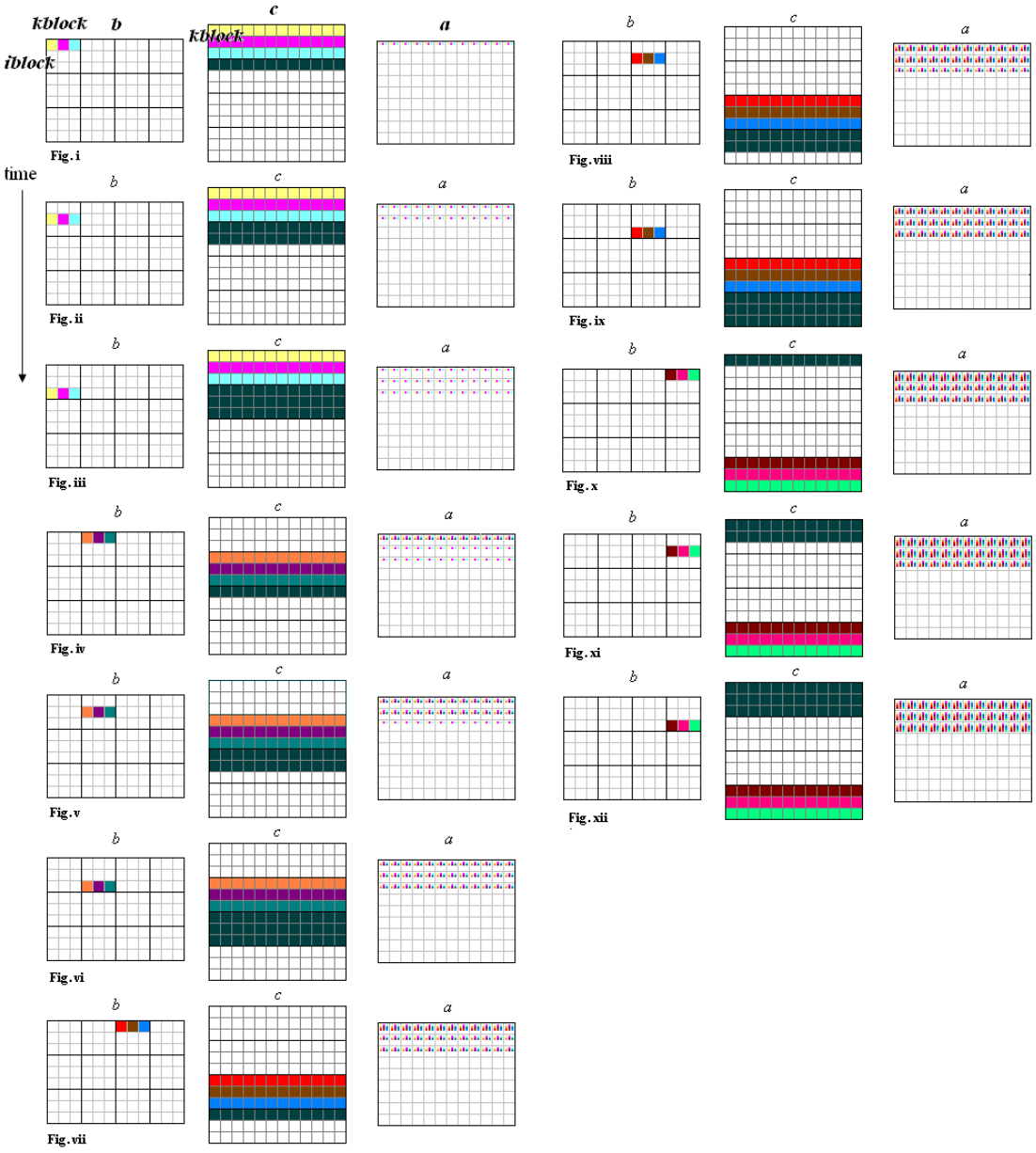

Figure 4.8. Matrix Multiplication with Interleaved Prefetch

4.2 Jacobi Method Benchmark

The jacobi method is used in real iterative solvers, therefore it is important that we investigate a potential gain arising from the use of the loop-optimization techniques and the prefetch optimization on the respective benchmark. In this benchmark, we use two matrices, one containing the initial values that will be used for the updating of the values of the other matrix. Typically, a matrix in the jacobi method is updated two, three or four times. In this benchmark we will update a matrix three times.

We present the code in C; array b will be the array with the initialized values and therefore the one that will be only once updated, while array a will be the array with the non-initialized values that will be twice updated. Both matrices naturally share the same dimensions, that is N for the row-major order dimension and M for the column-major order dimension.

Moreover, in this benchmark we introduce 1D blocking. Both matrices will be divided in respect to their row-major order dimension in iblocks. The reason for this choice arises from the fact that the innermost loop is the one with j as index, therefore the stride is already unit (optimal for C programs). Each time we will fetch to the cache blocks of size iblock*M, size that we will have to make sure that it is smaller than the size of the cache. Then these blocks will be repeatedly used for the first relaxation of the respective iblock of array a, the first relaxation of the respective iblock of array b and the second relaxation of the iblock of array a. In case that we include the prefetch thread as well (see **if**(use-prefetch) condition loop), then the latency of each first fetch of a block from the memory will be skipped, as the block will have already been fetched into the cache by the prefetch thead. This property has to do with the fact that the prefetch thread when run on a second core, which is our case, adds no overhead to the process or main thread that it came from.

```
/* relax for the first time the first two rows of matrix a */
for ( i=1 ;  i<3 ; i++ )
   for ( j=1 ; j<M-1 ; j++ )
       relax( a[i*M+j] ) ;

/* relax for the first time the two rows after the first two rows of matrix a */
/* relax for the first time the first two rows of matrix b */
for ( i=3 ; i<5 ; i++ )
   for ( j=1 ; j<M-1 ; j++ ) {
       relax( a[i*M+j] ) ;
       relax( b[((i-2)*M+j] ) ;
   }

/* relax for the first time the two rows after the first four rows of matrix a */
/* relax for the first time the two rows after the first two rows of matrix b */
/* relax for the second time the first two rows of matrix a */
counter= 1 ;
for ( ii=-1 ; ii<N-1 ; ii+=iblock ) {

     endofiblock= counter*iblock-1 ;
     counter++ ;

     if (use-prefetch)
         if ( ii<N-ibclok-1)
             prefetch from b[M*(ii+iblock-1)] to b[M*(ii+2*iblock-1)] ;

    for ( i=ii ; i<min(endofiblock, N) ; i++ ) {
     /* the first four rows have already been once or twice relaxed */
       if ( i>=5)
             for ( j=1 ; j<M-1 ; j++ ) {
                   relax( a[i*M+j] ) ;
                   relax( b[((i-2)*M+j] ) ;
                   relax( a[((i-4)*M+j] ) ;
             }
    }
}

/* relax for the first time the two last rows of array b */
/* relax for the second time the four last rows of array a */
i=N-2 ;
for ( j=1 ; j<M-1 ; j++ ){
      relax( b[((i-1)*M+j] ) ;
      relax( b[i*M+j] ) ;
}

for ( j=1 ; j<M-1 ; j++ ) {

      relax( a[((i-3)*M+j] ) ;
      relax( a[((i-2)*M+j] ) ;
      relax( a[((i-1)*M+j] ) ;
      relax( a[i*M+j] ) ;
}
```

Table 4.11.

```
// case relax( a[...] )
{
        a[i * M + j] = 0.25*( b[(i - 1) * M + j]+b[(i + 1) * M + j]+
                                                b[i * M + (j - 1)]+b[i * M + (j - 1)] ) ;
        break ;
}
// case relax( b[...] )
{
       b[i * M + j] = 0.25*( a[(i - 1) * M + j]+a[(i + 1) * M + j]+
                                                a[i * M + (j - 1)]+a[i * M + (j + 1)] ) ;
       break ;
}
```

Table 4.12. relax(...)

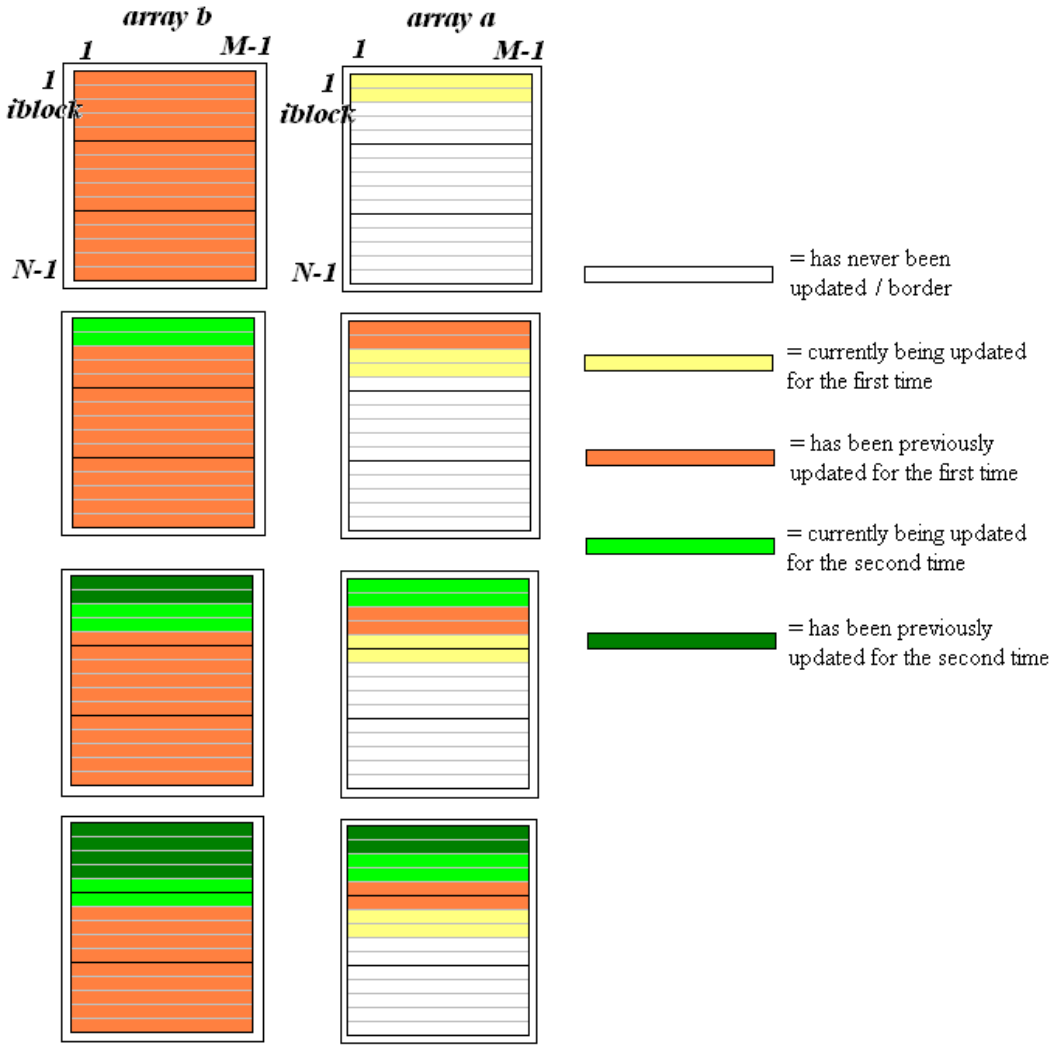

Figure 4.9. Loop-blocked Jacobi Benchmark

4.3 Computational Fluid Dynamics

The methologies in CPU design, memory organization, as well as the other optimization techniques have a significant impact on the performance of computational fluid

dynamics applications. The reason why will be explained in the last paragraph.

Computational fluid dynamics (CFD) is a computer modelling technique that can simulate the flow of liquids, gases and particles, reactions, heat transfer and other phenomena [8]. From its origins as a tool primarily for use in heat transfer, usage of computational fluid dynamics has spread to a wide range of other fields, such as aerodynamics and hydrodynamics, plasma instability studies in fusion, biofluid mechanics, species studies in quantum chemistry, geophysics, atmospheric flows, and so on. Marine scientists and other non-aeronautical researchers have shared the frustrations of turbulence modelling, accurately simulating transient phenomena fundamental to viscous flows, or even the inadequacies of background meshes to resolve the intricacies of small-scale effects [9].

Moreover, CFD offers a number of significant advantages, including cost effectiveness, risk reduction and safety. As far as the cost effectiveness is concerned, CFD allows testing of a large number of variables without modifying existing plants. Secondly, CFD can predict performance at any scale, thereby minimising the risk inherent in designing large-scale plants and reducing the number of pilot stages required to scale-up. Last but not least, CFD is particularly useful in simulating conditions where it is not possible to take detailed measurements [8].

CFD uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. It is one of the most important applications areas for high-performance computing [10]. The idea of high-performance computing is based on the fact that the process of solving a problem usually can be divided into smaller tasks, which may be carried out simultaneously with some coordination. As we mentioned before, in multicore systems each core allows the indepedent handling of a thread. In iterative versions of CFD applications that implement numerical methods, the computational kernels under specific circumstances can be considered as independent tasks. Therefore when a CFD application is able of being properly split up, it's performance on a multi-core system can be signifficantly improved.

Chapter 5

Results

In order to test the benchmarks, we used two machines: the Core Duo processor (nicknamed Yonah) and the Core 2 Duo processor (nicknamed Woodcrest) by Intel. In the appendix we provide the characteristics of each processor. Each time we measured one proportion against the bandwidth of the benchmark on a specific machine. Since the benchmarks have more than one proportions that could be alternated, each time we alterated one proportion, while kept the others on a fixed value. As for the bandwidth of the benchmarks, this is actually the bandwidth to the register file of the CPU, i.e. L1 to Registers, as this is the only available bandwidth to the benchmarks.

5.1 Addition Benchmark

In the addition benchmark there are three variable proportions (parameters in the code); the size of the array, the size of the block that we will use in order to "divide" the array for the loop-blocking technique and the number of the times that the process will be repeated, i.e. the number of iterations. As long as the size of the array is smaller than the size of the cache, which implies that the array can fit completely into the cache, we expect to get the maximum bandwidth of the benchmark. In the figure below, we present the schema of this benchmark:

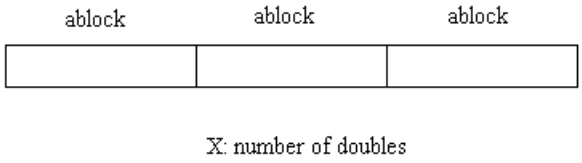

Figure 5.1. Array addition schema

Moreover, the bandwidth, while the array size does not exceed the size of the cache, should be same, both when no prefetch is included and when we introduce the prefetch thread. Since the prefetch thread runs on a second core simultaneously and independently from the main thread, we expect that it is not going to cause any overhead to the main process/thread. The only overhead that the prefetch thread can cause is at the time of its creation or killing by the process that created the prefetch thread on the first place. However, this overhead does not affect our measurements, since we only measure the execution time of the computational loops of the benchmarks and not the overall execution time of the program.

In all our measurements over the addition benchmark, we are going to fix the number of iterations to the smallest possible; i.e. two iterations. We do this, because this is the best way to demonstrate a possible performance gain from prefetch. We discussed how prefetch contributes to avoiding the memory latency each time a data block (from the

loop-blocking technique) has to be fetched from memory. With prefetch, every block resides in the cache before it is executed, as the prefetch thread has it fetched already from the time that the previous block was executed. Since for the calculation of the bandwidth the execution time of all the iterations is used, if the number of the iterations is large, then the execution time measured when prefetch is included will tend to the execution time measured as if prefetch was not included.

Even if the calculations are repeated 5 times, the performance gain of the prefetch will not be able to be demonstrated this way, as illustrated in the figure below. The bandwidths will have similar values when prefetch is included and when it is not.

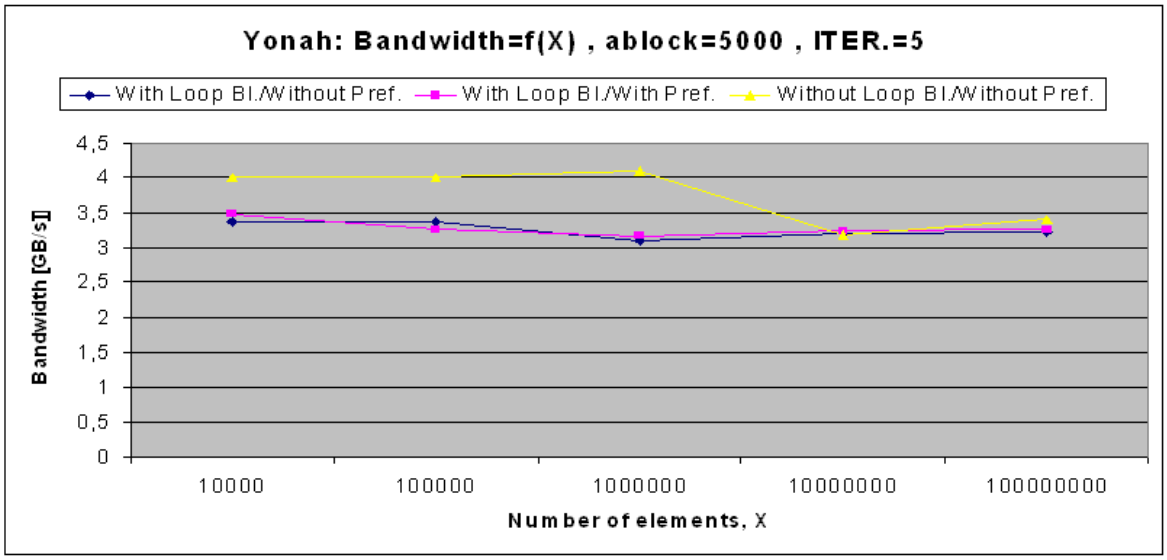

Figure 5.2. On Yonah: For many iterations, the gain of the prefetch cannot be demonstrated

On the other hand, the size of the block for the blocking does not draw any significant impact on the measurements, as long as it's size remains smaller than the size of the cache of course. Therefore the block size is going to be fixed on one value during all measurements, that is 5000 byte *8 doubles =40000 byte or 40 KB.

In the measurements that we conducted with the addition benchmark, we measured the bandwidth against the proportion X. X is the number of doubles in the array and from this proportion we can easily calculate the size of the array in byte; the latter is X*8 byte, while the bandwidth is calculated in GB/s. The size of the block is fixed to 40 KB, which is smaller than the size of the L2 cache (2MB for the Yonah and 4MB for the Woodcrest). The number of iterations is restricted to 2, as discussed above. The first graph illustrated below corresponds to the Yonah processor, while the second graph illustrated to the Woodcrest.

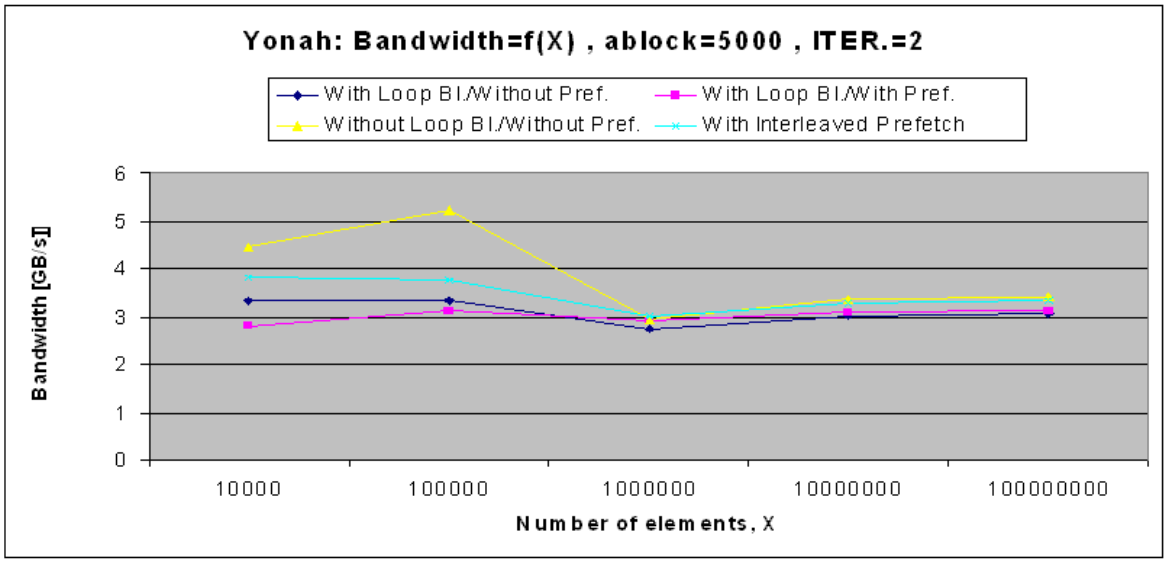

Figure 5.3. On Yonah: Bandwidth as a function of the array size

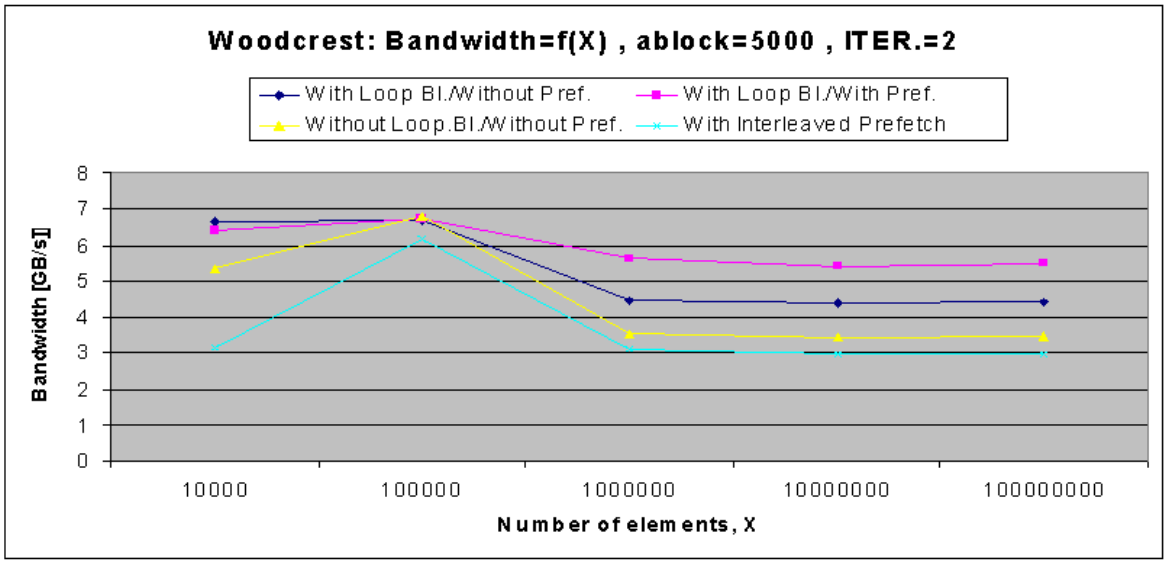

Figure 5.4. On Woodcrest: Bandwidth as a function of the array size

First of all, we are able to mark out the maximum bandwidth of the addition benchmark for each of the machines. As expected, in the first two measures the array can still fit into the cache, as its size is 80 KB and 800 KB respectively, numbers smaller than the 2 MB or 4MB which is the size of the cache in the Yonah and Woodcrest respec-

tively. Therefore, the corresponding bandwidth must be the maximum for both of the measures for the particular benchmark. On both machines this is true for the case that prefetch is not included. Then, for the three following measures, where the size of the array begins to exceed the size of the cache from 8 MB that is on the 3rd measure to 800 MB on the 5th, the bandwidth drops around 0,28 MB for the Yonah processor and 2,3 MB for the Woodcrest when prefetch is not included. This drop can be partially associated with the fetching of the blocks at the first time a block is to be calculated from the memory. It is clear that if prefetch is included, a greater drop in the bandwidth is avoided; the bandwidth drops by only 1,2 MB for the Woodcrest and by only 0,22 MB for the Yonah processor. The situation is different though, when the prefetch is interleaved into the code and not done by the prefetch thread, as in the previous case. In this case on the Yonah the bandwidth is improved even more than the improvement that is brought by the prefetch thread, therefore the drop is smaller than in the previous case. On the other hand, on Woodcrest interleaved prefetch results in a bandwidth worse than even the naive case and therefore, we notice a big drop in the measured values of the bandwidth.

In the graph below we demonstrate the gain in performance from prefetch by the prefetch thread and by prefetch interleaved in the code in percentages on both the Yonah and the Woodcrest processors.

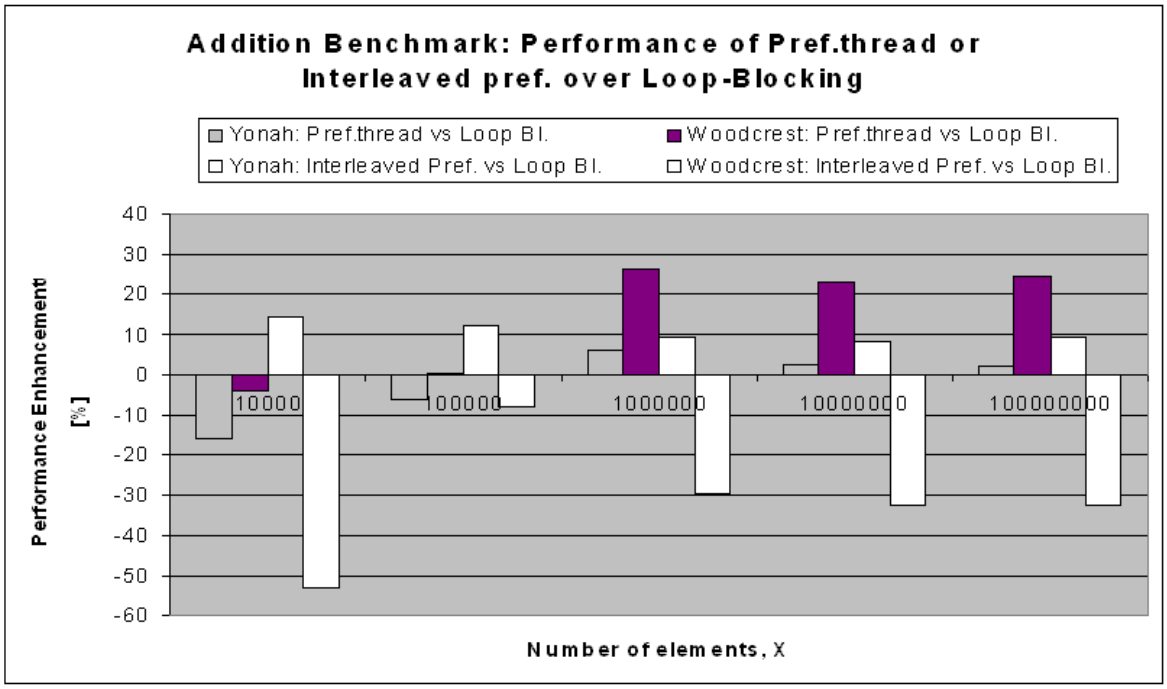

Figure 5.5. Addition Benchmark: Prefetch (from thread and interleaved) impact on the bandwidth

In the above graph, the maximum gain for the Woodcrest processor from thread prefetch can be as high as 26,2% (3rd measure). On the other hand, the maximum gain for the Yonah processor can be 5,9% (3rd measure as well) the most. This trend stands in the two other measurements as well: the gain from the Woodcrest processor drops to 23,1% in the worst case, while the gain from the Yonah to 2,2% in the last measure. However, when prefetch is interleaved into the code, benefit is only accomplished on the Yonah processor around 10%. On the Woodcrest, interleaved prefetch can decrease the bandwidth as much as 30%.

A remark that one can make is that there appear big differences among the results produced from the two processors. It looks like architectural differences between the two processors affect the measurements at a great extent.

5.2 Multiplication Benchmark

In the multiplication benchmark there are three variable proportions as well (parameters in the code); the size of the block because of the blocking in the row major order dimension which will be called iblock, the size of the block because of the blocking in the column major order dimension which will be called kblock and the three dimensions of the arrays. However, for this benchmark, we consider in all cases that the dimensions equal to each other, therefore it is enough that we define only one variable to describe the dimension proportion: X. In order to visualize the proportions, we introduce the following schema:

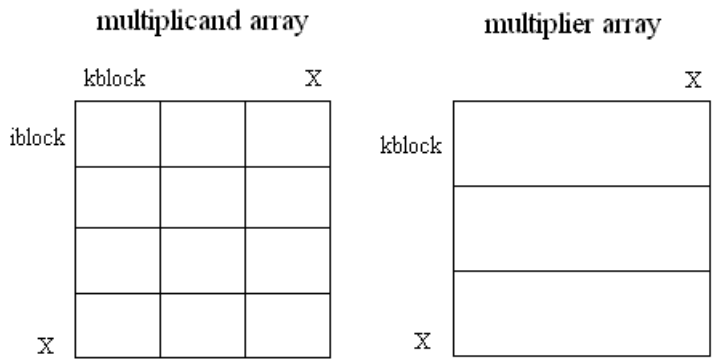

Figure 5.6. Matrix multiplication schema

As a first benchmark, we will measure the bandwidth in GB/s of the benchmark in relation to the size X. The kblock size will be fixed to 10 rows for the multiplier array and 10 columns for the multiplicand array, while the iblock size will be fixed to 2 rows

for the multiplicand array. We will take measurements for a starting X dimension of 200, which can be translated to array size of 200 * 200 * 8 byte = 320 KB, up to 1450 byte which means 1450 * 1450 * 8 byte = 16,82 MB. We will first present the results from the Yonah processor and then from the Woodcrest.

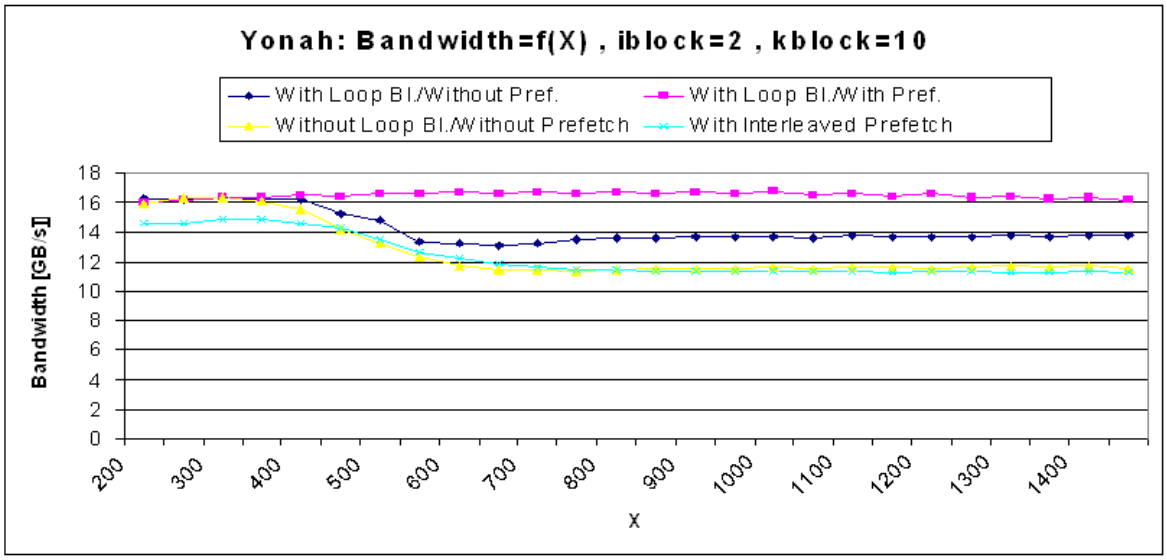

Figure 5.7. On Yonah: Bandwidth as a function of the dimension X

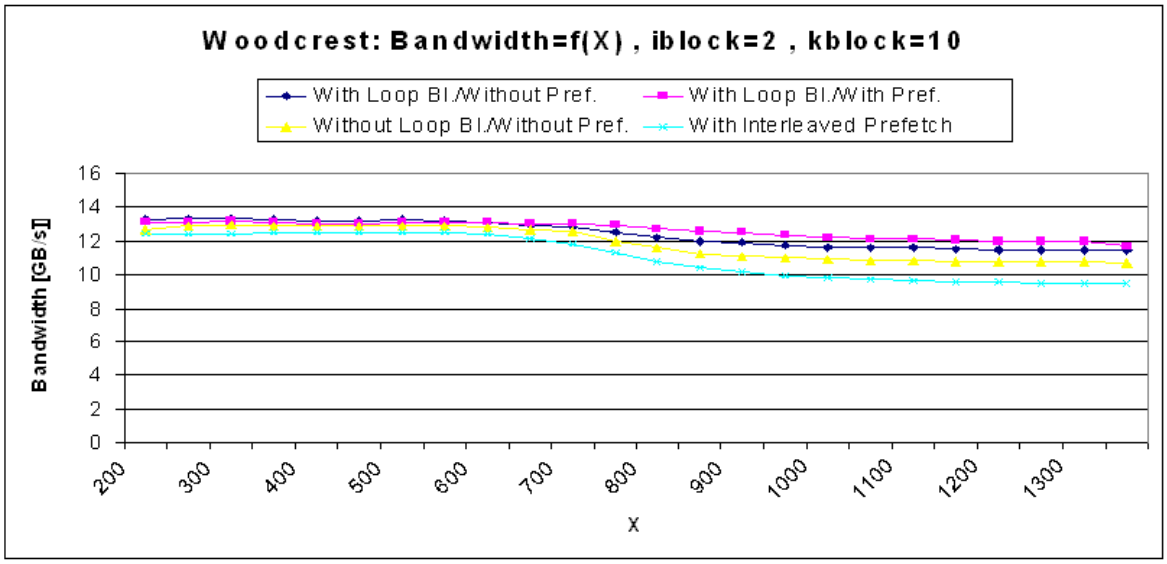

Figure 5.8. On Woodcrest: Bandwidth as a function of the dimension X

As long as the dimension X is smaller than 500 for the Yonah and 700 for the Woodcrest (respective array sizes will be 500*500*8 byte = 1,62 MB and 700*700*8 byte = 3,92 MB) the bandwidth will be maximum, as the array can still fit entirely in the cache. Afterwards, we notice that for the Yonah processor the bandwidth remains at maximum level if prefetch is included, while it drops around 3 GB when the prefetch is not employed. A similar behavior appears on the Woodcrest. The bandwidth after the critical size is passed drops around 1 GB when prefetch is included, but 2 GB when prefetch is not included. Then the gain from Yonah can rise for this benchmark as much as 26,82%, while the gain from Woodcrest does not exceed a percentage of 5,4. As for the interleaved prefetch, on Yonah this case results in a drop of the bandwidth to values as low as the values from the naive case. The percentage of drop can be as low as 20%. Even worse, on Woodcrest the bandwidth drops to values lower than the naive case values or around -16-17% comparing to the loop blocked version. A comparison of the percentages of the gains on the two machines is illustrated in the figure below:

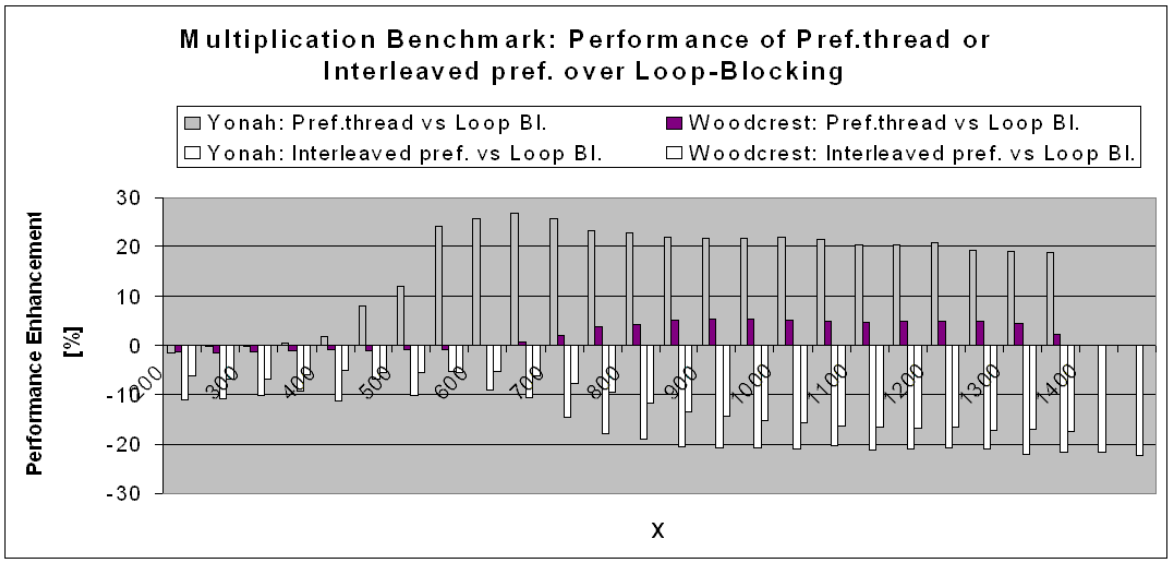

Figure 5.9. Multiplication Benchmark: Prefetch (threaded and interleaved) impact

Another interesting point arises from the relation between the size kblock and the gain percentage. This point can be illustrated in the following graphs. We first present the graph that depicts the bandwidth against the size X when the value of the size kblock is increased from 10 to 25 (rows for the multiplier matrix or columns for the multiplicand matrix).

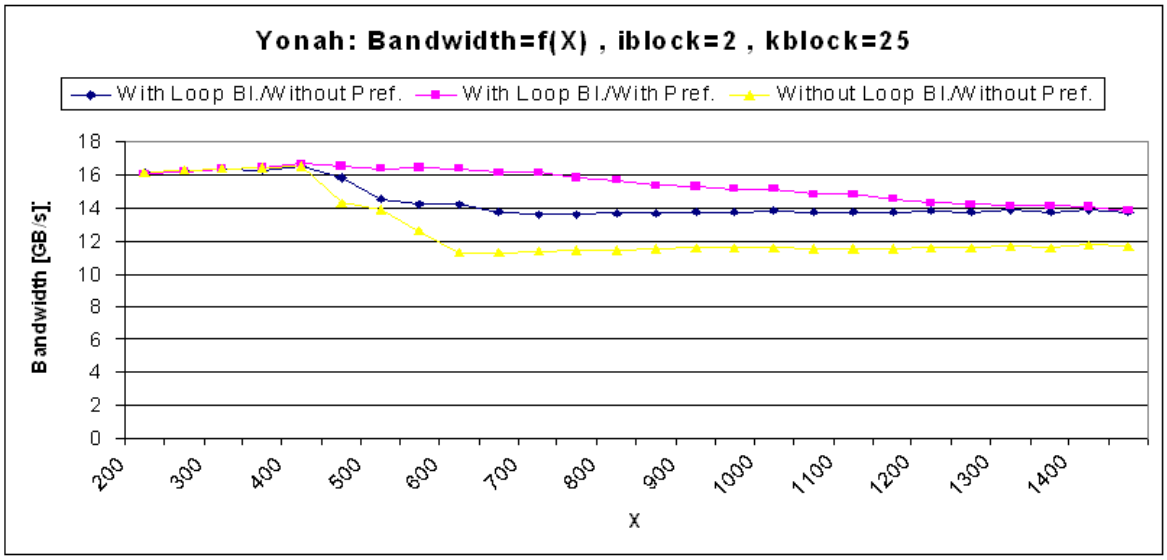

Figure 5.10. On Yonah: Bandwidth as a function of size X when kblock=25

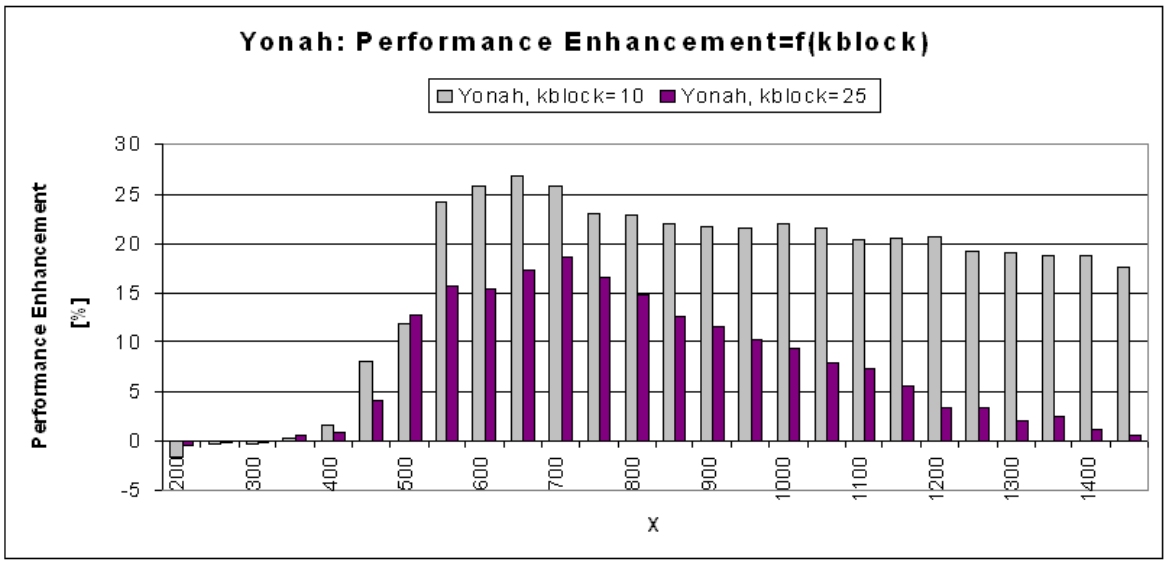

Figure 5.11. On Yonah: Effect of the size of kblock in the Performance Gain

In the last figure we notice that there is a significant drop in performance when the size kblock is switched from 10 to 25 or from 10*8=80 byte to 25*8=200 byte. Then, for large array sizes, the gain from Yonah drops to 18,6% in the best case, while as long as the dimension X decreases the gain tends almost to zero. A possible explanation for this

can be the following. The more the size of kblock is increased the more doubles have to be fetched from memory by the prefetch. This way it could be likely that not all doubles are fetched to the cache, when the calculation of a new data set has begun. Therefore, the gain of the actual prefetch is too little to be demonstrated in the above figure, in case that kblock equals to 25.

5.3 Jacobi Benchmark

The code for the jacobi benchmark has three parameters; the dimension of the column-major order M, the dimension of the row-major order N and the size iblock that is the size of the blocking on the row-major order dimension. There follows the visualization of this schema:

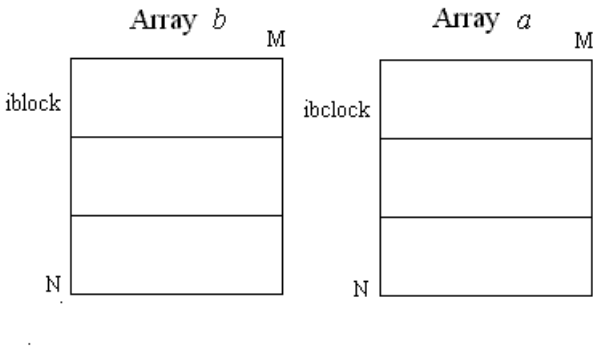

Figure 5.12. Jacobi Benchmark Schema

As in the previous benchmarks, we will measure the bandwidth in GB/s of the benchmark code on each machine against proportion X. X is dimension M (column-major order) , while dimension N (row-major order) is defined as 100 times greater than dimension N. We prefer this definition, because of the following assumption; the node relaxation is done row by row, while iblock is the number of rows. In our measurements, the important proportion is the iblock, because this is the proportion that actually has an effect on the measurements. Increasing the number of columns makes no difference. Since there is a limit to the extent that an array can be outgrowth (too large array dimensions result in memory allocating error), we prefer to increase the row-major order dimension than the column-major order dimension. This is the reason why we define dimension N as 100 times greater as dimension M.

We will take three measurements with this benchmark on each machine; we will measure the bandwidth of the naive code version, the bandwidth of the code with blocked loops and the bandwidth of the code with blocked loops including a prefetch thread, all against the size X. The results are illustrated in the figures below.

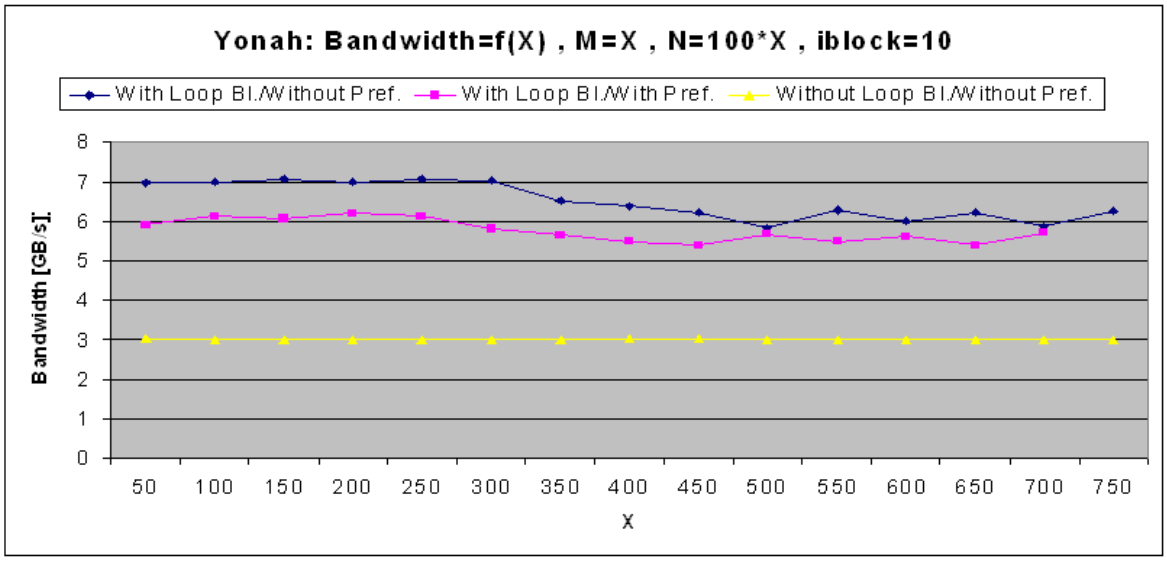

Figure 5.13. On Yonah: Bandwidth as a function of the size X

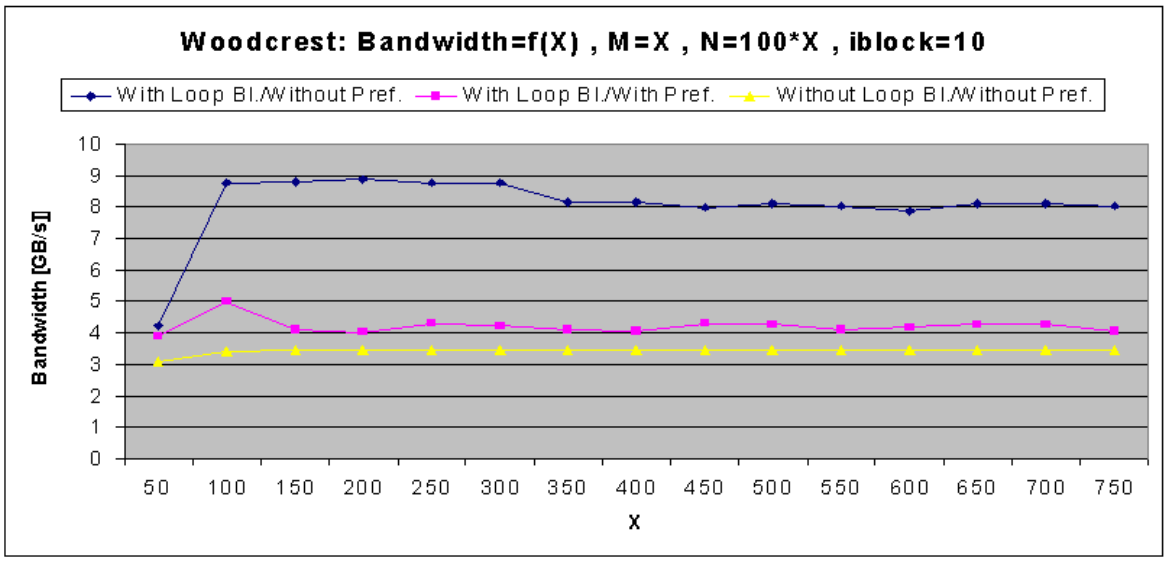

Figure 5.14. On Woodcrest: Bandwidth as a function of the size X

We notice that for both processors the bandwidth of the naive benchmark is really low; around 3,44 GB/s for Woodcrest and 3 GB/s for Yonah. Moreover, we notice that when the loops are blocked, the bandwidth increases a lot; the bandwidth on Yonah rises to 7 GB/s for array sizes smaller than 350*35000*8= 98 MB, while it drops to values around 6GB/s when the array size is greater than 98 MB. Moreover, it goes up to 8,8 GB/s for array sizes between 100*10000*8= 8MB and 300*30000*8= 72 MB on Woodcrest, while it drops to values around 8 GB/s for array sizes larger than 72 MB. We notice that for both machines and for the whole range of values, there is an increase of more than 100% of the bandwidth. In particular, the values of the bandwidth can increase from 94% to 136% comparing to the values of the naive implementation for Yonah and from 127% to 156% on the Woodcrest.

As for the case that the prefetch thread is included, we notice that the bandwidth is around 6 GB/s on Yonah for array sizes smaller than 98 MB, while for larger sizes it fluctuates around 5,5 GB/s. On the other hand, on Woodcrest, except for the case that the size of the array equals to 8 MB, the bandwidth fluctuates around 4 GB/s. In this case the gain from the implementation including prefetch fluctuates from 82% to 105% on Yonah and from 17% to 26% on Woodcrest. For both of the cores therefore, the bandwidth is greatly improved. The gains are illustrated in the figure below.

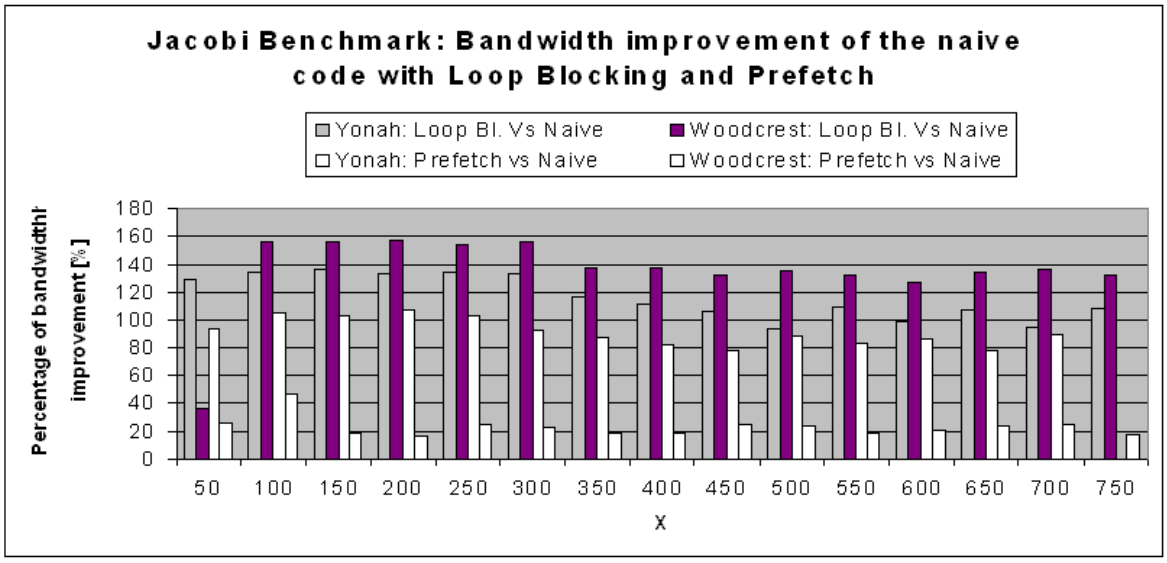

Figure 5.15. Bandwidth improvement of the naive code with loop blocking and prefetch

However, if we compare the values of the bandwidth when the loops are blocked to the values of the bandwidth when the loops are blocked and the prefetch is included, we notice a drop. In numbers, the bandwidth for the case when prefetch is included comparing to the loop-blocked version where prefetch is not included drops averagely 15% on Yonah, while it can drop as much as 54% on the Woodcrest.

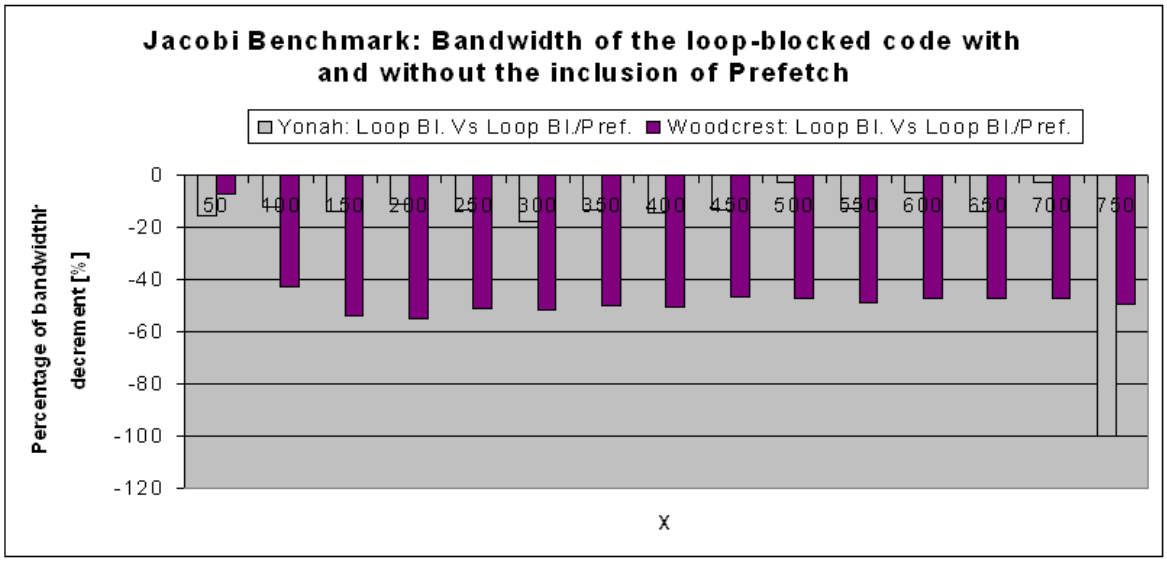

Figure 5.16. Bandwidth decrement because of the inclusion of prefetch on the loop-blocked code

Therefore, we can conclude that while the jacobi benchmark for array sizes such as not to fit in the cache can double or more the bandwidth when the loops are blocked, it drops the bandwidth when prefetch is included. The difference in the percentages of drop of the bandwidth between the measurements taken on the two cores leads us in the assumption of a possible connection of this phenomenon to an architectural difference between the cores. Apparently, this difference is responsible for the more than 50% drop of the bandwidth on the Woodcrest.

Chapter 6

Conclusion

Numerical codes ‘‘suffer” greatly from the increasing gap between CPU power and memory power. Indeed, since they typically perform on large data sets, actions must be taken so as to hide the memory latency caused by the above gap. Such actions include optimization techniques such as the data layout optimization, loop-optimizations and prefetching. In this thesis, we particularly looked for a potential benefit from realizing the prefetch optimization with the double-core processors Core Duo (Yonah) and Core 2 Duo (Woodcrest) utilized as prefetching engines. Our motivation was the shift that takes place in our times to include multiple cores on a single chip rather than one core, shift that at least for the near future seems like a one-way road.

In order to investigate a potential gain from the utilization of the two double-core machines as prefetch devices, we checked three benchmarks on these machines; two benchmarks that consist of basic operations encountered in numerical codes such as addition and (matrix) multiplication, while as third benchmark we implemented and checked the jacobi method, which can be utilized as a smoother in the multigrid method, one of the fastest iterative solvers nowadays. The measures were compared against naive implementations of the benchmarks, implementations where only the loop-blocking technique was applied, implementations where the loop-blocking technique with the inclusion of a prefetch thread was adopted or prefetch instructions were interleaved into the main code of the benchmark. The results varied according to the machine on which a benchmark was tested, the different assignement of values to the program parameters and of course the optimization or the combination of the optimization techniques used.

Then, for the addition and the Multiplication benchmarks, with prefetch on Yonah the bandwidth was improved as much as 27% (maximum) and between 20-25% on average, while the same benchmark run on Woodcrest brought an average improvement in the bandwidth of around 4- 5%. The inteleaved version of the prefetching caused a decrement in the bandwidth on both processors. On the other hand, the Addition benchmark run on the Yonah processor introduced an increment in the bandwidth of around 2-5%, while the respective increment when the benchmark was run on the Woodcrest was up to 26%. The interleaved version of the prefetch of this benchmark showed a significant improvement in the bandwidth when it was run on Yonah; around 10% increment. On the Woodcrest the bandwidth decreased as much as 30%. As for the Jacobi benchmark, a notable improvement by the prefetch in the bandwidth was accomplished (for this benchmark only thread prefetch was implemented) but only when compared against the naive implementation of the benchmark; the percentage of improvement fluctuated between 80-100% for the Yonah and around 20% for the Woodcrest. When the bandwidth was compared with the respective values for the case that alone the loop blocking technique was applied (without prefetching), there was a decrement of around 15% on Yonah and of around 50% on Woodcrest.

From the above results, there seems to be a strong connection between the effect of the prefetch, whether this is from a prefetch thread running on a second core or interleaved in the code, and the architecture of the double-core. Then this is the reason why the two cores exhibit so different effects from the prefetching; the benchmark that introduced an approximate 25% improvement on Yonah, introduced an approximate 5% improvement on Woodcrest, while the numbers were swapped for the other benchmark. The difference of the cache sizes between the cores or the difference of the clock rates might be aspects that influence the performance, although such thing was not any clear from the measurements that we performed. Moreover, the Jacobi benchmark introduced a decrement in the bandwidth on both cores, fact that leads us to consider that there might be a common reason to cause this decrement. One could ‘‘blame” the overhead of the prefetch instructions over particular codes like the code in the Jacobi benchmark, while such overhead was not observed in the other benchmarks (there was an increment in bandwidth in all cases). Last but not least, an optimal configuration of the parameters of the program could be vital for observing potential gain of the prefetch. For example, when the block size or the number of iterations were given particular values, no improvement from the prefetch was observed at all.

To conclude, there are many aspects to be considered in case someone would like to further exploit prefetching techniques on multi-core machines; the effect that the architecture and the properties of the processor might possibly introduce, the effect of the implementation infrastructure for the prefetch schema and the optimal configuration of the parameters of the code.

Since the introduction of multi-core processors is here to stay at least for the next few years, the results of this thesis demonstrated that it could be indeed worth for someone to draw his/her attention to the further investigation of prefetching techniques on multi-core processors, either from the hardware's point of view or from the software's. Moreover, the proven performance potential for numerical codes from prefetching techniques, taking into account the expansion of the latter into a vast spectrum of engineering applications, can comprise a good means of justification for his/her choice.

Appendix A

Processors used in the Experiments

	Core Duo (Yonah)	**Core 2 Duo (Woodcrest)**
Code name	**atbode84**	**atbode140**
Clock rate	2,16 GHz	2,6 GHz
L1 cache	32 KB	32 KB
L2 cache	2 MB	4 MB

Appendix B

References

[1] Markus Kowarschik. Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures, PhD thesis, Lehrstuhl fuer Informatik 10 (Systemsimulation), Institut fuer Informatik, Universitaet Erlangen-Nuernberg, Erlangen, Germany, July 2004

[2] J.L. Hennessy and D.A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publisher, Inc., San Francisco, California, USA, 3. edition, 2003.

[3] Ruud van der Pas, Memory Hierarchy in Cache-Based Systems, http://www.sun.com/blueprints

[4] Markus Kowarschik and Christian Weiss. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. Proceedings of the GI-Dagstuhl Forschungseminar: Algorithms for Memory Hierarchies, Lecture Notes in Computer Science (LNCS), Vol. 2625, Springer.

[5] Malik Silva. Cache Aware Data Laying for the Gauss-Seidel Smoother, ETNA Kent State University

[6] S.P.Vanderwiel and D.J.Lilja, Data Prefetch Mechanisms, in ACM Computing Surveys, Vol.32, No.2, June 2000.

[7] Malik Silva, Application Programmer Directed Data Prefetching , Master's Thesis, York University, Toronto, Canada, July 2001.

[8] Web-page of CSIRO (Commonwealth Scientific and Industrial Research Organisation), http://www.csiro.au/csiro/content/standard/pscq,,.html

[9] Web-page of the 13th Annual Conference of Computational Fluid Dynamics, http://www.cfd2005.org/

[10] Horst D.Simon, Parallel Computational Fluid Dynamics Implementations and Results, Article in the MIT Press, 1992, http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=8165

[11] Dr. G. Wellein, Dr. F. Brechtefeld, F. Deserno, G. Hager. Architektur und Performance - Charakteristik moderner HPC Bausteine, HPC Services, Regionales Rechenzentrum Erlangen (RRZE)

[12] Ulrich Ruede, Markus Kowarschik, Arndt Bode, Josef Weidendorfer. Performance Prediction, Analysis, and Optimization of Numerical Methods on Cache-Based Computer Architectures, Half-day Tutorial at ISCA 2004, Munich, Germany

[13] Josef Weidendorfer, Carsten Trinitis. Cache Optimization for Iterative Numerical Codes Aware of Hardware Prefetching, International Conference on Applied Parallel Computing (PARA'04), Copenhagen, Denmark, June 2004.

[14] Blaise Barney. POSIX Threads Programming, overview of threads programming and the POSIX threads API, www.llnl.gov/computing/tutorials/pthreads/ [15] Robert Love. CPU Affinity, Article in Linux Journal, 01-07-2003, http://www.linuxjournal.com/article/6799

[16] Alan Zeichick. Driving in the Fast Lane: What Multi-Core Computing Means for Programmers, Part I, Article on AMD Portal, August 16, 2005, http://www.devx.com/amd/Article/29117

[17] Nick Wyman. How Multigrid Solver Acceleration Works, Article on CFD Revier, November 30, 2001, http://www.cfdreview.com/article.pl?sid=01/11/28/2217256