White Paper

Brian Forde
Senior Platform Applications Engineer

John Browne
Platform Solutions Architect

Intel Corporation

Optimizing Non-Sequential Data Processing Applications

March 2010

Document Number: 323654-001

Abstract

Data processing applications that manage large numbers of data flows, either by indexing or pointer chasing, may be subject to frequent instruction pipeline stalls due to data cache misses. This paper outlines techniques for use on Intel® Architecture processors to lessen the effect of instruction pipeline stalls in certain application designs.

It is assumed that the reader is familiar with the ‘C’ programming language, IA-32 assembly language, and the GNU* ‘C’ compiler (GCC).

Contents

Figures

Introduction to Cache

High-performance CPUs depend on the availability of instructions and data to the processor cores (the execution units) for their high throughput. When there are delays in this availability, stalls may occur in the instruction pipeline of the execution unit. To reduce the occurrence of stalls, instruction and data caches that store frequently used data are placed in proximity to the processor cores, thus reducing the need for the execution units to fetch data from external system memory.

In applications that process large amounts of independent data (video streaming, communications protocols, database servers, etc.), there are basically two types of data: non-temporal data (for example, control structures associated with a single video link), which changes less frequently, and temporal data (for example, video frame data), which changes more frequently. It is desirable that the non-temporal data remain in-cache, while the temporal data are read from the data source, usually via external system memory. It is possible, when the volume of data passing through a processor is very large, that the non-temporal data may be evicted from cache.

When the execution unit accesses a memory location whose data is not present in cache, the cache subsystem must fetch that data from external memory. Loading the data from external memory into the cache incurs a penalty (memory read latency) normally measured in nanoseconds. Typically the data is read from memory (by the cache controller / memory controller) in blocks of the same size as the data cache line.

If there is a minimal amount of data processing/manipulation to be done per cache line (memory) read, then the execution bottleneck is the memory interface. In other words, the rate of progression of the execution unit through the task is determined by the rate at which data can be fetched from external memory.

Intel® Architecture processors provide a number of hardware-triggered data prefetch mechanisms that monitor application data access patterns. On detection of certain access patterns, the prefetch logic triggers a data prefetch operation automatically. These mechanisms are very effective in certain types of applications. The hardware prefetchers can reduce the effective latency of cache misses.

In addition to the hardware prefetch logic, the latest Intel Architecture processors incorporate multiple non-blocking 'Fill Buffers' to track outstanding cache misses beyond the data cache. Multiple 'Load' and 'Store' Buffers are also implemented in the Memory Execution Unit. These, and other features, allow multiple memory read operations to be in flight simultaneously.

In the majority of applications, data is accessed in a predictable fashion (for example, video processing, where each block of a frame follows sequentially in memory or at least at a predictable 'stride' from the previous). In such 'sequential access' applications it is relatively easy for the prefetch logic, implemented in hardware, to predict which data will be accessed next by the software application code.

Certain packet processing applications, in particular those that include Segmentation and Reassembly (SAR) algorithms, suffer from a level of 'unpredictability' in the data access stride. For such applications, the prefetch logic implemented in hardware may be unable to predict accurately which data cache lines should be prefetched next. In the worst case, the prefetch logic may mis-predict, and cause useful data to be evicted from cache, only to be replaced by data not yet needed by the execution unit.

Software-controlled prefetching is available on Intel Architecture processors, allowing the software great flexibility in maintaining the content of the cache hierarchy. The prefetch instruction is a hint to the processor, and may not be honored.

In most cases, prefetching (either hardware- or software-triggered) the data from memory into the cache hierarchy reduces or eliminates the latency associated with access to external memory. In addition, if the prefetch operation is performed in a timely manner, the effects of any additional latency due to remotely located data in a Non-Uniform Memory Architecture (NUMA) system may also be alleviated.

The Data Translation Lookaside Buffer (DTLB) in the processor under consideration supports either 4 Kbyte or 2M/4M pages. It also implements a Second-level TLB (STLB) that supports 512 4-Kbyte pages.

Hardware-triggered prefetch logic is limited to strides of less than 4 Kbytes. Hardware-triggered prefetch also suffers when a DTLB miss occurs. That is, if the address of the data being accessed does not exist in the DTLB, then a hardware-triggered prefetch may fail. Software prefetch does not suffer from this effect. Software prefetch succeeds even when a DTLB miss occurs. In any case, when a DTLB miss occurs, the DTLB must be populated before the required data can be fetched from cache or external memory.

Environment

The scenarios described in this paper were run on a platform that included the following features:

- GreenCity Customer Reference Board
- Intel® Xeon® Processor E5530 @ 2.40 GHz, uniprocessor configuration
- 6 GB (6 x 1GB) DDR3 1333 MHz
- CentOS* 5.4 running vanilla Linux* 2.6.28.9
- GCC version 4.1.2 20080704 (Red Hat* 4.1.2-46)

Calculated Read Address

In most data processing applications, the data to be processed / manipulated is located at an address in memory which is known well in advance of the data being required by the execution unit. One such class of applications is audio / video processing (encoding / decoding / compression / decompression). In such applications, the data is usually presented in the form of an array (or ring buffer) of data, sometimes several megabytes in size. The data in the buffer may be accessed sequentially (using a fixed stride) or more randomly, but always by reference to some offset from the base address of the buffer.

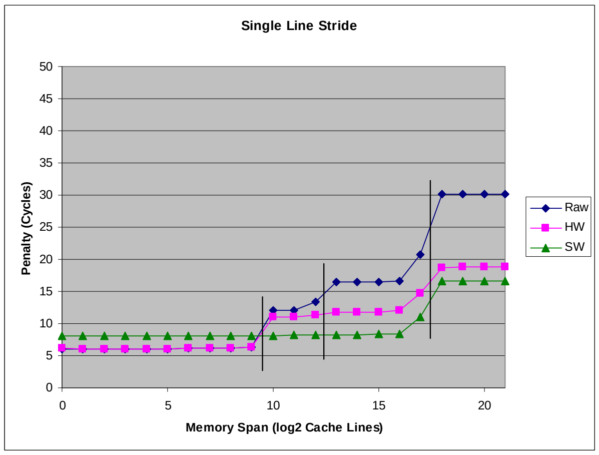
In the code below, a block of data is copied from a buffer at a regularly increasing offset in memory (the *stride*). The blocks are cache-line-aligned, to ensure that they do not straddle cache lines. The size (42 bytes) of the data copied in this example is taken from a real-world application (a Radio Network Controller (RNC), which uses Transport Blocks of 336 bits).

```
#define CACHE_LINE_LENGTH 64


static inline void prefetch(
                const void *ipAddr)
{
  __asm__ __volatile__ ("prefetchnta (%0)" \
                        : /* nothing */ \
                        : "r" (ipAddr));
}


void  Spin(
     char *ipData)
{
  char *lpDst;
  .
  .
  while(...)
  {
    memcpy(lpDst,
           ipData + (lvOffset * CACHE_LINE_LENGTH),
           42);

    prefetch(ipData + (lvOffset + (STRIDE * 8)) * CACHE_LINE_LENGTH);

    lvOffset += STRIDE;
  }
}
```

If the buffer size is small, then the memory location being accessed remains in cache, and so the cost, in terms of clock cycles, of the above code loop is small. As the size of the buffer increases, the buffer can no longer fit into the first-level cache (DL1) and so each loop incurs the penalty associated with accessing the mid-level cache (MLC), and so on. As the size of the buffer increases further, finally exceeding the size of the last-level cache (LLC), every access to the buffer incurs the penalty associated with accessing external memory.

Figure 1 illustrates the cost, in terms of clock cycles, of access to the various levels of the cache hierarchy and external memory (when using base addressed memory access and a stride length of one cache line (64 bytes)). The blue, pink, and green traces indicate the cost of a) the algorithm without prefetching, b) the algorithm with hardware prefetching enabled, and c) the algorithm with software prefetching enabled. In the case of software prefetching, the cache line eight strides ahead is prefetched into the cache hierarchy, as indicated in the previous code fragment.

Figure 1. Cache Clock Cycle Penalty



The four regions of the graph, indicated by vertical lines, represent the four levels of the cache hierarchy, namely data cache levels 1 (DL1), 2 (MLC), 3 (LLC), and external memory. Each region shows the cost, in terms of clock cycles, of access to a specific cache level (DL1, MLC, LLC, or external memory). The boundaries of the DL1, MLC, and LLC are clearly visible in the traces.

From the above, it can be seen that when access is limited to DL1 (the leftmost region), there is no benefit to prefetching, and, in fact, a penalty is incurred when software prefetching is used, as the prefetch instruction has a cost associated with issuance, with no resulting benefit (the data is already in-cache). When accessing external memory, software prefetch has a marked benefit, reducing the loop cost by almost 50%.

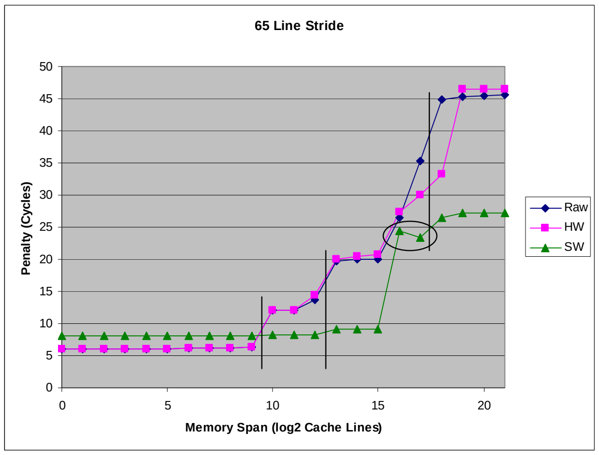
In the processor, under examination, the MLC and LLC are a shared resource, acting as both data and instruction caches. This explains the slight increase in penalty towards the right of both the MLC and LLC regions.

The newest Intel Architecture processors incorporate multiple read buffers, located between the execution units and the cache/memory controllers, which allow multiple read operations to be in flight at any given moment. Thus, the penalty indicated in the above graph is not the absolute penalty for any one read operation, but rather the cost of one of many consecutive iterations of the above code loop.

The Effect of DTLB Miss on Prefetching

While the access stride is such that the hardware prefetch logic is unable to predict accurately which cache lines should be prefetched, software prefetch may still be useful. Figure 2 illustrates the effect of a large access stride on the algorithm described in the previous section. In this case the access stride used in the algorithm is 65 cache lines, ensuring that each access refers to a different page, and thus DTLB entry.

Figure 2. Cache Clock Cycle Penalty (large access stride)



From Figure 2, it is evident that the hardware prefetch logic has little effect on the cost of loop iteration in this scenario. Software prefetch has a marked benefit, reducing the loop cost by almost 50% when accessing external memory. Again, when access is limited to DL1 (the leftmost region), there is no benefit to prefetching, and in fact, a penalty is incurred when software prefetching is used, as the prefetch instruction has a cost associated with issuing, with no benefit resulting (the data is already in cache).

In addition, in this scenario, the penalty associated with LLC miss is higher than in the previous example. This is as a result of incurring a DTLB miss penalty on every access to external memory (stride is 65 cache lines, thus ensuring a DTLB miss on every access).

In the above figure, the STLB boundary is evident, at 16/17 (circled). Between 2M and 8M, although the data is in cache, a TLB miss occurs for each data access, and so the loop incurs the associated penalty. In that case there is a minimum penalty associated with the population of the DTLB entry, even though the data at the required address is already in the LLC.

Pointer Chasing

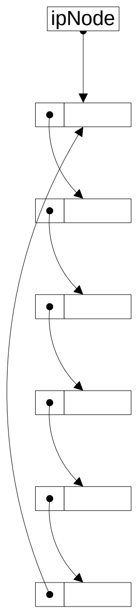
In many applications, the address (location) of the next datum to be accessed by the execution unit is dependent on a datum which has been previously read (for example, a pointer or index in one structure refers to the next structure to be accessed by the execution unit). In its simplest form this is referred to as *pointer chasing*.

One obvious example of pointer chasing is the use of the sk_buff structure, a type of linked list, in the Linux kernel. Any application which requires the use of extensible data storage typically uses linked lists. A less obvious example of pointer chasing is IP address hashing. When a packet arrives at a network interface, its IP address is typically passed through a hash algorithm and the resulting number is then used as an index into a large hash table. Thus, the content of the received packet (the IP address) determines which portion of the hash table needs to be fetched into cache.

A typical example of pointer chasing is a linked list of data structures. The following code fragment typifies a pointer chasing loop, and is illustrated in the diagram on the right below.

Figure 3. Pointer Chasing Loop

```
typedef struct _Node
{
      struct _Node *pNext;
     unsigned char  mPayload[];

} Node_t;


void  Spin(
   Node_t *ipNode)
{
  unsigned char *lpDst;
  .
  .
  while(...)
  {
    Node_t *lpTemp = ipNode;

    ipNode = ipNode->pNext;
    prefetch(ipNode);

    memcpy(lpDst,
           lpTemp->mPayload,
           42);
  }
}
```



In pointer chasing algorithms, it is not possible for the execution unit to issue a read request to the memory location of the next node in the chain, until the current read request has been completed. This dependency can lead to instruction pipeline stalls. The more randomly the objects in the chain are distributed in memory (that is, the structures are not in any discernable order or spacing in memory), the less effective prefetch techniques (either hardware or software) can be. In a completely disordered chain, prefetching can have little beneficial effect on the access penalty to the structures.

Figure 4. Pointer Chasing

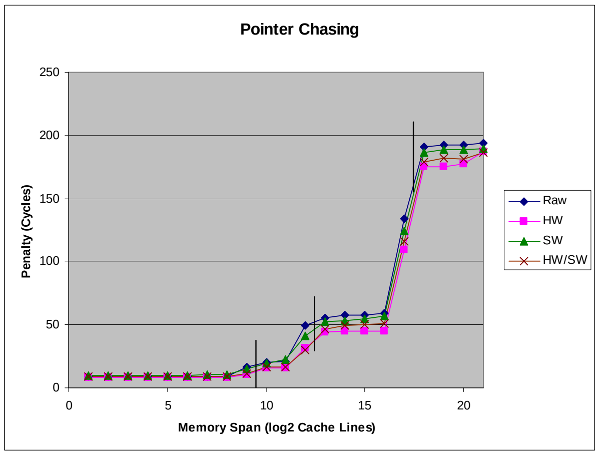


Figure 4 illustrates the effect (or non-effect) of prefetching on a completely disordered pointer chasing algorithm (linked list). As in the previous graphs, the four regions of the cache hierarchy are indicated by vertical lines. Each region shows the penalty, in terms of clock cycles, of access to a specific level of the cache hierarchy (DL1, MLC, LLC, and external memory).

Parallel Chains

As described in the preceding section, pointer chasing algorithms can (and generally do) cause stalls in the instruction pipeline. It is possible to make use of the multiple read buffers, described previously, which are available in recent Intel® Architecture processors, to reduce the penalty associated with fetches to external memory, even when using pointer chasing algorithms. By parallelizing the chains, it is possible to traverse multiple chains simultaneously, with minimal additional overhead, in terms of clock cycles, per loop iteration.

The following code fragment is a modified version of that shown in Figure 3. The code in red has been added in this example. In brief, this algorithm chases multiple pointer chains, in parallel, as indicated by the righthand diagram in Figure 5.

Figure 5. Pointer Chasing Loop – Multiple Chains

```
typedef struct _Node
{
      struct _Node *pNext;
     unsigned char  mPayload[];

} Node_t;


void  Spin(
   Node_t *ipNode[])
{
  unsigned char *lpDst;
  .
  .
  while(...)
  {
    unsigned  lvIndex;


    for(lvIndex = 0;
        lvIndex < WIDTH;
        lvIndex++)
    {
      Node_t *lpTemp = ipNode[lvIndex];

      ipNode[lvIndex] = ipNode[lvIndex]->pNext;
      prefetch(ipNode[lvIndex]);

      memcpy(lpDst,
             lpTemp->mPayload,
             42);
    }
  }
}
```

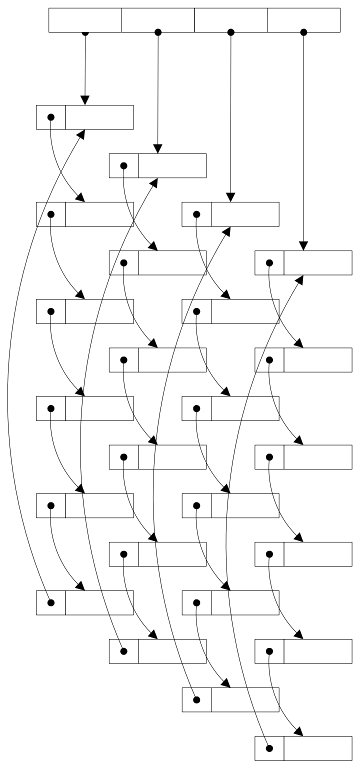
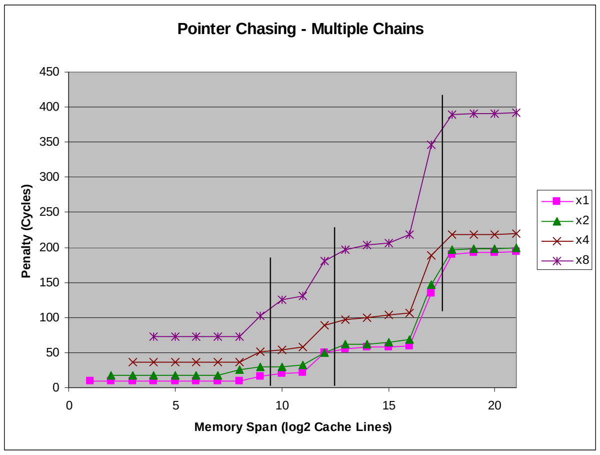


Figure 6 illustrates the cost, in terms of clock cycles, of traversal of multiple (1, 2, 4, and 8) pointer chains. In this example, no prefetch mechanisms are used.

Figure 6. Pointer Chasing – Multiple Chains



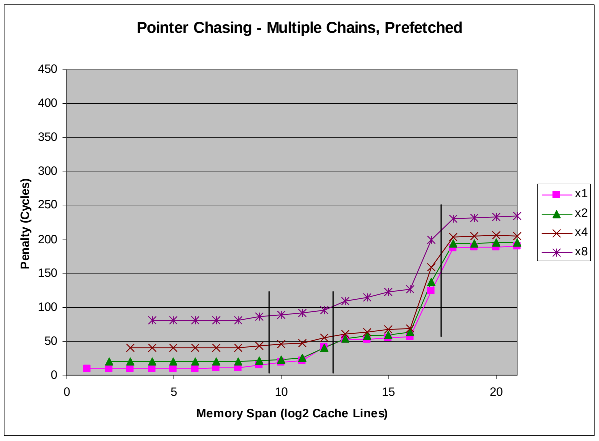
In this example, when the chains are restricted to cache (that is, the combined size of the chains is less than the size of the cache), doubling the number of chains traversed in parallel results in a doubling of the number of cycles it takes to traverse each loop. This is to be expected, as instruction pipeline stalls due to cache miss do not occur when access is restricted to DL1 cache.

As the length of the chains increase (moving to the right in the figure), the number of cycles per loop iteration increases. When the combined length of the chains exceeds the size of the last level cache (LLC), the doubling effect does not follow. The cost of chasing one, two, and four chains in parallel is almost the same (with minimal overhead). The cost of chasing eight chains in parallel is almost double that of chasing four chains.

Prefetching and Parallelizing

As described in Pointer Chasing, the pointer chains in these examples are highly randomized, so the use of hardware prefetching has no reducing effect on the penalty incurred as a result of a cache miss. The use of software prefetch, however, has a marked benefit in the case of chasing eight chains in parallel, and also has some benefit in the case of four parallel chains.

Figure 7. Pointer Chasing – Multiple Chains, Prefetched

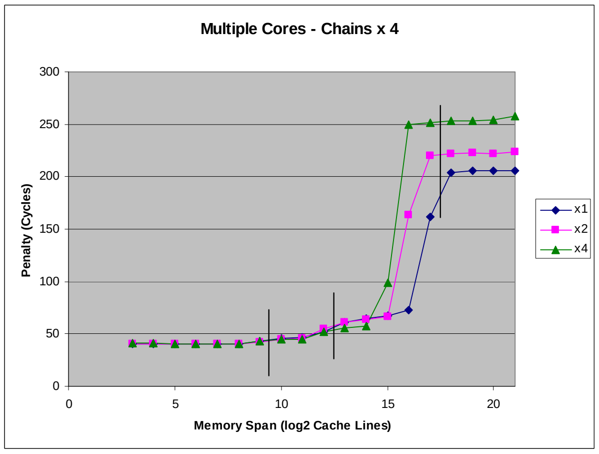


In this example, software prefetching is used with parallelized chain chasing. The results can be seen in Figure 7.

Multiple Logical / Physical Cores

When the workload described in the previous section is replicated on multiple cores of the processor, the cores do not interfere greatly with each other. The following figure illustrates the cost, in terms of clock cycles, of running four chains in parallel, on each of one, two, and four cores. Running the algorithms on multiple cores simultaneously causes the appearance of the memory access penalty to move left in the graph (that is, the apparent size of the LLC available to each core decreases). In addition, as the last level cache is shared, there is some interference (sharing of other resources) between the processes running on the four cores, which causes the duration of each loop to increase, but not dramatically.

Figure 8. Multiple Cores – Chains x 4



Conclusion

The investigative work described in this paper highlights a potential order of magnitude throughput improvement in certain data processing workloads. Multiple pointer chains can be traversed in parallel (in effect, simultaneously) at minimal additional cost, in terms of clock cycles, when accessing external memory.

In order to make full use of the hardware acceleration technology, specifically multiple fill and load buffers, available in the latest generation of Intel® Architecture processors, it will be necessary to modify the software architectures of certain applications to allow workloads to operate in parallel on each processor core. This will allow more of the available memory interface bandwidth to be usable by the software application running on the processor cores.

§

Author

Brian Forde is a Senior Platform Applications Engineer with the Intel Architecture Group at Intel Corporation.

John Browne is a Platform Solutions Architect with the Intel Architecture Group at Intel Corporation.

Terminology

DL1	First Level Data Cache
DTLB	Data Translation Lookaside Buffer
GCC	GNU Compiler Collection
LLC	Last Level Cache
MLC	Middle Level Cache
RNC	Radio Network Controller
STLB	Second-level Translation Lookaside Buffer