Cache Miss Analysis of Walsh-Hadamard Transform Algorithms

A Thesis

Submitted to the Faculty

of

Drexel University

by

Mihai Alexandru Furis

in partial fulfillment of the

requirements for the degree

of

Master of Science in Computer Science

March 2003

Acknowledgements

I would like to express my sincere gratitude to my graduate advisor, Professor Jeremy Johnson Ph.D., for his continuing support during the thesis preparation. Without his advice, guidance, understanding and patience everything would have been a lot more complicated.

Finally, I am grateful to all the members of the Computer Science and Mathematics Departments: professors, TAs or friends, with who I collaborated over the years at Drexel University.

Table of Contents

List of Tables

List of Figures

Abstract

Cache Miss Analysis of Walsh-Hadamard Transform Algorithms

Mihai A. Furis

Jeremy Johnson Ph.D.

Processor speed has been increasing at a much greater rate than memory speed leading to the so called processor-memory gap. In order to compensate for this gap in performance, modern computers rely heavily on a hierarchical memory organization with a small amount of fast memory called cache. The true cost of memory access is hidden, provided data can be obtained from cache. Substantial performance improvement in the runtime of a program can be obtained by making intelligent algorithmic choices that better utilize cache.

Previous work has largely concentrated on improving memory performance through better cache design and compiler techniques for generating code with better locality. Generally these improvements have been measured by using collections of benchmark programs, simulations and statistical methods. In contrast in this work investigates how the choice of algorithm affects cache performance. This is done for a family of algorithms for computing the Walsh-Hadamard transform a simple yet important algorithm for signal and image processing. The WHT is a particularly good starting point due to the large number of alternative algorithms that can be generated and studied. Moreover the WHT algorithms have an interesting strided memory access pattern that can be analyzed analytically. A procedure is developed to count the number of cache misses for an arbitrary WHT algorithm and this procedure is used to investigate the number of cache misses for different algorithms.

Chapter 1: Introduction

During recent years there has been an unprecedented increase in the speed of the processors. Processor performance improved 35% per year until 1986 and 55% after 1986. Unfortunately memory performance has not kept pace with processor speed leading to the processor – memory gap [1]. Even if the processor is very fast, overall performance of the computer can still be poor due to a slow memory access time. The processor will stall waiting for the memory operations to complete before it can execute the next instruction. In order to obtain high performance, it is necessary to reduce memory access time. The concept of a memory hierarchy was introduced to solve this problem. Memory is organized into a hierarchical structure with a small amount of fast memory, and increasingly larger amounts of decreasingly slower memory. The computer first looks in the fast memory called cache, to satisfy a memory request. Only if the access fails does the slower memory need to be accessed. If most accesses are located in the cache then slower memory access time does not severely affect overall performance.

The memory hierarchy design is based on two principles: 1) the principle of locality (both temporal and spatial), and 2) the cost/performance of the memory.

The principal of temporal locality says that it is likely for recently accessed memory to be accessed again in the near future. Spatial locality says that memory

locations that are nearby are likely to be accessed within a short period of time [2].

The cost of memory technology influences the memory hierarchy design. The faster the memory, the higher the cost, and consequently only a small amount of fast memory is practical. The size of the fastest memory is also limited by its physical location. The memory close to the processor is faster than memory located far from the processor since fewer wires are needed and the communication time is reduced. Thus only a limited amount of cache memory near the processor is available.

Because the need for cache memory is dependent on processor speed it is only seen with very fast processors. In 1980 processors did not have cache, while in 1985 several levels of cache become prevalent. At the present time six levels of cache can be seen frequently inside high performance computers.

Previous work has largely concentrated on improving memory performance through better cache design and compiler techniques for generating code with better locality [7], [8]. Generally these improvements have been measured by using collections of benchmark programs, simulations and statistical methods [5].

However, compiler work has been limited to simple loop based programs. Substantial performance improvement in the runtime of a program can be obtained by making intelligent algorithmic choices that better utilize cache and exploit spatial and temporal locality. Algorithm designers typically use operation counts to measure performance. This worked well with simple computers, however today's computers with complicated memory designs require performance models that account for the memory hierarchy. The goal of this thesis is to better understand how algorithmic choices affect the cache behavior of a program. Using cache simulators and hardware counters we compare the

cache behavior for different memory access patterns and use this data to predict the cache behavior of several simple yet important algorithms.

Previous work along these lines concentrated on simple data structures like linked lists or binary trees. The results obtained can be used to improve the runtime of programs that make extensive use of these data structures. For example great improvements have been obtained by applying the results to databases, programs that make extensive use of B trees [9], [10]. Other work has focused on theoretical results that do not readily aid the algorithm implementer [11] and [12]. Recent work on the number of cache misses in algorithms like matrix multiplication is closer to the work in this thesis [13].

This work investigates how the choice of algorithm affects cache performance. This is done for a family of algorithms for computing the Walsh-Hadamard transform a simple yet important algorithm for signal and image processing. The WHT is a particularly good starting point due to the large number of alternative algorithms that can be generated and studied [3]. Moreover the WHT algorithms have an interesting strided memory access pattern that can be analyzed analytically. A formula is developed to count the number of data cache misses for an arbitrary WHT algorithm and this formula is used to investigate the number of cache misses for different algorithms.

Previous work investigated the distribution of runtimes for different WHT algorithms [14]. Figure 1.1 shows the distribution of runtimes for WHT algorithms of size 2^{16}. All the algorithms have exactly the same number of arithmetic operations (NlogN), though they have very different data access patterns. Despite having the same number of arithmetic operations there is a wide range in runtime. The fastest program has a runtime of approximately 0.02 seconds while the slowest program has the runtime almost 0.1 seconds.

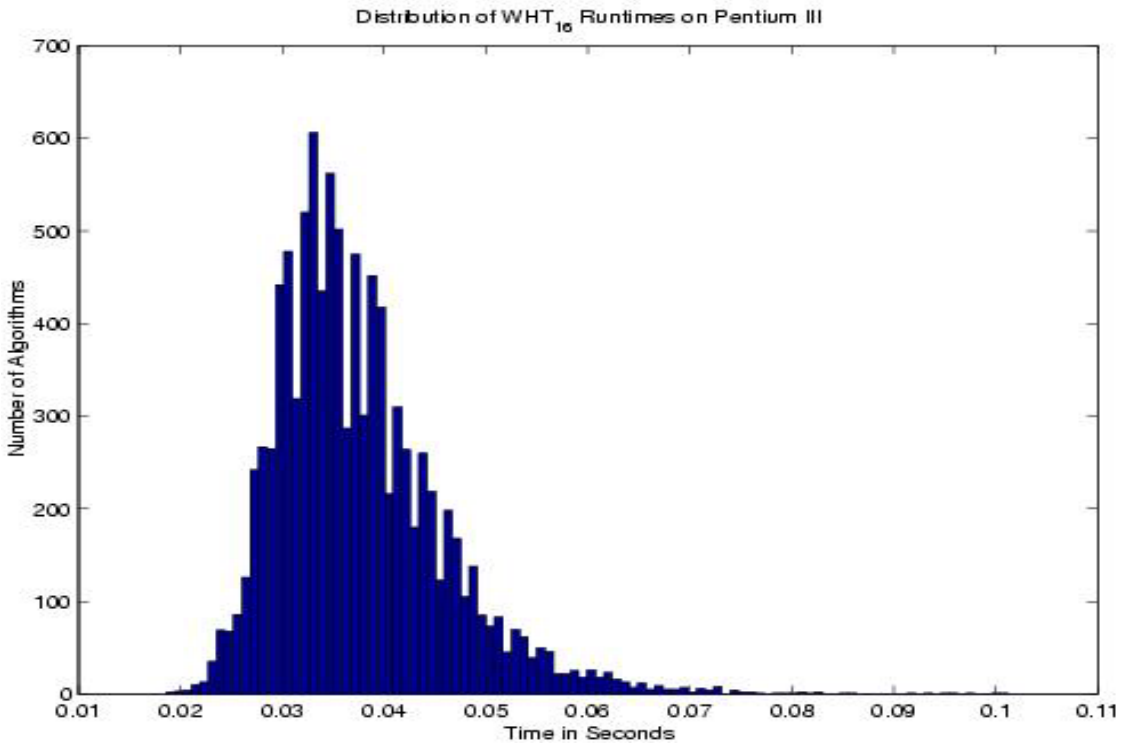


Figure 1.1 The distribution of runtimes for WHT algorithms

This suggests that runtime is heavily influenced by cache performance. Measurements in [14] shows that the number of cache misses vary dramatically in these algorithms.

In this thesis a formula is obtained for the number of cache misses for an arbitrarily WHT algorithm. This formula can be used to help the algorithm designer select a better algorithm to implement. Since the formula incorporates cache parameters it can be used to predict performance on machine designs that do not exist, and it can be used to relate algorithm design to cache parameters. This is not possible for search-based approaches that use runtime to find a fast implementation [3]. A cache simulator is used to verify the formula. The formula was derived by studying strided memory access and similar techniques could be

used for other programs with different strided memory access. Moreover, it gives a theoretical foundation for some of the empirical results in [3].

The thesis is organized as follows: Chapter 2 reviews cache memory organization and cache design. It includes a review of cache design parameters and performance equations. In Chapter 3 a program is used to stride through different levels of the cache and measure its performance. The program can also be used to analyze the cache configuration and determine its design parameters. Performance is measured in three ways: 1) using the UNIX time() command to measure the runtime, 2) using hardware performance counters to count the cache misses and 3) using a cache simulator. The cache simulator can simulate the execution of the same program on different cache designs. In Chapter 4 the cache performance of the WHT is analyzed. The chapter starts with a review of the Walsh-Hadamard transform and then presents results obtained from the cache simulator. Based on these results a formula for calculating the number of cache misses for the WHT algorithms is derived. Finally, in chapter 5 the formula is used to investigate the number of cache misses for different WHT algorithms.

Chapter 2: Review of cache memory structure and organization

This chapter reviews cache memory structure and organization. It starts with a short description of cache topology followed by a presentation of the key cache design parameters and ends with cache performance analysis. There are many good introductory books about cache memory (see [1] and [2]). The authoritative work in the field is "Cache and memory hierarchy design: a performance directed approach" written by Steven A. Przybylski, which contains a comprehensive presentation of both empirical and analytical cache models.

2.1 Hierarchical memory

There are three memory design goals:

- The size of the memory must be large and no constraints should be imposed on program or data size.
- The speed of the memory must be as fast as the fastest memory technology available.
- The cost of the memory must approach the cost of the cheapest memory technology available [2].

All of these goals cannot be achieved simultaneously since they are mutually exclusive. However, there is a practical compromise that approaches the ideal. The implementation is based on an hierarchy of memory levels. The memory closest to the processor is the fastest and the most expensive. This level is called the cache. The lowest level of the hierarchy is the slowest and uses the cheapest

type of memory available which usually consists of magnetic disks. Disk memory provides a huge amount of cheap storage space but has a large access time.

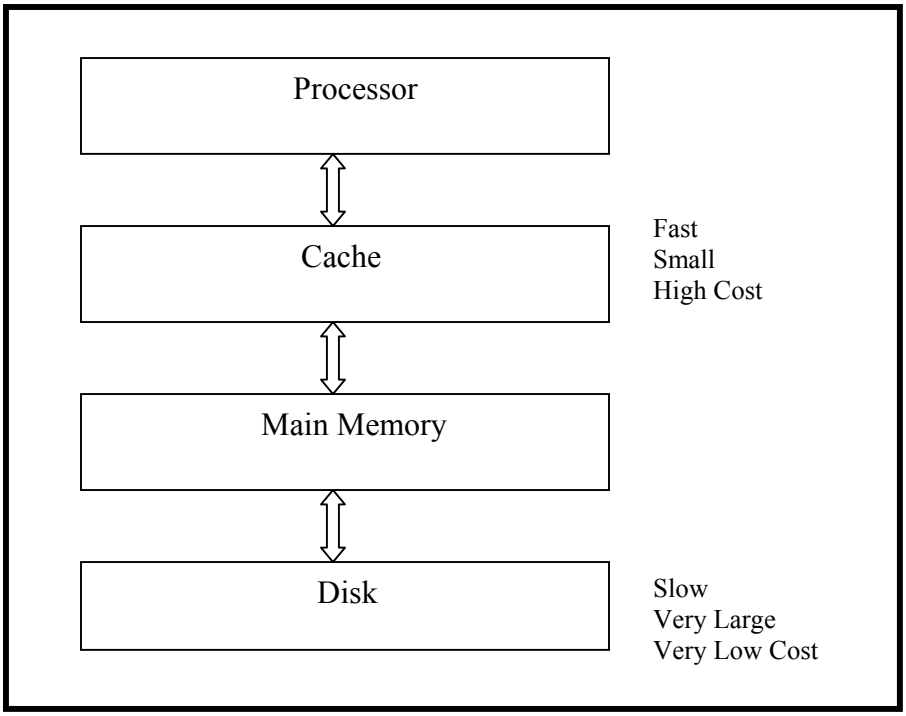


Figure 2.1 The memory hierarchy

The levels in the hierarchy are ordered by their speed, cost and size. When a memory access is issued the processor looks first into the level with fastest memory. In Figure 2.1 the cache level represents the fastest memory. If the required data is present inside the cache the data is transferred to the processor and the memory operation ends very quickly. However if the data is not present inside the cache another call is issued to the next level of the memory hierarchy, which in Figure 2.1 is the main memory. The process continues in a similar way until the level that contains the requested data is reached. So even if the lower level performance is poor, the system performance will approach the performance of the top level (in our example the cache memory), which is very

fast compared with the lower levels. This is due to the hierarchical structure of the memory and the locality principle stated in the introduction.

2.2 Cache topology

The cache topology deals with the number and interconnections of the caches between the processor and main memory. The cache memory is usually organized as an hierarchy of cache levels. The cache levels are labeled with L1 through Ln. Originally computers did not have any cache memory, now it is common to see two or more levels of cache. Such a design is shown in Figure 2.2

> Each level of cache is organized as a unified or split cache. A unified cache stores both instructions and data. A split cache has a separate data and instruction cache. The data cache stores only data while the instruction cache stores only instructions. The processor knows when a memory request refers to data or an instruction, and it sends the request to the corresponding data or instruction cache. The design in Figure 2.2 has a split level one cache and a unified level two cache

There are two reasons for having a split cache. The first is to eliminate a structural hazard that prevents pipelining with overlapped instructions. This allows the processor to read an instruction and write data in parallel. If the cache is unified then the processor cannot read and write at the same time and one of the instructions must be stalled. The other reason for splitting the cache is that the data and instruction caches can have different designs and can be optimized separately. A stream of instructions usually has very good spatial locality since instructions are read from consecutive memory addresses unless a branch is encountered.

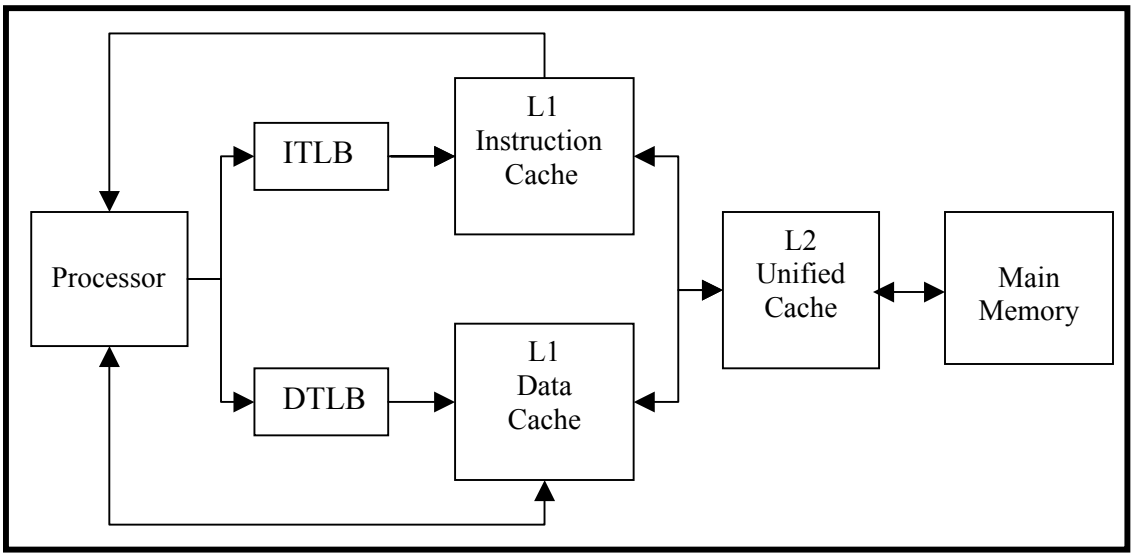


Figure 2.2 The memory structure and organization

The data-address stream has better temporal locality than spatial locality. After a memory location has been used by a program there is a good chance that it will be used again in the near future. The different access patterns for instructions and data imply different design optimizations that can be utilized separately for the instruction and data cache.

Figure 2.2 also shows two additional buffers called translation lookaside buffers (TLB): one for instructions (ITLB) and one for data (DTLB). These buffers are used to support a virtual memory system.

In a virtual memory system, user programs access virtual addresses which are then translated into physical addresses. Memory is grouped into pages and addresses are split into a page number and page offset. Virtual to physical translations are performed using a page table that maps virtual page numbers to physical page numbers. Virtual memory can be larger than physical memory, allowing very large programs and data structures. Pages not currently in memory reside on disk. If an access is made to a page not in memory, a page

fault occurs and one of the pages currently in memory must be replaced with the page just accessed. To reduce translation overhead a cache (TLB) of recent translations is used.

The hardware and/software must perform the following steps before accessing the real memory

1. Determine if the page holding the memory address is in the real memory.
2. If the page is present in real memory translate the page number, obtain the page starting address and concatenate the page address with the displacement, return the real memory address to processor and continue.
3. If the page is not present in memory call the page fault handler.

3.1 If there is a vacant page frame, fetch the page containing the requested memory address from the disk into real memory and update the TLB.

3.2 If there is no vacant place in real memory, evict a page to the disk and return to step 3.1 [2]

2.3 Cache read and write policy

When the processor makes a read or write access of the cache a policy must be provided and implemented in hardware. The policy is usually implemented in he cache controller which is responsible for managing all the transfers between caches and main memory.

READ HIT:

The referenced memory address is in the first level of cache. The content of the memory address is just transferred from L1 cache to the processor.

READ MISS:

1. If the referenced memory location is not present in the L1 cache then we have a cache miss and further actions have to be taken to bring the desired memory location into the L1 cache.

2. If there is no free block slot in the cache that can receive the desired block then a block must be evicted from the cache. Different replacement policies can be implemented: the evicted block can be the least recently used (LRU) or can be chosen in a random way from the cache. After an old block is evicted the cache controller reissues the read and finds a read miss with a vacant block step and the process continues with step 1.

WRITE HIT:

There are two policies in case of a write hit.

1. The write back policy

In this case the write is done only in the cache. When a write is made, the block's dirty bit is set, indicating that the block has been modified and does not contain the same data as the block in the main memory. Because the cache block and the corresponding block from the main memory are not the same, when there is a subsequent read miss, the block must be evicted from the cache and before eviction the block must be written back to memory so that it contains the current value.

2. The write through policy

Every time a write is done to a block contained in the cache, the data is also written to the corresponding location in main memory. A dirty bit is not required with this policy because the cache and the main memory are always coherent. When there is a subsequent read miss it is not necessary to evict the block, because there is another copy in the main memory. The new block can be simply overwritten.

2.4 Cache allocation and organization

The cache memory is organized in blocks or lines. When data is transferred between caches or between cache and main memory the entire block containing the desired address is transferred.

There are three ways in which the cache memory is organized depending on how a block is placed and found in the upper level of the memory hierarchy.

The first cache organization is called direct mapping. In a direct mapped cache a given address can only appear in one possible location in cache, and hence it is easy to map an address to its location in cache, and it is easy to determine if an address is in cache. The mapping is done using the formula:

(Block address) MOD (Number of blocks in the cache)

The disadvantage of a direct mapped cache is that an address might cause a miss while the cache is not full.

If a block can be placed anywhere in the cache the cache is said to be fully associative. A fully associative cache must be searched to determine if the desired address is in cache. However misses only occur when the cache is full.

The last cache organization called a set associative cache is a compromise between these two extremes. In this case the cache is split into sets, with each set containing the same number of blocks. A block that maps into a set can be placed in any block in the set. The mapping is done using the formula

(Block address) MOD (Number of sets in cache) [1]

2.5 Cache memory performance equations

First consider the performance model of a simple memory system containing only one level of cache and main memory. The cache is characterized by the cache access time, which represents the cache latency in case of a hit. Let's call this the HitTime or $t_{hit.}$ The probability of having a cache miss is P_{miss}. This

means that the probability of having a hit is 1-P_{miss} since P_{Hit} + P_{miss} = 1. The transfer time between the main memory and cache is T_{miss} or the miss penalty time. This means that the effective cache latency time is equal to

$$t_{avg} = (1 - P_{miss})t_{hit} + P_{miss}(t_{hit} + T_{miss}) =$$
$$= t_{hit} - P_{miss}t_{hit} + P_{miss}t_{hit} + P_{miss}T_{miss} = \quad (2.1)$$
$$= t_{hit} + P_{miss}T_{miss}$$

Now consider a more complicated cache performance model. In this case there is a multilevel cache. For simplicity a cache with only two levels L1 and L2 is considered. Using the performance model 2.1 for one level of cache

$$t_{avg1} = t_{hit1} + P_{miss1}T_{miss1}$$
$$t_{avg2} = t_{hit2} + P_{miss2}T_{miss2}$$

Since T_{miss1}=t_{avg2} we obtain

$$t_{avg1} = t_{hit1} + P_{miss1}T_{miss1} = t_{hit1} + P_{miss1}(t_{hit2} + P_{miss2}T_{miss2}) =$$
$$= t_{hit1} + P_{miss1}t_{hit2} + P_{miss1}P_{miss2}T_{miss2} \quad (2.2)$$

The performance equation for a cache with an arbitrary number of levels can be determined following the same logic as above.

Another formula for measuring cache performance is given in [1] and it has the advantage that it is independent of the hardware implementation.

The formula is: CPU time = (CPU execution clock cycles + Memory stall clock cycles) * Clock cycle time, where Memory stall clock cycles = Reads * Read miss rate * Read miss penalty + Writes * Write miss rate * Write miss penalty.

Combining reads and writes this simplifies to Memory stall clock cycles = Memory accesses * Miss rate * Miss Penalty.

Factoring out the instruction count (IC) and letting CPI be the average number of cycles per instruction, the CPUtime is equal to:

CPUtime = IC * (CPI execution +Memory stall clock cycles / Instruction) * Clock Cycle Time. (2.3)

2.6 Cache design parameters

There are two cache design parameters that dramatically influence the cache performance: the block size and the cache associativity. There are also many other implementation techniques both hardware and software that improve the cache performance but they will not be discussed here (see [1] and [4]).

The simplest way to reduce the miss rate is to increase the block size. However increasing the block size also increases the miss penalty (which is the time to load a block from main memory into cache) so there is a trade–off between the block size and miss penalty. We can increase the block size up to a level at which the miss rate is decreasing but we also have to be sure that this benefit will not be exceeded by the increased miss penalty.

The second cache design parameter that reduces cache misses is the associativity. There is an empirical result called the 2:1 rule of thumb which states that a direct mapped cache of size N has about the same miss rate as a 2 way set associative cache of size N/2. Unfortunately an increased associativity will have a bigger hit time. More time will be taken to retrieve a block inside of an associative cache than in a direct mapped cache. To retrieve a block in an associative cache, the block must be searched inside of an entire set since there is more than one place where the block can be stored.

Based on the cause that determines a cache miss we can classify the cache misses as compulsory, capacity and conflict misses. This classification is called the 3C model. Compulsory misses are issued when a first access is done to a block that is not in the memory, so the block must be brought into cache. Increasing block size can reduce compulsory misses due to prefetching the other elements in the block. If the cache cannot contain all the blocks needed during

execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. If the block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Increasing the associativity in general reduce the number of conflict misses and implicitly the runtime of the programs. However this is not true all the time. Minimizing the cache misses does not necessarily minimize the runtime. For example, there can be fewer cache misses with more memory accesses.

Chapter 3: Cache memory performance measurement tools

This chapter presents an overview of cache performance measuring tools. It also shows the impact of data access patterns on cache performance The first part of the chapter presents a program with a strided memory access pattern. The program introduces the strided memory access pattern and shows how the memory access pattern and cache configuration influences the runtime of the program.

The second part of the chapter presents the tools that have been used to measure cache performance. These tools are the performance counters and the Dinero cache simulator. The performance of the same program is measured using both performance counters and the cache simulator. The advantages and disadvantages of both tools are discussed.

3.1 A program for cache memory configuration and performance analysis

The following program uses segments of different sizes with different strides to invoke different levels of the cache. It runs the same code (the inner two loops) for different cache sizes and strides to be sure that all the cache levels are explored. The program computes the average memory access time for each stride and array size. Based on the timings it is possible to determine the cache configuration of the machine on which the program was executed. The code of was taken from the book Computer Architecture: A Quantitative Approach

written by David A. Patterson and John L. Hennessey. The program was initially written by Andrea Dusseau of University of California, Berkley [1].

```
#include <studio>
#include <time.h>
#define CACHE_MIN (1024) /* smallest cache */
#define CACHE_MAX (1024*1024)
#define SAMPLE 10
int x[CACHE_MAX];
double get_seconds() { /* routine to read time */
   clock_t t;
   t = clock();
   return (double) t/ (double) CLOCKS_PER_SEC;  }
int main(void)  {
      FILE *outptr;
           int register i, index, stride, limit, temp;
           int steps, tsteps, csize;
           double sec0, sec; /* timing variables. */
           outptr = fopen("output2.txt","w");
           for (csize = CACHE_MIN; csize <= CACHE_MAX; csize = csize * 2)
                    for (stride = 1; stride <= csize/2; stride = stride * 2) {
                             sec = 0; /* initialize timer */
                             limit = csize - stride + 1; /* cache size this loop */
                             steps = 0;
                             do { /* repeat until collect 1 second */
                               sec0 = get_seconds(); /* start timer */
                               for (i = SAMPLE * stride; i != 0; i = i - 1) /* larger sample */
                                  for (index = 0; index < limit; index = index + stride)
                                           x[index] = x[index] + 1; /* cache access */
                               steps = steps + 1; /* count while loop iterations */
                               sec = sec + (get_seconds() - sec0); /* end timer */
                             } while (sec < 1.0); /* until collect 1 second */
                             /* repeat empty loop to subtract loop overhead */
                             tsteps = 0; /* used to match no. while iterations */
                             do { /* repeat until same no. of iterations as above */
                               sec0 = get_seconds(); /* start timer */
```

```
                for (i = SAMPLE * stride; i != 0; i = i - 1) /* larger sample */
                    for (index = 0; index < limit; index = index + stride)
                            temp = temp + index; /* dummy code */
                    tsteps = tsteps + 1; /* count while loop iterations */
                    sec = sec - (get_seconds() - sec0); /* - overhead */
                } while (tsteps < steps); /* until = no. iterations */
                printf("Size:%7d Stride:%7d read+write:%140.f ns\n",  csize ,  stride ,
                        (double) sec*1e9/(steps*SAMPLE*stride*((limit-1)/stride+1)));
                fprintf(outptr," %7d %7d %7.f \n",  csize , stride ,
                        (double) sec*1e9/(steps*SAMPLE*stride*((limit-1)/stride+1)));
        }; /* end of both outer for loops */
fclose(outptr); }
```

The first loop of the program for (csize = CACHE_MIN; csize <= CACHE_MAX; csize = csize * 2) is iterating through different arrays with different sizes. The cache size range (which is represented by the csize variable in our program) starts from CACHE_MIN = 1024 = 2^{10}=1Kb and ends with CACHE_MAX = 1024 * 1024 = 2^{20}=1Mb. The size of the cache memory is doubled at every step (csize = csize *2) since cache sizes are typically powers of two csize will eventually be equal to the actual cache size.

The goal of the program is to determine the machine's cache memory configuration by analyzing the runtime of the program and to observe how the cache configuration is mirrored in the runtime of the program. Different strides are used to read from different blocks and levels of the cache memory. The measured runtime of the program will give valuable information about the memory configuration of the machine.

The program measures the time to access (read and write) elements of an array accessed at different strides. The size of the array (csize) and the stride (stride) are varied. For each size and stride the elements x[0], x[stride], …….,x[(csize/stride-1)stride] are repeatedly read and written. Figure 3.1 shows the access pattern for csize=16. To get a more accurate timing the elements are

accessed csize*SAMPLE times, when SAMPLE is a constant set to 10. Moreover the loop overhead is estimate by running and timing the loop without the array access, and the loop overhead is subtracted from the access time.

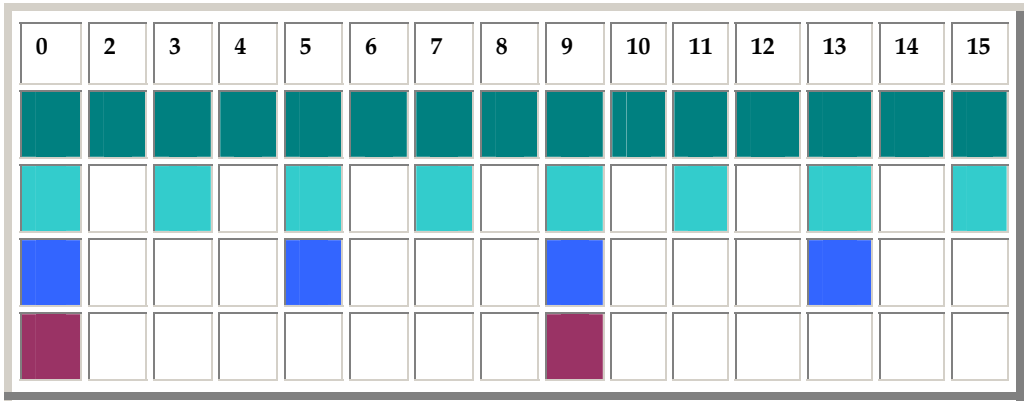


Figure 3.1 The stride pattern for the memory performance analysis program

Access time will depend on whether the read and write are in cache or not. Different access times are needed for different levels of cache. The strided access pattern influences the number cache misses depending on the block size and associativity.

To illustrate how the program can be used to empirically determine the cache configuration and to show the effects of strided memory access pattern and the cache configuration on runtime, the program was executed on a simple machine with the following configuration:

Identification: GenuineIntel, Pentium III

Hardware:

TLB

Instruction

4K-Byte Pages, 4-way set associative, 32 entries

4M-Byte Pages, fully associative, two entries

Data

4K-Byte Pages, 4-way set associative, 64 entries

4M-Byte Pages, 4-way set associative, eight entries

L1 Cache

Instruction

16K Bytes, 4-way set associative, 32 byte line size

Data

16K Bytes, 2-way or 4-way set associative, 32 byte line size

L2 Unified Cache

512K Bytes, 4-way set associative, 32 byte line size

The output of the program is plotted in the Figure 3.2. Each line in the chart corresponds to a different value of csize. From this chart we can determine the machine configuration and will compare it with the actual machine configuration. Every point from the chart corresponds to a certain csize and stride. The values of csize and stride are given in terms of integers, so when comparing to cache size it is necessary to multiply by sizeof(int)=4 (in this case)

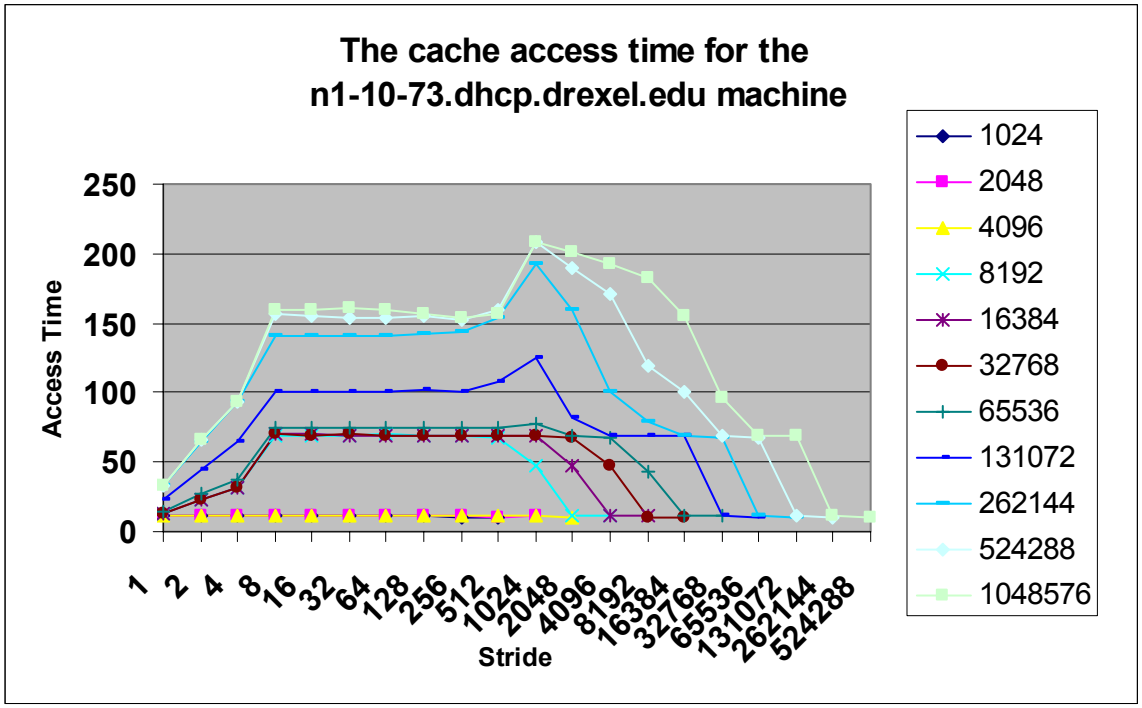


Figure 3.2 The cache access time for the n1-10-73 machine (Pentium III)

- **The number of cache levels**.

From the chart we can see that for csize equal to1024, 2048 and 4096 the memory accesses are approximately 15 ns. This means that the entire x array fits inside of the L1 cache, and no L2 cache requests are required. For csize of 8192, 16384, 32768, 65536, 131072 the memory access time is increased from 15 ns to 70 - 100 ns. This suggests that the memory access time involves the L2 cache since it increased drastically. In this case the x array does not fit in the L1 cache and access must resort to the L2 cache. For csize of 262144, 524288, and 1048576 the memory access time increased again from 70-100 ns to 150 ns. This is due to the fact that accesses to main memory are needed. All of these observations allow us to determine that the machine has two levels of cache: L1 and L2.

- **L1 cache size**

The size of L1 cache is determined from the first jump in access time. This occurs at csize = 8192. For all smaller values the L1 cache is large enough to hold the entire x array. Therefore the L1 cache size is L1_cache_size=4096 * sizeof (int) = 4096 * 4 = 16384 = 16 Kbytes.

- **L2 cache Size**

The size of the L2 cache is determined from the second jump in access time. This occurs at csize=262144. For all smaller sizes, the L2 cache is large enough to hold the entire x array. This means that the size of the L2 cache is L2_cache_size = 131072* sizeof (int) = 131072 *4= 524288 = 512 Kbytes

- **L1 and L2 cache block size**

The block size can also be determined from the data. For csize between 8192 and 131072 the number of L1 accesses increases until the stride size is equal to 8. When the Stride is 8 the number of L1 accesses is constant and the memory access time is between 70-100 ns. For strides 1, 2 and 4 the memory access time is

lower. The smaller access times are due to prefetching a block. With small strides several elements from the same block are accessed. When the stride is one all elements of the block are accessed. A reduced access time occurs until the stride is equal to the block size and only one element is stored inside a block.
For our data this suggest the block size is 8 * sizeof(int) = 8 *4 = 32 bytes.
The same logic applies for the L2 cache block which is also equal with 32 bytes since we have only misses for a stride bigger than 8.

- **The page size**

For a stride size of 1024 the memory access time increases when csize is greater than or equal to 131072. This suggests that there is a TLB entry miss. Before an access to the cache memory is done, the TLB is accessed. When the stride is greater than the page size, each access will be to a different page and a separate TLB entry is referenced. This means the page size is 1024 * sizeof(int) = 1024 *4 = 4 Kbytes.

- **The number of TLB entries**

TLB misses start when csize = 131072. This means that starting with this size the number of TLB accesses exceed the available TLB entries.
A cache size of 131072 integers is equal to 131072 * sizeof (int) = 131072 *4 = 4 Mbytes. So the largest number of elements that can be referenced by the TLB cannot exceed 4 Mbytes. Since the page size is 4 Kbytes. This means that the total number of pages that can be referenced by the TLB is 4 Mbytes/ 4 Kbytes = 2^{20} / 2^{12} = 2^{8} = 256 entries.

- **Associativity**

The associativity can be determined by looking at access time when the stride is larger than the cache size. When the stride is larger than the cache size each access will be mapped to a different set. As the stride varies a different number of elements get mapped to the same set, and this can be used to

determine the associativity. When stride = csize/2 two elements are accessed and an increased access time will occur for a direct mapped cache. When stride=csize/4, four elements are mapped to the same set and an increased accessed time will occur for direct mapped and 2-way set associative cache

When the stride size is csize/2 there are only two memory locations, which are accessed. They are 0 and csize/2. Since the stride size = csize/2 is greater then the L1 cache size then both of the two memory locations will map in the same memory set (0 in our case) if the cache organization is associative. If the stride size is csize/4 and is bigger then the L1 cache size then all the four values: 0, csize/4, csize/2, (3/4)*csize will also map in the same set (set 0).

For a stride of csize / 4 the access time is still equal with a L1 memory access time. For a stride size equal with csize/ 8 this is not true anymore and L2 accesses are needed since more then 4 blocks are mapped in the same set. This suggests the L1 cache is a 4-way set associative cache. The L2 associativity is determined in a similar fashion.

3.2 Performance monitors

Recent microprocessors have been designed to include special hardware support for measuring and monitoring performance. There are several programming APIs, for accessing these instructions, which measure various performance parameters. The performance monitors interface used for this thesis is called The Performance Counter Library, or PCL. More information about PCL can be found on its website at the address: http://www.fz-juelich.de/zam/PCL/.

The hardware support for performance measuring comes under the form of a set of performance counters with a defined set of countable events. The PCL interface allows us to initialize the set of counters with a specified set of events

and record these events. At the end of the program the interface can be used to retrieve the results.

The user has to specify the events that he wants to monitor during the execution of the program by initializing an array of events. The event list is passed to the PCLquery function, which is called to ask if the requested events (count floating point instructions, cycles or cache misses) can be measured on the system the program is running on. Then PCLstart is called to start counting the specified events. After the piece of code for which the events should be counted, PCLstop is called returning in the array result the requested numbers.

Table 2.1 shows a list of the memory hierarchy events that can be usually monitored on most machines.

Instruction and data caches are distinguished at each level. For unified caches (i.e. instruction and data are buffered in the same cache), it is often possible to distinguish instruction and data loads. Therefore on those caches, *PCL_LxICACHE_xxx* and *PCL_LxDCACHE_xxx* refer to events concerning instruction and data accesses, respectively. Due to the definition, the sum of cache reads and cache writes should be equal to cache read/writes and the sum of cache hits and cache misses should be equal to cache read/writes, too. Additionally, if two first level caches exist (instruction and data), the sum of instruction cache reads and data cache reads should be equal to cache reads.

Other types of events can monitor the number of instructions executed, the status of functional units or different rates and ratios (for example miss rates or floating point operations per second). Below is listed a new version of the Stride.c program which has been rewritten and makes calls to the PCL interface. The second nested loop and measurement of loop overhead have been deleted since timings are not made. The performance counters give the exact value of all L1 and L2 cache misses and hits.

Table 3.1 Events concerning memory hierarchy (x=1 or 2 for 1st or 2nd level cache)

Cache	
PCL_LxCACHE_READ	number of level-x cache reads
PCL_LxCACHE_WRITE	number of level-x cache writes
PCL_LxCACHE_READWRITE	number of level-x cache reads or writes
PCL_LxCACHE_HIT	number of level-x cache hits
PCL_LxCACHE_MISS	number of level-x cache misses
Data Cache	
PCL_LxDCACHE_READ	number of level-x data cache reads
PCL_LxDCACHE_WRITE	number of level-x data cache writes
PCL_LxDCACHE_READWRITE	number of level-x data cache reads or writes
PCL_LxDCACHE_HIT	number of level-x data cache hits
PCL_LxDCACHE_MISS	number of level-x data cache misses
Instruction Cache	
PCL_LxICACHE_READ	number of level-x instruction cache reads
PCL_LxDCACHE_WRITE	number of level-x instruction cache writes
PCL_LxICACHE_READWRITE	number of level-x instruction cache reads or writes
PCL_LxICACHE_HIT	number of level-x instruction cache hits
PCL_LxICACHE_MISS	number of level-x instruction cache misses
TLB	
PCL_TLB_HIT	number of hits in TLB
PCL_TLB_MISS	number of misses in TLB
Instruction TLB	
PCL_ITLB_HIT	number of hits in instruction TLB
PCL_ITLB_MISS	number of misses in instruction TLB
Data TLB	
PCL_DTLB_HIT	number of hits in data TLB
PCL_DTLB_MISS	number of misses in data TLB

```
#include <pcl.h>
#include <stdio.h>
#include <time.h>
#define CACHE_MIN (1024) /* smallest cache */
#define CACHE_MAX (1024*1024)
#define SAMPLE 10
int x[CACHE_MAX];
double get_seconds() { /* routine to read time */
        clock_t t;
    t = clock();
        return (double) t/ (double) CLOCKS_PER_SEC; }
```

```
int  main(int argc, char ** argv) {
FILE *outptr;
int register i, index, stride, limit, temp;
int steps, tsteps, csize;
double sec0, sec; /* timing variables. */
      /* PCL variables */
      int event_list[2];
      int ncounter,res;
      unsigned int mode;
      PCL_CNT_TYPE i_result_list1[2], i_result_list2[2];
      PCL_FP_CNT_TYPE fp_result_list1[2], fp_result_list2[2];
        /* Define what we want to measure. */
        ncounter = 2;
        event_list[0] = PCL_L1DCACHE_MISS;
        event_list[1] = PCL_L2CACHE_MISS;
        /* define count mode */
        mode = PCL_MODE_USER;
        outptr = fopen("output.txt","w");
        /* Check if this is possible on the machine. */
        if ( PCLquery(event_list, ncounter,  mode)  !=  PCL_SUCCESS)
           printf("requested events not possible");
           for (csize = CACHE_MIN; csize <= CACHE_MAX; csize = csize * 2)
                    for (stride = 1; stride <= csize/2; stride = stride * 2) {
                              sec = 0; /* initialize timer */
                              limit = csize - stride + 1; /* cache size this loop */
                              steps = 0;
                              res = PCLstart(event_list, ncounter, mode);
                              if (res != PCL_SUCCESS) printf("something went wrong");
                  /* do the work */
                                 for (i = SAMPLE * stride; i != 0; i = i - 1) /* larger sample */
                                    for (index = 0; index < limit; index = index + stride)
                                              x[index] = x[index] + 1; /* cache access */
                                 steps = steps + 1; /* count while loop iterations*/
       if ( PCLstop(i_result_list2,  fp_result_list2, ncounter) != PCL_SUCCESS)
           printf("problems with stopping counters");
```

```
        /* print out results (all integer results) */
        printf("csize = %d stride = %d:  %15.0f L1 cache misses and  %15.0f L2 cache misses\n",
             csize, stride, (double) i_result_list2[0],  (double) i_result_list2[1]);
        fprintf(outptr," %d  %d  %15.0f   %15.0f \n",
             csize, stride, (double) i_result_list2[0], (double) i_result_list2[1]);
                  }; /* endof both outer for loops */
fclose(outptr); }
```

The program counts L1 data cache misses and L2 unified cache misses. The program does not count the L1 instruction misses since we are not interested in counting them. Performance data similar to the timings in Figure 3.2 is given in Figure 3.3 and 3.4

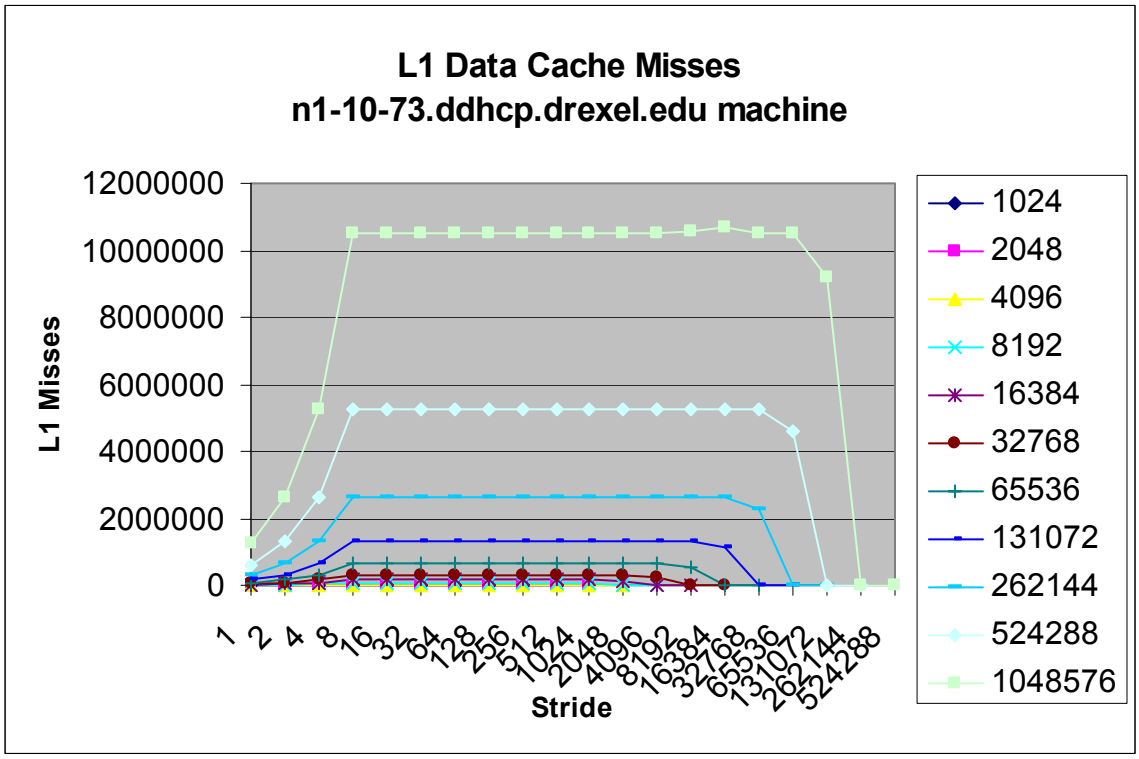


Figure 3.3 The number of L1 cache misses for the n1-10-73 machine (Pentium III) obtained using performance counters

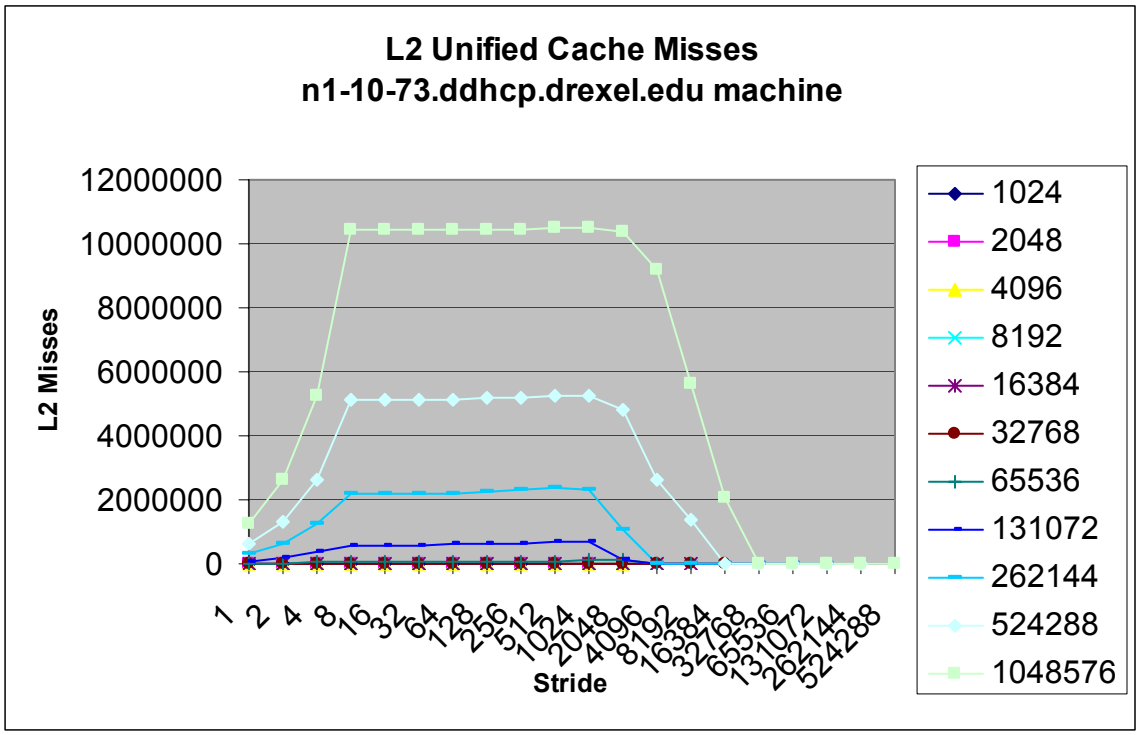


Figure 3.4 The number of L2 cache misses for the n1-10-73 machine (Pentium III) obtained using performance counters

From Figures 3.3 and 3.4 we can determine the L1 and L2 cache design parameters. However, we cannot determine the TLB related design parameters since we measured only the events related to the L1 and L2 cache.

- **The L1 block size**

For a stride size less than 8 the number of cache misses is very low suggesting that there are many array elements that fit into the same block. However for a stride bigger than 8 the number of cache misses becomes constant for a certain array size suggesting that the stride exceeded the block size. The L1 block size is 8 * sizeof (int) = 8 *4 = 32 bytes.

- **The L1 cache size**

For an array size less than or equal to 4096 the cache misses are very few. However if the array size is bigger than 4096 the number of L1 cache misses is

increasing drastically. This means that accesses to L2 cache are necessary and that the L1 cache size is 4096 * sizeof (int) = 4096 *4 =16384 = 16 Kbytes.

- **The L1 associativity**

We can determine the associativity of the L1 cache by looking at the graphics having a csize greater then the L1 size. We can see that for a stride size equal with csize/2 and csize/4 where csize/2 > L1 cache size and csize/4 > L1 cache size the number of cache misses is decreasing drastically. This means that the associativity of the L1 cache is 4 since we have 4 values that map in the same set (0, csize/4, csize/2 and (3/4)*csize).

From the second chart we can determine the following cache design parameters:

- **The L2 block size**

For a stride size bigger than 8 the number of L2 cache misses becomes constant. This implies that each array elements is stored in a different block and the stride exceeded the block size. The L2 block size is 8 * sizeof(int) = 8 *4 = 32 bytes.

- **The L2 cache size**

From the second chart we can see that the L2 cache misses increase drastically beginning with an array size of 262144 bytes. This means that all the cache accesses for an array size less than 262144 bytes are done inside the L2 cache. So the L2 cache size is 131072 * sizeof (int) = 131072 * 4 = 524288 = 512 Kbytes

- **The L2 associativity**

The L2 associativity can be determined by looking at the L2 graphics for a stride bigger then the L2 cache size. In this case we have more then 8 values (csize/8 case) mapping in the first set. Unfortunately, from our graphics we cannot determine the L2 associativity since all the stride sizes are less then the L2 size. The graphics must be drawn again for stride sizes bigger then the L2 cache size.

3.3 Cache simulators

Performance monitors are useful for measuring different events inside of the machine on which they run. If we want to run a program on a machine with a different cache configuration we have to construct the machine with the desired configuration. This is impractical and for this reason cache simulators have been developed. They allow us to simulate the execution of a program on different machines without building the actual machines.

The cache simulator used for this thesis is a trace-driven cache simulator developed by Mark Hill from University of Wisconsin at Madison. More information about the Dinero cache simulator can be obtained on its web page at http://www.cs.wisc.edu/~markhill/DineroIV/ .

The cache simulator has two major programs: a tracer and the actual cache simulator. The tracer reads the object code of a program and generates a trace of all the memory operations. All the memory operations are recorded by storing their type (read/write) and memory location. There are different trace formats used by cache simulators. Dinero is using a simple format called the dinero format.

The Dinero input format also known, as "din" is an ASCII file with one label and one memory address per line. The rest of the line is ignored so that it can be used for comments. The new extended Dinero format contains also a third field on each line which represents the number of bytes that are read or written starting with the specified address. By varying the last field we can simulate different machine word sizes.

The Dinero format label field specifies:

LABEL = 0 read data

1 write data

2 instruction fetch

3 escape record (treated as unknown access type)

4 escape record (cause cache flush).

Addresses are specified in hexadecimal with 0 <= ADDRESS <= FFFFFFFF and the hexadecimal addresses are NOT preceded by "0x".

A small dinero trace sample is listed below.

2 220

1 7fffcca4

0 54

1 7fffcca0

1 7fffcc9c

1 7fffcc98

1 7fffcc94

1 7fffcc90

2 56

Simulation results are determined by the input trace and the cache parameters. Unfortunately since each memory address requires at least 10 ASCII characters the size of trace files is usually very big. Rather than using a general memory tracer, the preceding stride program was modified to generate a dinero file with all accesses to the array x. This saves time and allows us to focus on the strided memory accesses. Since the program has only one statement that contains memory operations x[index] = x[index] + 1 we replaced this instruction with a print statement that records the memory access (each assignment instruction x[index] = x[index] + 1 is equivalent to a read and write memory operation). In fact only one access to x[index] was recorded.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define CACHE_MIN (1024) /* smallest cache */
#define CACHE_MAX (1024*1024)
#define SAMPLE 1
int x[CACHE_MAX];

int main(int argc, char *argv[])
{       FILE *outptr;
        int register i, index, stride, limit, temp;
        int mystride=1;
        int csize=CACHE_MIN;
        int address=0;
        outptr = fopen("trace.txt","w");
/* Get command line arguments */
if (argc == 3) {
          csize = atoi(argv[1]);
          stride = atoi(argv[2]);
        }
        else {
          stride = mystride;
        }
        printf("csize=%d\n",csize);
        printf("stride=%d\n",stride);
              limit = csize - stride + 1; /* cache size this loop */
                                        for (i = SAMPLE * stride; i != 0; i = i - 1) /* larger sample */
                                          for (index = 0; index < limit; index = index + stride)
                                                  { x[index] = x[index] + 1; /* cache access */
                                                   fprintf(outptr," 1 %x \n", index*4); }
fclose(outptr);
}
```

The program takes the csize and the stride as command line parameters and generates a trace for running the program with a certain cache size and using a certain stride size.

The size of the trace file can be calculated as follows:

1. The outer loop is executed for SAMPLE * stride times.

2. The inner loop is executed for (limit -1/ stride)+1 = [(csize –stride) / stride]+1 = [(csize /stride)-1]+1=csize/stride

3. The total number of memory accesses is SAMPLE * stride * csize/stride = SAMPLE * csize. SAMPLE was set to 1 which implies the number of lines in the trace is csize.

4. Each line requires 10 chars = 10 bytes to store the memory address.

5. The total number of bytes necessary to store the trace given csize and stride csize * 10 bytes.

A PERL script has been written to automate the generation of the trace files for all the csizes and strides. At the end of each trace generation the cache simulator is called and the actual simulation of the case is executed. The results of the simulations are stored in a file called “final”. The process is repeated until all the cases are executed. The L1 and L2 cache misses are recorded. The PERL script code is listed below:

```
#!/usr/bin/perl
#dinero configuration
#level L1 instruction cache size
$L1_i="-l1-isize 16k ";
#level L1 instruction cache block  size
$L1_ib="-l1-ibsize 32 ";
#level L1 instruction cache associativity
$L1_ia="-l1-iassoc 4 ";
#level L1 data cache size
$L1_d="-l1-dsize 16k ";
```

```
#level L1 data cache block size
$L1_db="-l1-dbsize 32 ";
#level L1 data cache associativity
$L1_da="-l1-dassoc 4 ";
#level L2 unified cache size
$L2_u="-l2-usize 512k ";
#level L2 unified cache block size
$L2_ub="-l2-ubsize 32 ";
#level L2 unified  cache associativity
$L2_ua="-l2-uassoc 4 ";
# format old dinero
$format="-informat d";
$options=$L1_i.$L1_ib.$L1_ia.$L1_d.$L1_db.$L1_da.$L2_u.$L2_ub.$L2_ua.$format;
open(FINAL,">final") || die " Sorry, out file doesn't exists\n";
for ($csize=1024; $csize <= 1024*1024; $csize=$csize*2) {
for ($stride=1; $stride <= $csize / 2; $stride = $stride * 2)
 { print FINAL $csize." ".$stride;
system "mytracer ".$csize." ".$stride;
print "dinero/d4-7/dineroIV ".$options." < trace.txt > out\n";
system "dinero/d4-7/dineroIV ".$options." < trace.txt  > out";
open(OUT,"<out") || die " Sorry, out file doesn't exists\n";
while (<OUT>) {
chomp;
if ($_ =~ /^ Demand Misses/)  { print $_."\n";
                                @words = split;
                                print FINAL  " ".$words[2]; }
 }
print FINAL "\n";
close(OUT); } # for loop
} # for loop
close(FINAL);
```

The final results obtained using the Dinero simulator are given in Figures 3.5 and 3.6. The L1 data cache misses and L2 unified cache have been plotted for each csize and stride.

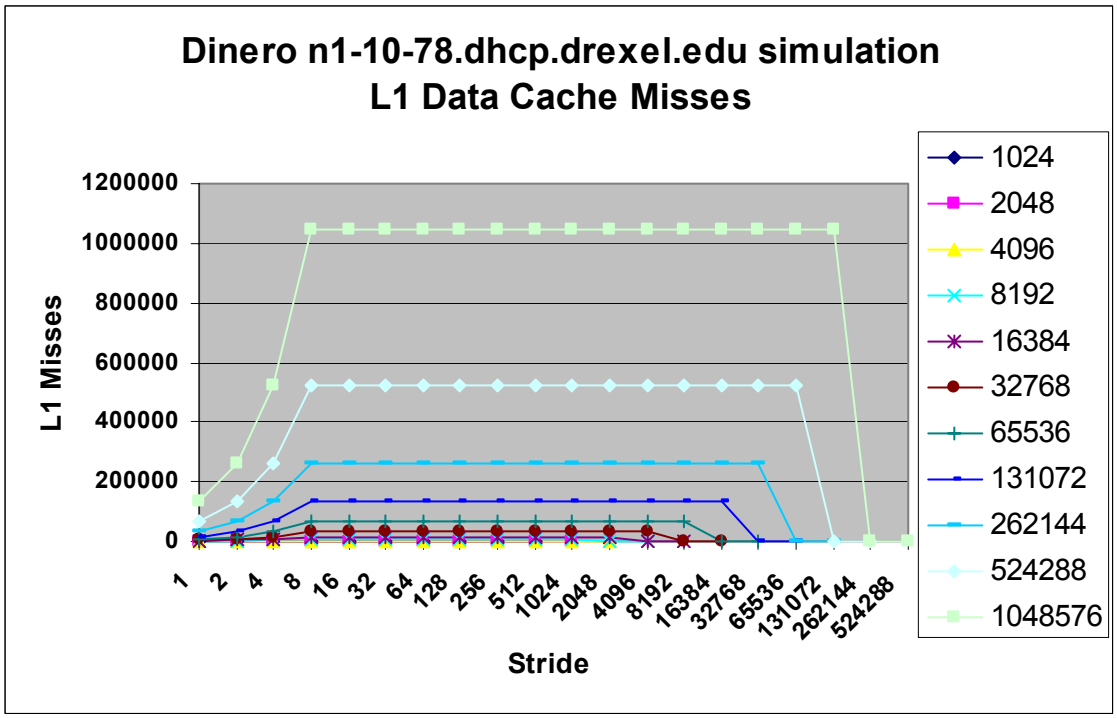


Figure 3.5 The number of L1 cache misses obtained using the cache simulator

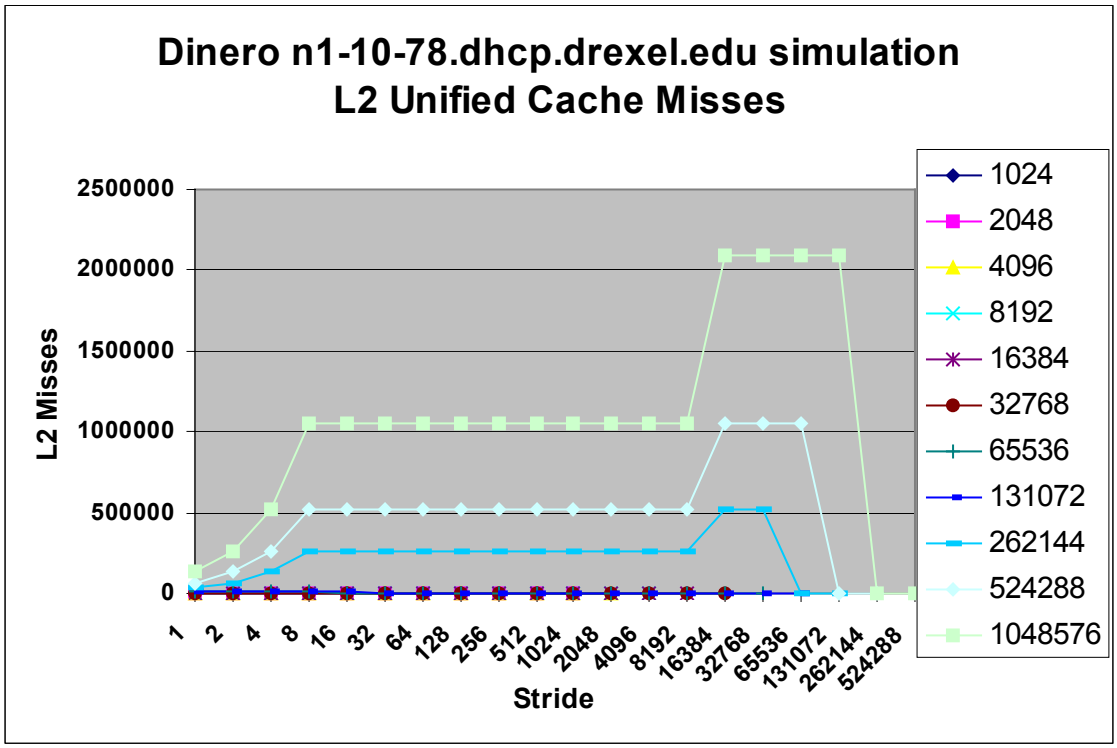


Figure 3.6 The number of L2 cache obtained using the cache simulator

Chapter 4: The Walsh – Hadamard transform memory access analysis

This chapter starts with a presentation of a family of Walsh-Hadamard transform algorithms and then derives a recurrence relation for calculating the number of cache misses of an arbitrary algorithm in this family. Using the general recurrence an exact formula for the number of cache misses is derived for WHT algorithms with radix one.

4.1 The Walsh – Hadamard transform

The Walsh–Hadamard Transform of a signal x, of size $N = 2^n$ is the matrix vector product $WHT_N * x$ where:

$$WHT_N = \bigotimes_{i=1}^{n} DFT_2 = \underbrace{DFT_2 \otimes ..... \otimes DFT_2}_{n} \qquad (4.1)$$

The matrix

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad (4.2)$$

is the 2-point DFT matrix and $\otimes$ denotes the tensor or Kronecker product. The tensor product of two matrices is obtained by replacing each entry of the first matrix by that element multiplied by the second matrix. Thus, for example

$$WHT_4 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \qquad (4.3)$$

If the operand matrices have the sizes m x n and p x q then the result matrix of the tensor operation will have size mp x nq.

Algorithms for computing the WHT can be derived using the following properties of the tensor product:

1. The tensor product operation is associative: $(A \otimes B) \otimes C = A \otimes (B \otimes C)$

2. Provided the dimensions are compatible, the product of tensor products is the tensor product of the products:

$$(A \otimes B)(C \otimes D) = AC \otimes BD \qquad (4.4)$$

3. $I_m \otimes I_n = I_{mxn}$, where I_n is the n x n identity matrix

4. $I_1 \otimes A = A \otimes I_1 = A$

In addition to the above properties it is easy to see that the Kronecker product is not commutative, i.e. $A \otimes B \neq B \otimes A$.

Using these properties

$$\begin{aligned} &WHT_{2^n} = WHT_2 \otimes WHT_{2^{n-1}} = WHT_2 I_2 \otimes I_{2^{n-1}} WHT_{2^{n-1}=} = \\ &= (WHT_2 \otimes I_{2^{n-1}})(I_2 \otimes WHT_{2^{n-1}}) \end{aligned} \qquad (4.5)$$

This factorization can be used for a recursive algorithm to compute the WHT.

An iterative algorithm that calculates the WHT can be obtain from the factorization:

$$WHT_{2^n} = \prod_{i=1}^{n} (I_{2^{i-1}} \otimes WHT_2 \otimes I_{2^{n-i}}), \qquad (4.6)$$

which follows by induction:

For n=2

$$\begin{aligned} &WHT_4 = (WHT_2 \otimes WHT_2) = WHT_2 I_2 \otimes I_2 WHT_2 = \\ &= (WHT_2 \otimes I_2)(I_2 \otimes WHT_2) = (I_1 \otimes WHT_2 \otimes I_2)(I_2 \otimes WHT_2 \otimes I_1) \end{aligned}$$

Assume the formula is true for n-1, i.e.

$$WHT_{2^{n-1}} = \prod_{i=1}^{n-1} (I_{2^{i-1}} \otimes WHT_2 \otimes I_{2^{n-i-1}})$$

To prove $WHT_{2^n} = \prod_{i=1}^{n} (I_{2^{i-1}} \otimes WHT_2 \otimes I_{2^{n-i}})$ observe

$WHT_{2^n} = (WHT_2 \otimes I_{2^{n-1}})(I_2 \otimes WHT_{2^{n-1}})$ which by induction is

$$(WHT_2 \otimes I_{2^{n-1}})(I_2 \otimes \prod_{i=1}^{n-1} (I_{2^{i-1}} \otimes WHT_2 \otimes I_{2^{n-i-1}})) = (WHT_2 \otimes I_{2^{n-1}})(\prod_{i=1}^{n-1} (I_{2^i} \otimes WHT_2 \otimes I_{2^{n-i-1}})) =$$

$$= (WHT_2 \otimes I_{2^{n-1}})(\prod_{i=2}^{n} (I_{2^{i-1}} \otimes WHT_2 \otimes I_{2^{n-i}}))$$

A general formula that includes both the iterative and recursive cases can be expressed as

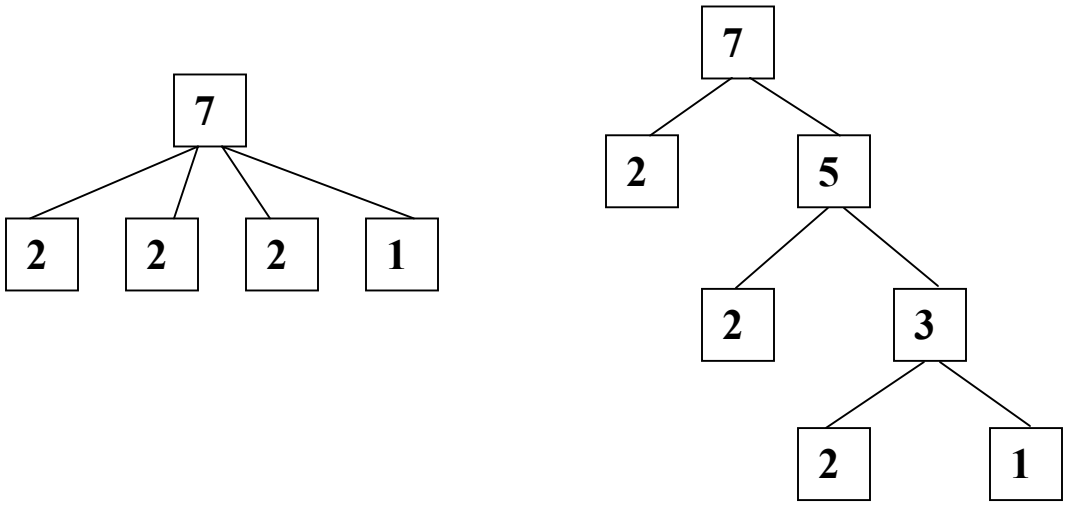
$$WHT_{2^n} = \prod_{i=1}^{t} (I_{2^{n_1+.....+n_{i-1}}} \otimes WHT_{2^{n_i}} \otimes I_{2^{n_{i+1}+.....+n_t}}) \quad \text{where} \quad n = n_1 + ... + n_t \qquad (4.7)$$

Depending on how the equation is applied we can obtain different algorithms for calculating the Walsh Hadamard transform, with varying amounts of iteration and recursion. Each factorization resulting from the above equation can be represented by a tree. The application of the equation 4.7 corresponds to an expansion of one tree node into children nodes whose sum equals the size of the parent node. A tree constructed in such a way is called a partition tree [3]. If all the leaf nodes have size equal to r except possibly one, which has size less than r, the WHT algorithm is said to have radix r. The two trees in Figure 4.1 have radix 2. The first corresponds to an iterative algorithm, and the second to a recursive algorithm.

4.2 The WHT software package

This section presents the software package used for the runtime measurements. The WHT package comes from [3] and can compute the WHT using any of the algorithms derived from equation 4.7.

A grammar is given for describing WHT algorithms and pseudocode is given for computing WHT. This section ends by describing the changes that have been made to the code for generating a memory trace. The trace records all the accesses to the input vector x. The algorithm is in-place so that x is also used for the output.



a) b)

Figure 4.1 Partition trees for iterative and recursive algorithms for WHT_{2^7}.

a) Tree for the iterative WHT algorithm of radix 2
b) Tree for the recursive WHT algorithm of radix 2

4.2.1 Algorithms for calculating the WHT

WHT algorithms are represented syntactically using a grammar for describing the recursion.

```
W (n) : : =small[n]   |
        split [W (n1), ……,W (nt)]   #  n=n1+…+nt
```

The string split [W (n1),….., W (nt)] corresponds to an application of the equation 4.7. For example the strings split [small [1], split [small [1], small [2]]] and split[split[small[2],small[1]] , small[1]] corresponds to the examples in Figure 4.1. Each leaf node of the tree is represented by the grammar as small[size]. If the node is not a leaf node then the node is called a split node and is represented as split[list of children].

Let $N = N_1....N_t$, *where* $N_i = 2^{n_i}$ and let $x_{b,s}^{M}$ denote the vector (x (b), x (b + s), ….., x(b+(M-1)s)). Evaluation of $x = WHT_N x$ using equation 4.7 is performed using the following algorithm.

$R = N;\ S = 1;$
$for\ i = 1,.....t$
  $R = \frac{R}{N_i}$
  $for\ j = 0,....., R-1$
    $for\ k = 0,....., S-1$
      $x_{jN_iS+k,S}^{N_i} = WHT_N * x_{jN_iS+k,S}^{N_i};$
  $S = S * N_i;$

This algorithm is applied recursively to compute WHT_{2^n} and accepts a stride parameter so that x can be accessed by the required stride. Several code generators are provided for producing code for small[n]. The code for small[1] , small[2] and the code corresponding to the above pseudocode is listed in Appendix A. A parser is provided for reading WHT expressions and translating them into a data structure called WHT tree. It is also worth notify that the algorithm is in place because the input and the output of the algorithm is represented by the same vector x. The matrix WHT is never stored inside memory due to its large sizes. It's elements are created by recursion when they

are needed. Only the vector x is stored inside the memory and obviously being the only source of data cache misses (ignoring temporary variables).

4.2.2 Memory trace generation

Another version of the package was created to produce memory traces to be analyzed by the dinero cache simulator.

The program described in the previous section has been modified to record all the vector accesses. As the algorithm recourses, the stride and base of the input are updated to indicate the subvector used in the recursion. See the pseudocode in the previous section or the code in Appendix A. The vector x is only accessed by the code for small that is used in the base case of the recursion. Therefore only the small code had to be modified to produce the memory trace. Print statements were added in the code for small to record all read and write accesses to the vector x (see Appendix A for the modified code for small[1] and small[2]).

Each memory location is read every time the program walks through the chunk. The implementation of small makes use of temporary variables to stores the chunk elements. For this reason the chunk elements are read only twice. First the values x[i] + x[S] and x[i]-x[S] are calculated and stored inside temporary variables. All the later use of this values will be done using the temporary variables instead of the actual vector elements. After the elements from a chunk has been read twice there are some calculation that are done using this values and at the end they written back in the x vector.

4.3 Formula for calculating the cache misses generated by the Walsh Hadamard transform

Let W_n be an algorithm to compute $x = WHT_{2^n} * x$. The objective of this section is to derive a formula to count the number of cache misses in the execution of W_n with a cache of size $C = 2^c$ with block size $B=2^b$ and associativity $A=2^a$. To simplify the analysis we only consider read and write accesses to the vector x.

The vector x is only accessed in the code implementing small, which corresponds to the leaf nodes in the partition tree associated with W_n. The first step in the analysis is to relate the memory access pattern to the structure of a partition tree. Let W_m be a leaf node in a partition tree of size n. The number of times small[m] is called is equal to 2^{n-m} . The code for small[m] reads the vector x for $2 * 2^m$ times and writes to x for 2^m times (see the code in Appendix A). Therefore the number of memory accesses in the execution of small[m] in the tree n is $3*2^m*2^{n-m}=3*2^n$.

Let l be the number of leaf nodes in the partition tree corresponding to W_n, then the number of read/write accesses in W_n are equal to $3 * l * 2^n \leq 3 * n * 2^n$. Thus any tree with the same number of leaf nodes has the same number of memory accesses.

A recursive call to a WHT algorithm accesses a subset of elements of the vector x called a chunk. The elements can be specified by a base address and stride as indicated in the pseudo-code in section 4.3. Let n_i be a child of the node n in the split corresponding to the factorization

$$WHT_{2^n} = \prod_{i=1}^{t} (I_{2^{n_1+.....+n_{i-1}}} \otimes WHT_{2^{n_i}} \otimes I_{2^{n_{i+1}+.....+n_t}}) \quad \text{where} \quad n = n_1 + ... + n_t .$$

The chunk of elements accessed by $WHT_{2^{n_i}}$ is $X^{2^{n_i}}{}_{b,S}$, where $S = 2^{n_{i+1}+....+n_t}$ and $b = j * 2^{n_i+.....n_t} + k$ *for* $0 \leq j \leq 2^{n_1+.....+n_{i-1}}$ *and* $0 \leq k < 2^{n_{i+1}+.....+n_t}$.

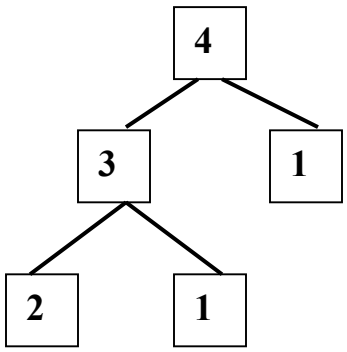
The elements of the chunk $X^{M}{}_{b,S}$ accessed by a small[m] node with M=2^m is

x[b], x[b+S], x[b], x[b+S], x[b+2*S], x[b+3*S], x[b+2*S], x[b+3*S], ….x[b+(M-2)*S], ,x[b+(M-1)*S], x[b+(M-2)*S], x[b+(M-1)*S],

x[b], x[b+S], …,x[b+(M-1)*S].

The first 2*M accesses are reads and the last M accesses are writes (see Appendix A).

Consider the tree:



$$WHT_{16} = (WHT_8 \otimes I_2)(I_8 \otimes WHT_2) =$$
$$((WHT_4 \otimes I_2)(I_4 \otimes WHT_2) \otimes I_2)(I_8 \otimes WHT_2)$$

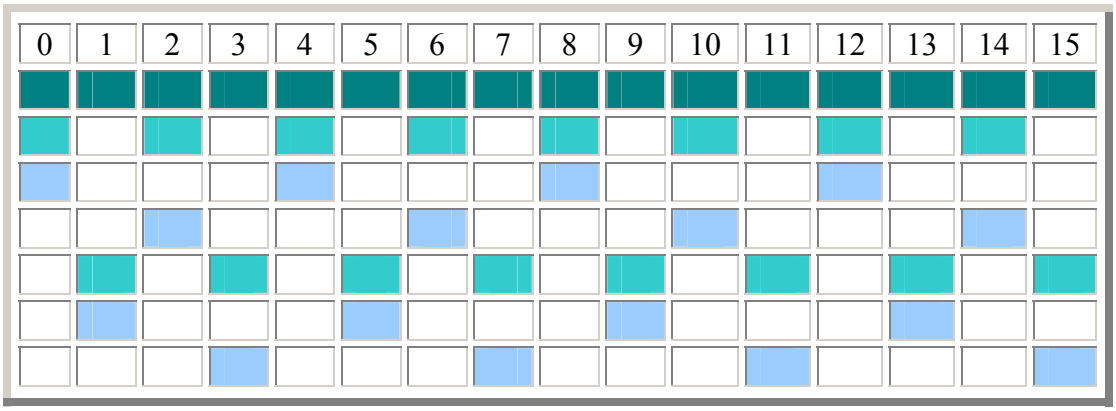


Figure 4.2 Interleaved split stride pattern

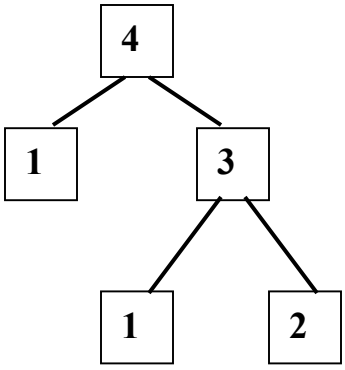
The memory access pattern corresponding to this tree is shown in Figure 4.3. For each leaf of the tree there is a corresponding walk through the vector. Each walk through the vector is represented by a different color. As can be seen from the stride pattern each right identity matrix is splitting the previous chunk

in S new chunks where S is the stride at which the vector is split and it is equal with the size of the right identity matrix in the corresponding factorization.
Using the notation start : stride : end, the elements of the vector x are initially accessed by 0:1:15. This is indicated in the top row of Figure 4.3. The two recursive calls access the subvectors 0:2:14 and 1:2:15. These accesses are interleaved as indicated in Figure 4.7. The two chunks are split in two again to obtain 0:4:12, 2:4:14, and 1:4:13, 3:4:15. However all the above splits have the same range between 0 and 15 even if the start address is different. For this reason we define the base of a chunk as the first address from the vector that is used. For example we can rewrite the chunk 3:4:14 as 0:4:15 and specify that the base is 3.
The entire trace for our example can be written as 0:1:15, 0:2:14, 0:4:12, 3:4:14, 1:2:15, 1:4:13, 4:4:15.

As a second example, consider the tree in Figure 4.4. This tree grows to the right, which means there are more left identity then right identity matrices in the factorization. So the vector will be split mostly into "cut" splits. A cut splits the vector into adjacent pieces with one subvector following another. Figure 4.4 shows the memory trace for the tree.

Lemma 1

Given a cache memory having size C = 2^c, block size B = 2^b and associativity A=2^a**.** The chunk $X^M{}_{b,S}$ fits inside the cache (i.e. all of its elements can be stored at the same time inside the cache if M <= $\left\lceil \frac{C}{\frac{A}{S}} \right\rceil * A$.



$$WHT_{16} = (WHT_2 \otimes I_8)(I_2 \otimes WHT_8) =$$
$$= (WHT_2 \otimes I_8)(I_2 \otimes (WHT_2 \otimes I_4)(I_2 \otimes WHT_4))$$

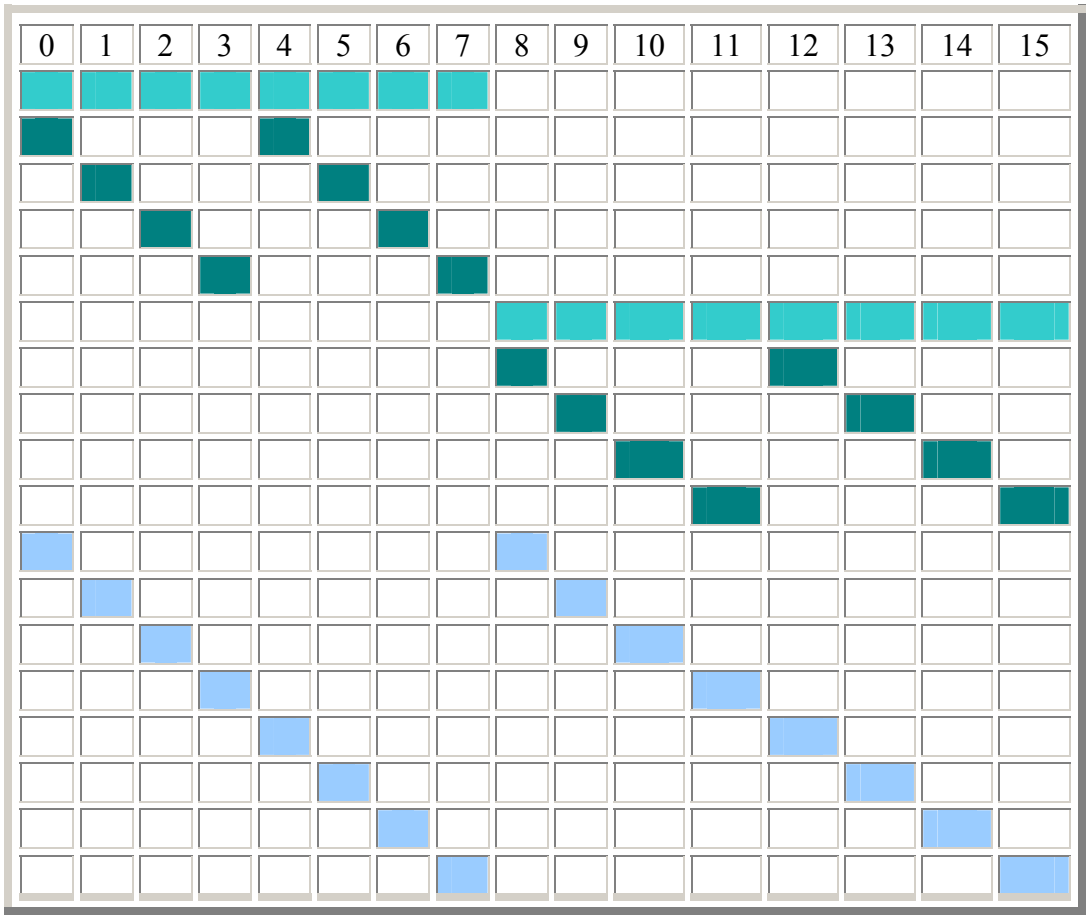


Figure 4.3 The cut split stride pattern

Proof: The number of vector elements that can be stored inside of a cache slot is B *A. To find the number of cache slots we have to divide the cache size C by the number of elements that can be stored inside of a cache slot. This means that

$NoOfCacheSlots = \frac{C}{B * A}$. The number of elements from a vector chunk that can be stored in the same block is $\left\lceil \frac{B}{S} \right\rceil$. Therefore, the number of chunk elements that can be stored inside the cache at the same time is $\left\lceil \left\lceil \frac{B}{S} \right\rceil * NoOfCacheSlots \right\rceil * A = \left\lceil \left\lceil \frac{B}{S} \right\rceil * \frac{C}{A * B} \right\rceil * A = \left\lceil \frac{\frac{C}{A}}{S} \right\rceil * A$.

The following lemma calculates the number of cache misses for the base case when W_m=small[m].

Lemma 2.

Assume W_m is small[m] and does not fit in cache, then the number of misses when computing $W_m X^M{}_{b,S}$ is equal to

1) 3 * M, if $S \geq C \; and \; A = 1$
2) $2 * M * \frac{1}{\left\lceil \frac{B}{S} \right\rceil}$, if $S < C \; or \; (S \geq C \; and \; A > 1)$

Proof: If $S \geq C \; and \; A = 1$ all accesses x[b],x[b + S],x[b],x[b + S], ….,x[b + (M2)* S], x[b + (M-1) * S], x[b + (M-2) * S], x[b + (M-1) * S] are misses. If $S < C \; or \; (S \geq C \; and \; A > 1)$ then the second access to x[b + i * S], x[b + (i+1) * S] are still in cache as are the entire blocks containing them.

The next lemma computes the number of cache misses generated during the computation of $(I_L \otimes W_m \otimes I_R) X^{M*R*L}{}_{b,S}$.

Lemma 3

The number of cache misses for calculating $(I_L \otimes W_m \otimes I_R) X^{M*R*L}{}_{b,S}$ when $X^{M*R*L}{}_{b,S}$ does not fit into the cache is equal with R * L* Misses($W_m X^M{}_{b',R*S}$) where $b' = iMR + k, 0 \leq i \leq L \; and \; 0 \leq k < R$.

Proof: For each $X^{M*R*L}{}_{b,s}$ we have R different chunks having the same memory address space but different bases. Since all of them are processed one after the other, the previous chunk wipes out the elements needed by the next chunk. After a memory address space has been processed the algorithm starts calculating the chunks with the next memory address space. Since there are L different memory address spaces and R different strides for each space and no elements are left in the cache from one chunk to the next, the total number of cache misses is R * L* Misses($W_m X^M{}_{b',R*S}$).

The theorem , uses these lemmas to derive a recurrence for the number of cache misses for an arbitrary WHT algorithm. To carry out the recursion it is necessary to recursively compute the number of misses $I_{2^l} \otimes W_n \otimes I_{2^r}$ instead of just W_n.

Theorem

Let W_n be a WHT algorithm to compute WHT_{2^n}. Assume n is split into t children $W_{n_1}, W_{n_2}, ..., W_{n_t}$ when W_{n_i} compute $WHT_{2^{n_i}}$. Then the number of misses to compute $I_{2^l} \otimes W_n \otimes I_{2^r}$ is

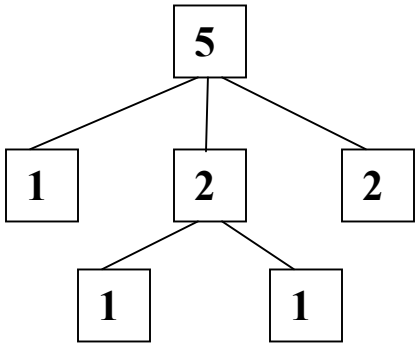
$$Misses(l, W_n, r) =$$

$$= \begin{cases} 2^n * \dfrac{1}{\left\lceil \dfrac{B}{2^r} \right\rceil} & , if\ 2^n * 2^r \le \left\lfloor \dfrac{\dfrac{C}{A}}{2^r} \right\rfloor * A \\ 3 * 2^n & , if\ W_n\ is\ small\ and\ A = 1 \\ 2 * 2^n * \dfrac{1}{\left\lceil \dfrac{B}{2^r} \right\rceil} & , if\ W_n\ is\ small\ and\ (2^r < C\ or\ (2^r \ge C\ and\ A > 1)) \\ \sum_{i=1}^{t} 2^{n-n_i} * M(n_1 + ... + n_{i-1} + l, W_{n_i}, n_{i+1} + ... + n_t + r) & \end{cases}$$

Proof:

The first base case occurs if the chunk accessed by W_n fits in cache. If W_n is small but the chunk does not fit in cache, the second base case follows from lemma 2. The general case follows from summing over the children in the split. Lemma 3 is used to calculate the number of cache misses of $I_{2^{n_1+.....+n_{i-1}}} \otimes W_{n_1} \otimes I_{2^{n_{i+1}+.....+n_t}}$. The first key observation is that there are 2^{n-n_i} recursive calls for node n_i and the stride for the recursive calls is $2^r * 2^{n_1+.....+n_{i-1}}$. A second key observation is that as the algorithms moves from one factor to the next, the previous factor wipe out the elements of the cache needed by the next factor. Thus the misses for each factor are independent.

4.4 Example calculation of the number of cache misses generated by the Walsh Hadamard transform



The trace for the tree above is represented in the table below. The number of cache misses are calculated for a memory system having the configuration C=8, B=1 and A=1.

Table 4.1 Example of WHT memory trace

Stride SR	Chunks	Chunk Range	Chunk No. Of Elements	Cache Misses
1	0:1:3 , 4:1:7 , 8:1:b, c:1:f , 10:1:13, 14:1:17, 18:1:1b , 1c:1:1f	4	4	8*4
4	0:4:4 , 8:4:c	8	2	2*2
8	0:8:8, 4:8:c	16	2	2*3*2
4	1:4:5 , 9:4:d,	8	2	2*2
8	1:8:9 , 5:8:d	16	2	2*3*2
4	2:4:6, a:4:e	8	2	2*2
8	2:8:a , 6:8:e	16	2	2*3*2
4	3:4:7 , b:4:f	8	2	2*2
8	3:8:b , 7:8:f	16	2	2*3*2
4	10:4:14, 18:4:1c	8	2	2*2
8	10:8:18 , 14:8:1c	16	2	2*3*2
4	11:4:15 , 19:4:1d	8	2	2*2
8	11:8:19, 15:8:1d	16	2	2*3*2
4	12:4:16 , 1a:4:1e	8	2	2*2
8	12:8:1a , 16:8:1e	16	2	2*3*2
4	13:4:17 , 1b:4:1f	8	2	2*2
8	13:8:1b , 17:8:1f	16	2	2*3*2
10	0:10:10, 1:10:11, 2:10:12, 3:10:13, 4:10:14, 5:10:15, 6:10:16, 7:10:17, 8:10:18, 9:10:19, a:10:1a, b:10:1b,c:10:1c, d:10:1d, e:10:1e, f:10:1f	32	2	16*3*2

Total number of cache misses = 4* 8 + 8*(2*2 +2*3*2)+16*3*2=32+8*16+96 =32+128+96=256

During the first step of the algorithm we check if $2^5 / 1 = 32 <= L1Capcity(1,8,1) = 8$.

The function $L1Capacity(S,C,A) = \left\lceil \frac{C}{\frac{A}{S}} \right\rceil * A$ represents the number chunk elements that can be stored inside the cache. If the number of elements from the chunk is <= the L1Capacity(S,C,A) then the chunk fits inside the cache.

Since the vector doesn't fit inside the cache we apply the $M(0,W_5,0)$ procedure. The number of cache misses can also be computed using the recurrence relation. Since W_5 does not fit in cache and it is not small $Misses(0,W_5,0) = 2^3 * Misses(3,W_2,0)+2^3*Misses(1,W_2,2)+2^4*Misses(0,W_1,4)$.

The rightmost child fits inside cache. The middle child is neither small nor fits inside the cache, so the number of misses must be computed recursively. The leftmost child does fit in the cache but falls under the second small case. Therefore the number of misses is equal to $2^3*2^2 + 2^3*[2*Misses(2,W_1,2)+2*Misses(1,W_1,3)]+2^4*2*3=32+2^3*[2*2+2*2*3]+96=$

$=32+128+96=256$

4.5 Analytical formula for the number of cache misses generated by an iterative WHT algorithm with the radix one

In this section I an analytical formula is obtained for the number of cache misses for an iterative tree with the radix one. Having a general procedure for calculating the number of cache misses is very helpful but obtaining an exact formula can reduce the running time of the procedure from linear time O(n) to constant time. The formula was determined based on the empirical data obtained by simulating various cache configurations and by studying the trace patterns; however , it easily follows from the recurrence relation issued in section 4.3.

During the derivation the following notation is used to fully describe the cache memory configuration:

The size of the WHT matrix is: $N = 2^n$

The size of the cache is: $C = 2^c$

The size of the block is: $B = 2^b$

The Associativity is A=2^a

Before starting the simulations we can observe that the stride $S = 2^s$ where s is an integer varying between $0 \le s \le n-1$. In other words the WHT vector is accessed for n times, each time using a different stride.

To determine the formula we start simulating a simple memory configuration consisting of a single level of cache C = 8 and having the block size B=1 and associativity 1.

We start to simulate the iterative version of the WHT since this one has simpler memory access pattern than the recursive version of the algorithm.
We can see that each memory location is accessed three different times for each stride. This is due to the way in which the elements of the vector are calculated

Table 4.2 show the memory trace for N=8. The WHT vector is accessed three times with the strides 1, 2 and 4.

The total number misses for the stride = 0 is N since all the N elements of the vector can be contained in the L1 cache. For S=2 and 4 the elements of the vector are already present in the L1 cache so there are additional 0 misses. This means that the total number of misses is equal with N for the case when N <= C and implicitly s < c. But the number of strides that are less than c is equal with c.

Table 4.2 Memory trace for N=8, C=8, B=1 and associtivity 1

Stride = 1	Stride = 2	Stride = 4
Misses=N	Misses = 0	Misses=0
r 0 1	r 0 1	r 0 1
r 1 1	r 2 1	r 4 1
r 0 1	r 0 1	r 0 1
r 1 1	r 2 1	r 4 1
w 0 1	w 0 1	w 0 1
w 1 1	w 2 1	w 4 1
r 2 1	r 1 1	r 1 1
r 3 1	r 3 1	r 5 1
r 2 1	r 1 1	r 1 1
r 3 1	r 3 1	r 5 1
w 2 1	w 1 1	w 1 1
w 3 1	w 3 1	w 5 1
r 4 1	r 4 1	r 2 1
r 5 1	r 6 1	r 6 1
r 4 1	r 4 1	r 2 1
r 5 1	r 6 1	r 6 1
w 4 1	w 4 1	w 2 1
w 5 1	w 6 1	w 6 1
r 6 1	r 5 1	r 3 1
r 7 1	r 7 1	r 7 1
r 6 1	r 5 1	r 3 1
r 7 1	r 7 1	r 7 1
w 6 1	w 5 1	w 3 1
w 7 1	w 7 1	w 7 1

If N > C at every pass through the vector's elements must be reloaded. For a stride S >= C the number of cache misses is 3*N according to Lemma 2. The formula for a memory configuration having B=1 and associativity 1 is

$$L1\_Misses(N,C) = \begin{cases} N & , for\ N \le C \\ c*N + 3*(n-c)*N, & for\ N > C \end{cases}$$

For N < B it is trivial that the number of cache misses is 1. For B < N < C the number of cache misses is equal to N / B. If N > C and S <C the number of cache misses is N /B and for a stride S >= C the number of cache misses is 3*N. The total number of cache misses is then c*N/B + (n - c) * 3 * N.

$$L1\_Misses(N,C,B) = \begin{cases} 1 & , \quad for\ N \le B \\ \frac{N}{B} & , \quad for\ B < N \le C \\ c * \frac{N}{B} + (n-c) * 3 * N, & for\ N > C \end{cases}$$

When the associativity A ≥ 2 we have the following cases. If N < C the associativity does not influence the number of cache misses. All the cache misses are generated when the elements of the vector are loaded for the first time into the memory.

If N > C then the number of cache misses is affected by the associativity. If the stride S > C then two successive memory accesses located at distance S will map into the same block. But because the associativity is bigger than 2 both of the two successive memory addresses can be stored in memory at the same time. This means that an associativity bigger than 2 will not reduce the cache misses further The final formula for the iterative case is

$$L1\_Misses(N,C,B,A) = \begin{cases} 1 & , \quad for\ N \le B \\ \frac{N}{B} & , \quad for\ B < N \le C \\ c * \frac{N}{B} + (n-c) * 3 * N, & for\ N > C\ and\ A = 1 \\ n * \frac{N}{B} & , \quad for\ N > C\ and\ A \ge 2 \end{cases}$$

4.6 Analytical formula for the number of cache misses for a recursive WHT algorithm with the radix one

In this section a formula for the radix one recursive WHT is derived. This formula also follows from the recurrence in section 4.3. The number of misses for the recursive case is less than the iterative case as expected due to better locality.

The stride S is a power of two and it is in the range $2^0 - 2^{n-1}$. First assume B=1 and A=1. When N ≤C there are N misses. When N>C the things are more complicated for a stride S ≥C. In this case all the memory accesses of the type x[b + i * S] and x[b + (i+1)*S] will map into the same memory location causing cache misses. The number of cache misses is 3*N (lemma 2) since all the memory accesses cause a cache misses and each memory location is accessed three different times and interleaved. Thus the number of cache misses satisfies the recurrence

$$L1\mathrm{R}(N,C) = \begin{cases} N, \; for \; N \le C \\ 3*N + 2*L1\mathrm{R}(\frac{N}{2}), \; for \; N > C \end{cases}$$

Expanding the recurrence leads to

$$\begin{aligned} & L1\mathrm{R}(N,C) = 3*N + 2*L1\mathrm{R}(\tfrac{N}{2}) = \\ & = 3*N + 2*(3*\tfrac{N}{2} + 2*L1\mathrm{R}(\tfrac{N}{4})) = \\ & = 3*N + 2*(3*\tfrac{N}{2} + 2*(3*\tfrac{N}{4} + 2*L1\mathrm{R}(\tfrac{N}{8}))) = \\ & = \underbrace{3*N + 2^1*3*\frac{N}{2^1} + 2^2*3*\frac{N}{2^2} + \ldots\ldots\ldots}_{n-c \; times} + N = \\ & = (n-c)*3*N + N \end{aligned}$$

and

$$L1\mathrm{R}(N,C) = \begin{cases} N, \; for \; N \le C \\ (3*(n-c)+1)*N, \; for \; N > C \end{cases}$$

Next consider the influence of the block size B. If N<B then the case is trivial and there is only one cache miss for the single block that will contain the whole vector. For B < N < C the number of cache misses is N/B. If the stride S >= N then the number of cache misses for a walk trough the vector is 3*N (Lemma 2) .

This leads to the recurrence

$$L1\mathrm{R}(N,C,B)=\begin{cases}1, for\ N\le B\\ \frac{N}{B}, for\ B<N\le C\\ 3*N+2*L1\mathrm{R}(\frac{N}{2}), for\ N>C\end{cases}$$

Solving the above recurrence results the formula

$$L1\mathrm{R}(N,C,B)=\begin{cases}1, for\ N\le B\\ \frac{N}{B}, for\ B<N\le C\\ 3*(n-c)*N+\frac{N}{B}, for\ N>C\end{cases}$$

The associativity only influences the number of cache misses for a stride S >= N.

In this case the number of cache misses is reduced from 3*N to N/B, and

$$L1\mathrm{R}(N,C,B,A)=\begin{cases}1, for\ N\le B\\ \frac{N}{B}, for\ B<N\le C\\ 3*N+2*L1\mathrm{Re}c(\frac{N}{2}), for\ N>C\ and\ A=1\\ \frac{N}{B}+2*L1\mathrm{Re}c(\frac{N}{2}), for\ N>C\ and\ A>1\end{cases}$$

Solving this recursive recurrence we obtain

$$L1\mathrm{R}(N,C,B,A)=\begin{cases}1, for\ N\le B\\ \frac{N}{B}, for\ B<N\le C\\ 3*(n-c)*N+\frac{N}{B}, for\ N>C\ and\ A=1\\ (n-c+1)*\frac{N}{B}, for\ N>C\ and\ A>1\end{cases}$$

Chapter 5: Cache miss distribution of random WHT algorithms

In this chapter we investigate the number of cache misses for WHT algorithms. An empirical distribution of the number of cache misses for random WHT algorithms is studied. The effect of cache parameters on the distribution is investigated.

5.1 Random generation of WHT partition trees

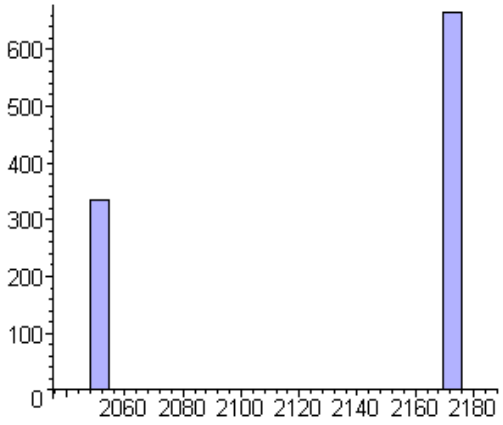
A random partition tree can be generated using the following procedure. First generate a random number j between 0 and 2^{n-1}-1, and then convert the number j to its binary representation. Also consider a string of n 1's . The bits of the number j are placed between adjacent 1's. Adjacent 1's are added until a 1 bit is encountered to obtain an ordered partition. For example the partition generated from the number j = 0100110 is [2,3,1,2]. The process is illustrated below.

```
  2  |     3     | 1 |    2      =  [2,  3,  1 , 2]
1  1   1   1   1   1   1   1
 .    .   .   .    .   .    .
 0    1   0   0    1   1    0
```

The procedure is applied recursively to each integer in the partition, until a cutoff size is reached, to produce a partition tree. An implementation of this procedure is in Appendix B.

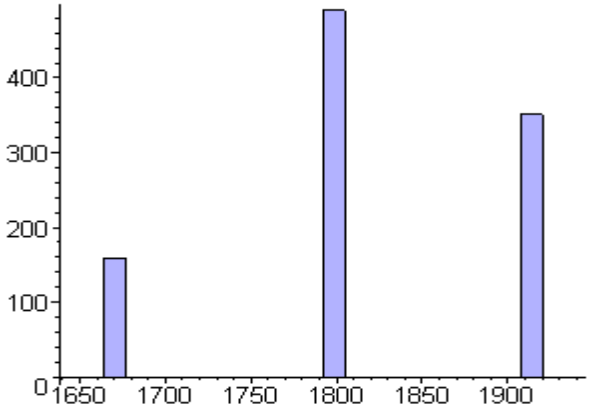
5.2 Empirical data

The distribution of cache misses is obtained for 1000 random trees of size 7 with leaf nodes equal to 1. Figure 5.1 shows histogram of cache misses with varying cache size and block size and associativity 1. In addition to the histograms a sample tree from the clusters with the maximum and minimum cache misses is given.



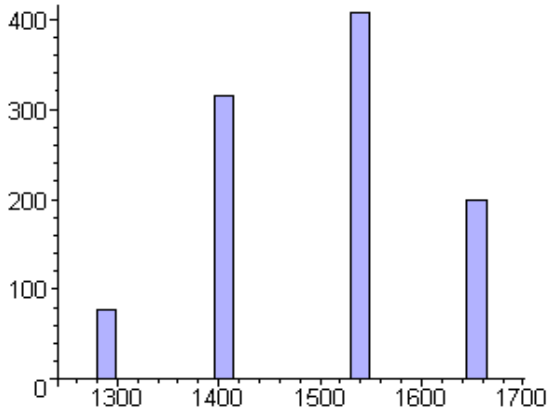
Min: [[1, 1, 1, [1, 1]], [1, 1]]
Max: [[1, 1], [1, 1, 1, 1], 1]

Cache Size = 4



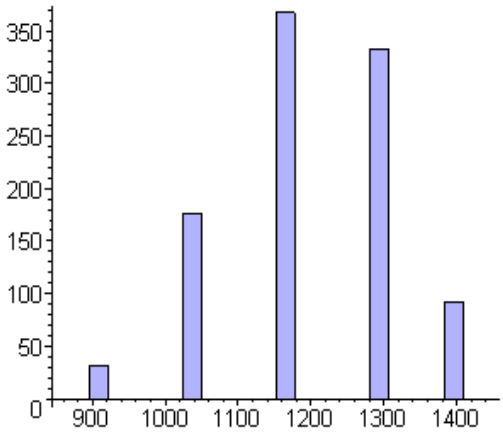
[1, [1, 1], 1, [1, 1, 1]]
[1, [[1, 1], 1, 1], 1, 1]

Cache Size = 8



Min: [[1, 1], 1, [[[1, 1], 1], 1]]
Max: [[1, 1], [[1, 1], 1], 1, 1]

Cache Size = 16



[[1, 1], [[1, 1, [1, 1]], 1]]
[[[1, 1], 1, 1, 1], 1, 1]

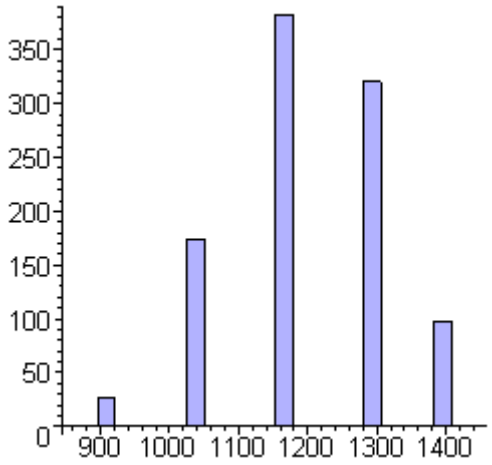
Cache Size = 32

Figure 5.1 The data cache size influence on the cache miss distribution

A discrete distribution is obtained, with cache misses in discrete clusters. The number of cache misses increases as the cache size increases. In fact the number of clusters appears to be the logarithm of the cache size.

In the next experiment the cache size is fixed and the block size is varied. Figure 5.2 show, the distributions for block size 1, 2, 4 and 8. In general the number of misses decreases, but after block size 2, the minimum number of misses remains around 750. The cache size is 32 and the associativity is set to one.

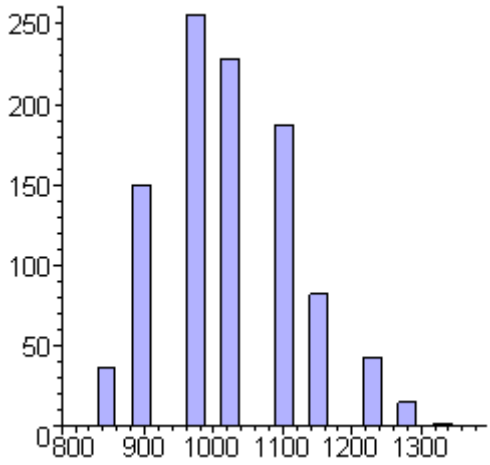
More clusters are obtained as the block size increases with the new clusters appearing near the old clusters. The tendency is to redistribute trees from clusters having a high number of cache misses to clusters with a smaller number of cache misses.



Min: [[1, 1], [[1, 1, [1, 1]], 1]]

Max: [[[1, 1], 1, 1, 1], 1, 1]

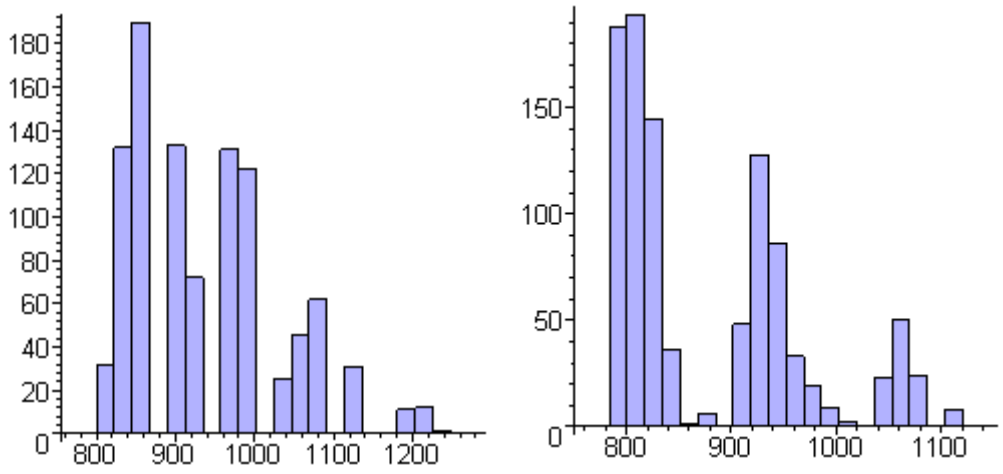
Block Size = 1



[1, 1, [[1, 1], [1, 1], 1]]

[1, [1, 1, 1, 1, 1], 1]

Block Size = 2



Min: [[1, 1], [[1, 1, 1], 1, 1]] [[1, 1], [1, 1, 1, 1, 1]]

Max: [1, [1, 1, 1, 1], 1, 1] [[1, [[1, 1], 1], 1], 1, 1]

Block Size = 4 Block Size = 8

Figure 5.2 The influence of the block size on the cache miss distribution

In the last experiment the cache size and block size are fixed and the associativity was varied. Using the same set of input trees with the cache size set to 32 and the block size set to 4, the associativity ranged from 1 to 8.

As the associativity is increased from 1 to 2 the minimum and maximum number of cache misses obtained is reduced dramatically. However further increasing of the associativity does not influence the minimum and maximum number of cache misses. This is due to the successive memory locations that are accessed twice by the read operations and once by the write operations.

So when the stride is bigger than the cache size and the associativity is 2 there are at least two successive memory locations that can be stored at the same time inside the cache. For this reason the number of cache misses is reduced significantly. Another tendency that can be observed is that the trees are now

redistributed to the left part of the distribution. This means there is a larger number of trees that generate a small number of cache misses.

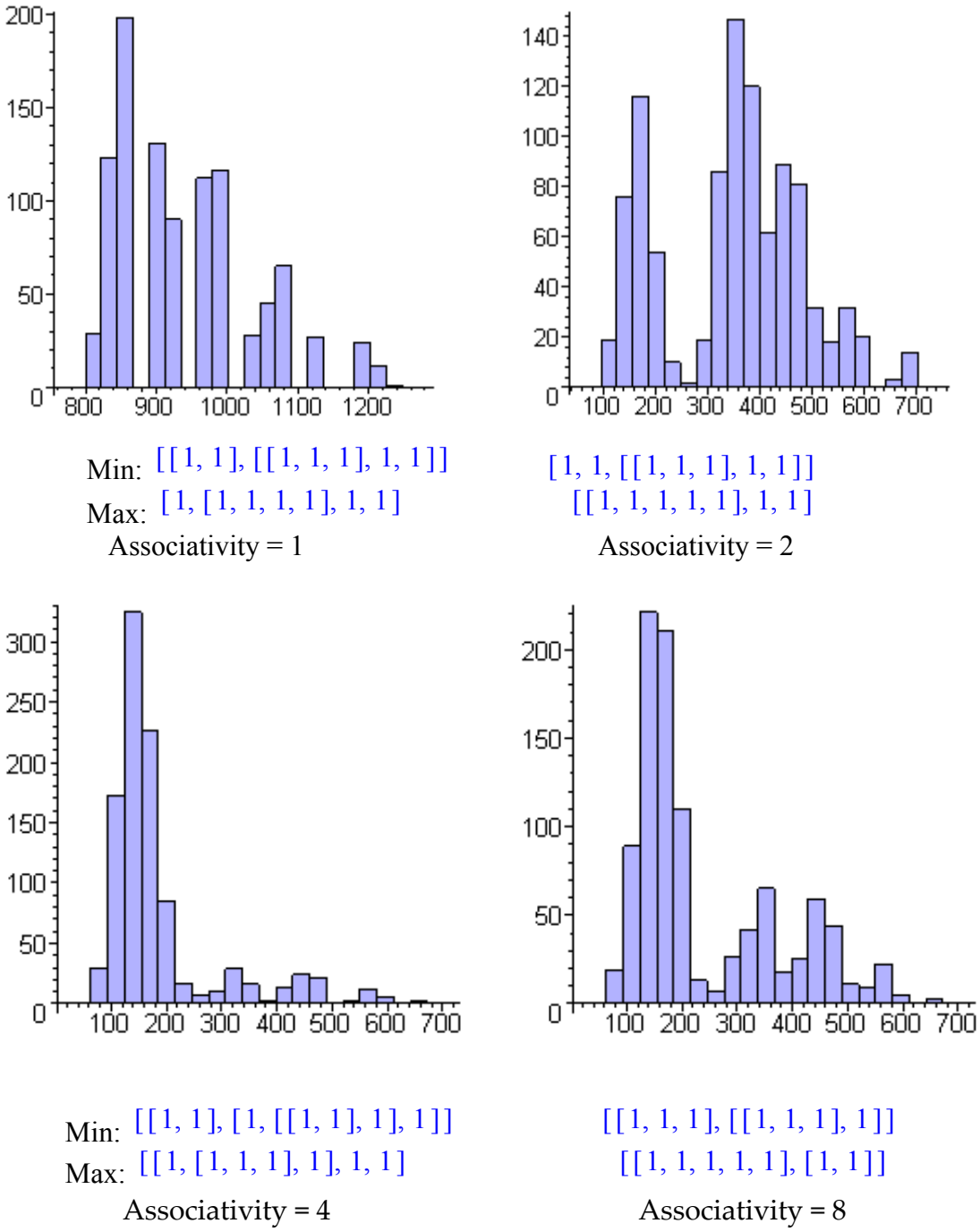


Figure 5.3 The influence of associativity on the cache miss distribution

A key observation from all of the data in this chapter is that there is a wide range of performance (cache misses), and choosing a good algorithm can dramatically improve performance. Depending on the machine configuration the number of cache misses can be reduced by one third to even 7 times less than the worst possible tree for that machine.

Chapter 6: Conclusion

The formula presented in this thesis includes the key cache parameters: cache size, block size and associativity. Having such a formula allows algorithms to be compared and optimized on different architectures without having to actually run the program. Preliminary observations showed the influence of cache design on the number of cache misses for different WHT algorithms. This work provides tools to design and search for algorithms with good cache behavior.

In future work one could try to derive similar models for other families of algorithms, and use the resulting model to search for good implementations. In terms of the WHT it would be interesting to analytically determine the optimal algorithm in terms of cache misses and to prove properties about the distribution of cache misses for these algorithms and the relationship of the distribution to cache design parameters.

In this thesis we investigate the number of cache misses required by different algorithms to compute the WHT transform, an important algorithm used in signal and image processing. In previous work it was shown that different algorithms have dramatically different performance despite having the same number of arithmetic operations. Search was used to find a good algorithm. While some intuitive explanations were given to explain the variations in performance a performance model and clear explanation were not given. It was conjectured that cache utilization plays an important role in performance. In this

work a formula was obtained for the number of cache misses for an arbitrary WHT algorithms, and it was shown that the WHT algorithms exhibit a wide range in the number of cache misses.

Bibliography

[1] John L. Hennessy, Davis A. Patterson. Computer architecture: a quantitative approach (2nd edition), Morgan Kaufmann, January 1996.

[2] Harvey Cragon. Computer architecture and implementation, Cambridge University Press, January 2000.

[3] Jeremy Johnson, Markus Püskel. In search of the optimal Walsh-Hadamard transform. Proc. ICASSP 2000, pp. 3347--3350

[4] Harvey Cragon. Memory systems and pipelined processors. Jones & Bartlett Pub. January 1996

[5] Steven A. Przybylski. Cache and memory hierarchy design: a performance directed approach, Morgan Kaufmann, November 1990.

[6] Neungsoo Park, Viktor Prasanna. Cache Conscious Walsh-Hadamard transform. Proc. ICASSP 2001, Vol. II

[7] Randy Allen, Ken Kennedy. Optimizing compilers for modern architectures: a dependence-based approach, Morgan Kaufmann, October 2001.

[8] Michael Wolfe, Leda Ortega. High performance compilers for parallel computing, Addison-Wesley, June 1995.

[9] Joshua P. MacDonald, Ben Y Zhao. Cache Performance of Indexing Data Structures

[10] Jun Rao, Kenneth Ross. Cache Conscious Indexing for Decision Support in Main Memory, Columbia University Technical Report CUCS – 019-98, Dec 1, 1998

[11] J. S. Vitter and E. A. M. Shriver. ``Algorithms for Parallel Memory I: Two-Level Memories,'' double special issue on Large-Scale Memories in Algorithmica, 12(2-3), 1994, 110-147. A shortened version appears in ``Optimal Disk I/O with Parallel Block Transfer,'' Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC '90), Baltimore, MD, May 1990, 159-169.

[12] J. S. Vitter and E. A. M. Shriver. ``Algorithms for Parallel Memory II: Hierarchical Multilevel Memories,'' double special issue on Large-Scale Memories in Algorithmica, 12(2-3), 1994, 148-169. A shortened version appears in ``Optimal Disk I/O with Parallel Block Transfer,'' Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC '90), Baltimore, MD, May 1990, 159-169.

[13] Siddhartha Chatterjee, Erin Parker. Exact Analysis of the Cache Behavior of Nested Loops. PLDI 2001.

[14] Hung-Jen Huang. Performance Analysis of an Adaptive Algorithm for the Walsh Hdamard Transform, MS Thesis 2003, Drexel University

[15] Mateo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In ICASSP ’98, volume 3, pages 1381 – 1384, 1998. http://www.fftw.org

[16] J. R. Johnson, Markus Püskel. WHT: An adaptable library for computing the Walsh-Hadamard transform, 1999. http://www.ece.cmu.edu/~spiral/wht.html

Appendix A: Code from the WHT package

This appendix contains code from the WHT package [3] referred to in this thesis. Code for small[1] (apply_small1) and small[2] (apply_small2) are presented as is the code for an arbitrary split node (apply_split). The code presented for small[1] and small[2] contains print statements used to generate the memory trace used by the cache simulator. Complete source code for the package is available from http://www.ece.cmu.edu/~spiral/

```
void apply_small1(Wht *W, long S, wht_value *x)
{
  wht_value t0;
  wht_value t1;
  t0 = x[0] + x[S];
  printf("r %x 1\n",&x[0]-start_address);
  printf("r %x 1\n",&x[0]-start_address+S);
  t1 = x[0] - x[S];
  printf("r %x 1\n",&x[0]-start_address);
  printf("r %x 1\n",&x[0]-start_address+S);
  x[0] = t0;
  x[S] = t1;
  printf("w %x 1\n",&x[0]-start_address);
  printf("w %x 1\n",&x[0]-start_address+S);
}

void apply_small2(Wht *W, long S, wht_value *x)
{
  wht_value t0;
  wht_value t1;
  wht_value t2;
  wht_value t3;
  t0 = x[0] + x[S];
  printf("r %x 1\n",&x[0]-start_address);
  printf("r %x 1\n",&x[0]-start_address+S);
  t1 = x[0] - x[S];
  printf("r %x 1\n",&x[0]-start_address);
  printf("r %x 1\n",&x[0]-start_address+S);
  t2 = x[2 * S] + x[3 * S];
  printf("r %x 1\n",&x[0]-start_address+2*S);
```

```
  printf("r %x 1\n",&x[0]-start_address+3*S);
  t3 = x[2 * S] - x[3 * S];
  printf("r %x 1\n",&x[0]-start_address+2*S);
  printf("r %x 1\n",&x[0]-start_address+3*S);
  x[0] = t0 + t2;
  x[2 * S] = t0 - t2;
  x[S] = t1 + t3;
  x[3 * S] = t1 - t3;
  printf("w %x 1\n",&x[0]-start_address);
  printf("w %x 1\n",&x[0]-start_address+2*S);
  printf("w %x 1\n",&x[0]-start_address+S);
  printf("w %x 1\n",&x[0]-start_address+3*S) }

/* Split WHT
   ---------
   A WHT_N can be split into k WHT's of smaller size
   (according to N = N_1 * N_2 * ... * N_k):

                        WHT_N_1 tensor 1_(N/N_1) *
     1_N_1              tensor WHT_N_2 tensor 1_(N/N_1N_2) *
       ...
       ...
     1_(N_1...N_(k-1)) tensor WHT_N_k

   The WHT_N is performed by stepping through this product
   from right to left.
*/

static void apply_split(Wht *W, long S, wht_value *x) {
  int nn;
  long N, R, S1, Ni, i, j, k;
#ifdef IL_ON
  int nIL;
#endif

#ifdef PCL_PROFILE
  PCL_CNT_TYPE result;
  PCL_FP_CNT_TYPE fp_result;

/*
  if( PCLquery( &event, 1, PCL_MODE_USER ) != PCL_SUCCESS ) {
     printf( "blah\n" );
     exit(5);
  }
*/
#endif

  nn = W->priv.split.nn;
```

```
 if (nn > SPLIT_MAX_FACTORS) {
   Error_int2("%d > %d, too many factors", nn, SPLIT_MAX_FACTORS);
 }

 N  = W->N;
 R  = N;
 S1 = 1;

 /* step through the smaller whts */
 for (i = nn-1; i >= 0; i--) {
   Ni = W->priv.split.ns[i];
 R /= Ni;
#ifdef PCL_PROFILE
   PCLread(&result, &fp_result, 1);
   W->priv.split.Ws[i]->pcl -= result;
#endif

#ifdef IL_ON
   if((W->priv.split.Ws[i])->type != wht_small_il) {
      for (j = 0; j < R; j++)
           for (k = 0; k < S1; k++)
             wht_apply(W->priv.split.Ws[i], S1*S, x+k*S+j*Ni*S1*S);
   } else {
      nIL = (W->priv.split.Ws[i])->nILNumber;
for (j = 0; j < R; j++)
           for (k = 0; k < S1; k+=nIL)
             wht_apply_4_para(W->priv.split.Ws[i], S1*S, S, x+k*S+j*Ni*S1*S);
   }
#else
   for (j = 0; j < R; j++)
     for (k = 0; k < S1; k++)
       wht_apply(W->priv.split.Ws[i], S1*S, x+k*S+j*Ni*S1*S);
#endif

#ifdef PCL_PROFILE
   PCLread(&result, &fp_result, 1);
W->priv.split.Ws[i]->pcl += result;
#endif
   S1 *= Ni;
 }
```

Appendix B: Maple program for calculating the number of cache misses generated by WHT algorithms

The appendix contains the program for calculating the number of cache misses generated by a WHT algorithm, which was written in Maple v8. The program uses a list representation for partition trees. For example the partition tree from the 4.4 paragraph is represented as [1, [1, 1], 2]. The procedure Capacity(S,C,A) returns the number of elements from a chunk with stride S that can be stored inside a cache of size C and associativity A. A chunk fits into the cache if the chunk number of elements is less or equal to Capacity(S,C,A). The CacheMisses(T,C,B,A) procedure returns the number of cache misses generated by a WHT algorithms represented by the tree T. The program contains also a procedure for generating a random partition tree following the method presented in Chapter 5.

```
> TreeSize := proc(T)
>   if type(T,posint) then
>     RETURN(T);
>   else
>     foldl(`+`,0,op(map(TreeSize,T)));
>   fi;
> end;
```

```
TreeSize := proc (T)
    if type(T, posint) then RETURN(T)
    else foldl(`+`, 0, op(map(TreeSize, T)))
    end if
end proc
```

```
> MyT:=[1,[1,1],2];
```

$$MyT := [1, [1, 1], 2]$$

```
> nops(MyT);
```

$$3$$

```
> TreeSize(MyT);
```

$$5$$

```
> Capacity := proc(S,C,A)
```

```
>   return ceil((C/A)/S)*A;
> end;
```

$$Capacity := \mathbf{proc}\ (S, C, A)\ \mathbf{return}\ \ \mathrm{ceil}(C/(A \times S)) \times A\ \mathbf{end\ proc}$$

```
> LeafNode := proc(T)
>  return type(T,posint);
> end;
```

$$LeafNode := \mathbf{proc}\ (T)\ \mathbf{return}\ \ \mathrm{type}(T, posint)\ \mathbf{end\ proc}$$

```
> LeafNode([1,1]);
```

$$false$$

```
> CacheMisses := proc(T,C,B,A)
>
>   if (2^TreeSize(T)<= Capacity(1,C,A)) then
>      return ((2^TreeSize(T))/B);
>   else
>     return WHTMisses(1,T,1,C,B,A);
>   fi;
> end;
```

$$CacheMisses := \mathbf{proc}\ (T, C, B, A)$$
$$\quad \mathbf{if}\ 2\text{^}\mathrm{TreeSize}(T) \le \mathrm{Capacity}(1, C, A)\ \mathbf{then}\ \mathbf{return}\ \ 2\text{^}\mathrm{TreeSize}(T)/B$$
$$\quad \mathbf{else}\ \mathbf{return}\ \ \mathrm{WHTMisses}(1, T, 1, C, B, A)$$
$$\quad \mathbf{end\ if}$$
$$\mathbf{end\ proc}$$

```
> WHTMisses := proc(L,T,R,C,B,A)
> local ParentSize, LSiblings, RSiblings, ChildrenMisses, S,i;
>   ParentSize := TreeSize(T);
> LSiblings:= TreeSize(T);
> ChildrenMisses:=0;
> RSiblings:=0;
> for i from nops(T) by -1 to 1 do
> S:=R*(2^RSiblings);
> LSiblings:=LSiblings - TreeSize(T[i]);
>
> if  (2^TreeSize(T[i])) <= Capacity(S,C,A) then
ChildrenMisses:=ChildrenMisses+((2^ParentSize) /ceil(B/R));
> else
>   if LeafNode(T[i]) then
>   if (S >= C and A=1) then  ChildrenMisses := ChildrenMisses + 3 *
(2^ParentSize);
>             fi;
>             if ( S <  C and A = 1 ) or
>                ( S >= C and A > 1 ) or
>                ( S <  C and A > 1 )  then ChildrenMisses :=
ChildrenMisses + 2 *(2^ParentSize)/ceil(B/R);
>             fi;
>
>   else ChildrenMisses := ChildrenMisses + (2^( ParentSize -
TreeSize(T[i]))) *
WHTMisses(L*(2^LSiblings),T[i],R*(2^RSiblings),C,B,A);
>   fi;
>
```

```
> fi;
> RSiblings := RSiblings + TreeSize(T[i]);
> od;
> return ChildrenMisses;
>
> end;
>
WHTMisses := proc (L, T, R, C, B, A)
local ParentSize, LSiblings, RSiblings, ChildrenMisses, S, i;
    ParentSize := TreeSize(T);
    LSiblings := TreeSize(T);
    ChildrenMisses := 0;
    RSiblings := 0;
    for i from nops(T) by -1 to 1 do
        S := R×2^RSiblings;
        LSiblings := LSiblings − TreeSize(T[i]);
        if 2^TreeSize(T[i]) ≤ Capacity(S, C, A) then
            ChildrenMisses := ChildrenMisses + 2^ParentSize/ceil(B/R)
        else
            if LeafNode(T[i]) then
                if C ≤ S and A = 1 then
                    ChildrenMisses := ChildrenMisses + 3×2^ParentSize
                end if;
                if S < C and A = 1 or C ≤ S and 1 < A or S < C and 1 < A then
                    ChildrenMisses :=
                    ChildrenMisses + 2×2^ParentSize×ceil(B/R)
                end if
            else ChildrenMisses := ChildrenMisses +
                2^(ParentSize − TreeSize(T[i]))×
                WHTMisses(L×2^LSiblings, T[i], R×2^RSiblings, C, B, A)
            end if
        end if;                                          end proc
        RSiblings := RSiblings + TreeSize(T[i])
    end do;
    return ChildrenMisses

> T2:=[[2,1],1];
>
                                T2 := [[2, 1], 1]

> T1:=[1,[1,2]] ;
                                T1 := [1, [1, 2]]
```

```
> TreeSize(T2);
```

$$4$$

```
> CacheMisses(T2,2,1,1);
```

$$112$$

```
> MyT;
```

$$[1, [1, 1], 2]$$

```
> TreeSize(MyT);
```

$$5$$

```
> trace(WHTMisses(1,MyT,1,8,1,1));
> LeafNode([1, 1]);
```

$$false$$

```
> TreeSize(MyT);
```

$$5$$

```
> BinToPart := proc(t,b)
> local bp, P, n;
> bp := [op(convert(b,list)),1];  P := [];
> while bp <> [] do
>   n := 1;
>   while bp[1] = 0 do
>     n := n+1;
>     bp := bp[2..-1];
>   od;
>   P := [n,op(P)];
>   bp := bp[2..-1];
> od;
> RETURN(P);
> end;
```

```
BinToPart := proc (t, b)
local bp, P, n;
    bp := [op(convert(b, list)), 1];
    P := [ ];
    while bp ≠ [ ] do
        n := 1;
        while bp[1] = 0 do n := n + 1; bp := bp[2 .. -1] end do ;
        P := [n, op(P)];
        bp := bp[2 .. -1]
    end do ;
    RETURN(P)
end proc
```

```
> OrderedPartition := proc(n)
>   local c, increment;
>   c := `FAIL`;
```

```
>
> increment := proc(n,c::array(1..nonnegint))
>     local i,j;
>     for i to n do
>       c[i] := c[i] + 1;
>       if c[i] < 2 then
>         RETURN(eval(c));
>       fi;
>       c[i] := 0;
>     od;
>     FAIL;
>   end;
>
>   proc()
>     if c = `FAIL` then c := array([0$(n-1)]); else c := increment(n-
1,c) fi;
>     if c <> `FAIL` then
>        RETURN(BinToPart(n,c));
>     else
>        RETURN(`FAIL`);
>     fi;
>   end;
> end;
>
>
```

OrderedPartition := **proc** (*n*)
local *c*, *increment*;
c := *FAIL*;
increment := **proc** (*n*, *c*::array(1 .. *nonnegint*))
local *i*, *j*;
for *i* **to** *n* **do**
c[*i*] := *c*[*i*] + 1;
if *c*[*i*] < 2 **then** RETURN(eval(*c*)) **end if** ;
c[*i*] := 0 **proc** ()
end do ; **if** *c* = *FAIL* **then** *c* := array([0 $ (*n* − 1)])
FAIL **else** *c* := *increment*(*n* − 1, *c*)
end proc ; **end if** ;
if *c* ≠ *FAIL* **then** RETURN(BinToPart(*n*, *c*)) **end proc**
else RETURN(*FAIL*)
end if
end proc

```
> BinToPart(5,[0,0,0,0]);
```

[5]

```
> BinToPart(5,[1,0,0,1]);
```

[1, 3, 1]

```
> BinToPart(5,[1,1,0,1]);
```

$$[1, 2, 1, 1]$$

```
> p4 := OrderedPartition(4);
```

$$p4 := \textbf{proc}\ ()$$
$$\quad \textbf{if}\ c = FAIL\ \textbf{then}\ c := \text{array}([0\ \$\ (4-1)])\ \textbf{else}\ c := increment(4-1, c)\ \textbf{end if}\ ;$$
$$\quad \textbf{if}\ c \neq FAIL\ \textbf{then}\ \text{RETURN}(\text{BinToPart}(4, c))\ \textbf{else}\ \text{RETURN}(FAIL)\ \textbf{end if}$$
$$\textbf{end proc}$$

```
> for i from 1 to 2^3 do
> p4();
> od;
```

$$[4]$$
$$[3, 1]$$
$$[2, 2]$$
$$[2, 1, 1]$$
$$[1, 3]$$
$$[1, 2, 1]$$
$$[1, 1, 2]$$
$$[1, 1, 1, 1]$$

```
> BinaryInt := proc(n,ind::nonnegint)
>   local t,i,D,tind;
>   if ind >= 2^n then
>     ERROR("integer must be less than the product of the radices");
>   fi;
>   D := []; tind := ind;
>   for i from n by -1 to 1 do
>     D := [tind mod 2,op(D)];
>     tind := floor(tind/2);
>   od;
>   RETURN(D);
> end;
```

```
BinaryInt := proc (n, ind::nonnegint)
local t, i, D, tind;
    if 2^n ≤ ind then
        ERROR("integer must be less than the product of the radices")
    end if;
    D := [];
    tind := ind;
    for i from n by -1 to 1 do D := [tind mod 2, op(D)]; tind := floor(1/2×tind)
    end do;
    RETURN(D)
end proc
```

```
> randint := rand(1..2^100):
> RandWHT := proc(n,CUTOFF)
> local randsplit,c,n1;
>
> if ((n <= CUTOFF)) then RETURN(n); fi;
> randsplit := randint() mod (2^(n-1)-1);
> c := BinToPart(n,BinaryInt(n-1,randsplit+1));
> #RETURN(map(RandWHT,c));
> RETURN(map(n1->RandWHT(n1,CUTOFF),c));
> end;
```

```
RandWHT := proc (n, CUTOFF)
local randsplit, c, n1;
    if n ≤ CUTOFF then RETURN(n) end if;
    randsplit := randint() mod (2^(n − 1) − 1);
    c := BinToPart(n, BinaryInt(n − 1, randsplit + 1));
    RETURN(map(n1 → RandWHT(n1, CUTOFF), c))
end proc
```

```
> cmin:=10000000;cmax:=0;
```

$$cmin := 10000000$$

$$cmax := 0$$

```
> MinTree:=[];MaxTree:=[];
```

$$MinTree := [\ ]$$

$$MaxTree := [\ ]$$

```
> data := [];
```

$$data := [\ ]$$

```
> for i from 1 to 1000 do
> RandTree:=RandWHT(7,1);
> cachem:=CacheMisses(RandTree,32,4,8);
> if (cachem < cmin) then cmin:=cachem; MinTree:=RandTree; fi;
```

```
> if (cachem > cmax) then cmax:=cachem; MaxTree:=RandTree; fi;
> data := [CacheMisses(RandTree,32,4,8),op(data)]:
> od:
>
>
>
> MinTree;CacheMisses(MinTree,4,8,1);
```

$$[[1, 1, 1], [[1, 1, 1], 1]]$$

$$1968$$

```
> [[1, 1, 1], 1, [1, 1, 1]];
```

$$[[1, 1, 1], 1, [1, 1, 1]]$$

```
> MaxTree;
```

$$[[1, 1, 1, 1, 1], [1, 1]]$$

```
> nops(data);
```

$$1000$$

```
> max(op(data));
```

$$2176$$

```
> min(op(data));
```

$$2048$$

```
> with(stats);
```

```
Warning, these names have been redefined: anova, describe, fit,
importdata, random, statevalf, statplots, transform
```

$$[anova, describe, fit, importdata, random, statevalf, statplots, transform]$$

```
> with(stats[statplots]);
```

```
Warning, these names have been redefined: boxplot, histogram,
scatterplot, xscale, xshift, xyexchange, xzexchange, yscale, yshift,
yzexchange, zscale, zshift
```

$$[boxplot, histogram, scatterplot, xscale, xshift, xyexchange, xzexchange, yscale, yshift, yzexchange, zscale, zshift]$$

```
> histogram(data,area=count,numbars=20);
```