Robust Method to Determine Cache and TLB Characteristics

A Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Varadharajan Chandran, B.Tech

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2011

Master’s Examination Committee:

Dr. P. Sadayappan, Advisor

Dr. Atanas Rountev
Abstract

Modern compilers of self-optimizing computing systems require the values of hardware parameters such as the capacity of cache and number of entries in translation lookaside buffer (TLB)[3]. Additional information such as the cache block size and page size along with the Hit Time and Miss Penalty of these hardware entities are needed to apply static and dynamic compiler optimization techniques.

 Currently, no tool or approach is robust enough to provide accurate information regarding the underlying hardware. They fail when encountering multiple levels of cache and TLB present in current generation of processors.

 In this Thesis, we describe a staggered and robust approach to first detecting the hardware entities and their sizes and then disambiguating between the entities by deducing their block sizes. We describe the novel algorithms for measurement of the hardware parameters and the mathematical intuition behind the disambiguation. Experimental evaluations of this technique on traditional workstations, laptops and servers show that our approach produces more accurate and complete results than existing tools.
I dedicate my work to my parents Mr. V.R. Chandran and Mrs. Shyamala Chandran and
grandfather Mr. V. Ramakrishnan for being a tremendous source of inspiration and a
guide to life
Acknowledgments

I would like to thank my advisor, Dr. P. Sadayappan, for his support, guidance and encouragement for my Master’s Thesis.

I would like to thank Dr. Atanas Rountev for serving on my Master’s examination committee.

I would especially like to thank Mr. Mahesh Ravishankar for collaborating with me on this project.

I would also like to thank my lab colleagues Tom Henretty, Justin Holewinski, Naser Sedaghati for the technical inputs that helped me in my Master’s program.

I would like to thank Dr. Radu Teodorescu for his technical inputs and letting me present my work before the Architecture group.

I am grateful to my Advisor and Department of Computer Science and Engineering for supporting me financially during my Master’s program.

Finally, I would like to thank my parents for motivating me to reach the stars.
Vita

2009 ........................................ B.Tech. Information Technology,
   Anna University
   Chennai, Tamil Nadu, India.

2009-2011 ................................. Graduate Student,
   Department of Computer Science and En-
   gineering,
   The Ohio State University.

Fields of Study

Major Field: Computer Science and Engineering

Studies in:
   High Performance Computing  Dr. P. Sadayappan
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Objective</td>
<td>2</td>
</tr>
<tr>
<td>2. Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Memory Hierarchy</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Previous Approaches</td>
<td>4</td>
</tr>
<tr>
<td>3. Methodology</td>
<td>6</td>
</tr>
<tr>
<td>3.1 Calibrator framework</td>
<td>6</td>
</tr>
<tr>
<td>3.2 Naive approach</td>
<td>7</td>
</tr>
<tr>
<td>3.3 Robust Approach</td>
<td>8</td>
</tr>
<tr>
<td>3.3.1 Detection</td>
<td>8</td>
</tr>
<tr>
<td>3.3.2 Algorithm for Detection</td>
<td>11</td>
</tr>
<tr>
<td>3.3.3 Disambiguation</td>
<td>12</td>
</tr>
<tr>
<td>3.3.4 Algorithm for Disambiguation</td>
<td>19</td>
</tr>
<tr>
<td>3.4 Similarity between TLB and Cache</td>
<td>21</td>
</tr>
<tr>
<td>3.5 Determining the Miss Penalties</td>
<td>23</td>
</tr>
</tbody>
</table>

vi
4. Results ................................................................. 28
    4.1 Intel Core i7 - Nehalem architecture ....................... 28
    4.2 Intel Core 2 Quad ............................................ 30
    4.3 Intel Core2 Duo P8700 ....................................... 31
    4.4 Intel Xeon 3070 .............................................. 32
    4.5 AMD Phenom ................................................... 33

5. Current Limitations and Future Work ............................ 34

Bibliography ................................................................ 36
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Jump from 128 to 160 spots at stride 2048 in Intel Core i7</td>
</tr>
<tr>
<td>3.2</td>
<td>Normal stride of 1024 without lower stride added</td>
</tr>
<tr>
<td>3.3</td>
<td>Pattern 1: Stride of 2048 with added lower stride 1024</td>
</tr>
<tr>
<td>3.4</td>
<td>Jump from 86 to 140 spots at stride 2048 + 1024 in Intel Core i7</td>
</tr>
<tr>
<td>3.5</td>
<td>Jump from 32 to 40 spots at stride 8192 in Intel Core i7</td>
</tr>
<tr>
<td>3.6</td>
<td>Normal stride of 8192 without lower stride added</td>
</tr>
<tr>
<td>3.7</td>
<td>Jump from 64 to 80 spots at stride 8192 + 4096 in Intel Core i7</td>
</tr>
<tr>
<td>3.8</td>
<td>Pattern 2: 8192 stride with added lower stride 4096</td>
</tr>
<tr>
<td>4.1</td>
<td>L1 Data cache comparison with other approaches for Intel Core i7 920 @ 2.67Ghz</td>
</tr>
<tr>
<td>4.2</td>
<td>L2 Data cache comparison with other approaches for Intel Core i7 920 @ 2.67Ghz</td>
</tr>
<tr>
<td>4.3</td>
<td>L1 TLB comparison with other approaches for Intel Core i7 920 @ 2.67Ghz</td>
</tr>
<tr>
<td>4.4</td>
<td>L2 TLB comparison with other approaches for Intel Core i7 920 @ 2.67Ghz</td>
</tr>
<tr>
<td>4.5</td>
<td>L1 Data cache comparison with other approaches for Intel Core2 Quad - Q6670</td>
</tr>
<tr>
<td>4.6</td>
<td>L1 TLB comparison with other approaches for Intel Core2 Quad - Q6670</td>
</tr>
</tbody>
</table>
4.7 L2 TLB comparison with other approaches for Intel Core2 Quad - Q6670 . 30
4.8 L1 Data cache comparison with other approaches for Intel Core2 Duo - P8700 31
4.9 L1 TLB comparison with other approaches for Intel Core2 Duo - P8700 . 31
4.10 L2 TLB comparison with other approaches for Intel Core2 Duo - P8700 . 31
4.11 L1 Data cache comparison with other approaches for Intel Xeon - 3070 . 32
4.12 L1 TLB comparison with other approaches for Intel Xeon - 3070 . . . . . 32
4.13 L2 TLB comparison with other approaches for Intel Xeon - 3070 . . . . . 32
4.14 L1 Data cache comparison with other approaches for AMD Phenom 9850 . 33
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Spots vs Stride graph showing the movement of jumps. Measurements given in $10^{-11}$ seconds</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Change in pattern from Decrease to Increase</td>
<td>19</td>
</tr>
<tr>
<td>3.3 Step Function of Miss Time for TLB</td>
<td>26</td>
</tr>
<tr>
<td>3.4 Step Function of Miss Time for Cache</td>
<td>27</td>
</tr>
</tbody>
</table>
Chapter 1: INTRODUCTION

Knowledge of data cache memory and data TLB characteristics is becoming increasingly important in compiler optimizations. Cache-conscious algorithmic design is the basis of tuned numerical libraries, such as the BLAS and LAPACK and has been shown to improve full application performance significantly [13, 7]. Compilers and automatically tuned software systems such as ATLAS [11] and PHiPAC [6] need accurate information about Cache and TLB characteristics to generate optimal code. To reduce the running time in scientific codes, its memory access patterns can be optimized by transformations such as loop tiling and data reorganization [3]. These systems would benefit from more accurate cache characterization for new architectures.

Published information about detailed cache and TLB characteristics can be difficult to find or may be inaccurate or out-of-date. On some machines, it may be possible to determine some of this information by reading special registers or records in the processor or operating system [4]. However, most processors and operating systems do not support such mechanisms or provide very limited support. It will often be necessary to determine or verify this information empirically. Existing tools such as lmbench [10], Calibrator [9] and X-ray [8] measure some of these memory hierarchy parameters, but our experiments show that none of them offer the robustness or accuracy of our micro-benchmarks.
This thesis describes a machine independent methodology of timing microbenchmark codes. This timing data can be fed into our analytical model which gives us accurate numbers for cache and TLB Characteristics. We describe how our methodology builds on and extends previous work based on timings of microbenchmarks, with the introduction of a robust analytical model that detects distinct hardware entities (both cache and TLB) and disambiguates between them to resolve the interference that arises between Cache and TLB measurements. Our approach gives accurate results for the Intel Core 2 Duo, Intel Core 2 Quad, Intel Core i7 and Intel Xeon and we also discuss ideas for future work and applications of the results.

1.1 Objective

Our objective is the measurement of the following physical characteristics of the underlying hardware. These parameters are obtained for every level of cache and TLB.

- Capacity
- Sets
- Associativity
- Block Size
- Hit Time
- Miss Penalty
Chapter 2: BACKGROUND

2.1 Memory Hierarchy

A CPU cache is a cache used by the central processing unit of a computer to reduce the average time to access main memory. The cache is a smaller, faster memory which stores copies of the data from the most frequently used main memory locations. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory. When the processor needs to read from or write to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. The time to access a memory location that is already present in cache is Hit Time. The time taken to access a memory location not present in the cache is Miss Penalty. Most modern desktop and server CPUs have at least three independent caches: an instruction cache to speed up executable instruction fetch, a data cache to speed up data fetch and store, and a translation lookaside buffer (TLB) used to speed up virtual-to-physical address translation for both executable instructions and data. Data cache is usually organized as a hierarchy of more cache levels (L1, L2, etc). High performance computing tasks are data intensive and compiler optimizations are targeted towards making code fit into the data cache.
2.2 Previous Approaches

Previous work measured the time to execute simple memory-bound loops on arrays of different sizes and with different sizes and estimated the cache and TLB characteristics from the results [12, 14]. The data cache and TLB characteristics can be calculated once the number of memory references and the number of cache and TLB misses are known for the different array sizes and strides. Previous work inferred from the number of misses from the timing results. In [12], an analytical model is developed for a single cache that identifies four cases that show up as four distinct regions on the timing curves. The model assumes a single blocking cache with an LRU replacement policy and it can determine cache size, cache line size and cache associativity from the timing results. The same model is used for TLB characterization and the values for the array size and stride at which the TLB phenomena occur are different enough from those for the cache phenomena that the effects can be isolated from one another.

In [14], the analytical model from [12] is first extended from four to six cases to include transitional portions of the timing curves. The model is further extended to a system with two caches. The two-cache model assumes that the second-level cache includes the first-level, that both caches have the same block size, that associativity is non-decreasing, that there is not prefetching and that the replacement policy for both caches is LRU. Six cases are identified and some fairly complicated timing formulas are given that allow the cache characteristics of the two caches to be calculated from timing results for the microbenchmarks. Next the model is extended to a two-cache system with a TLB. Some assumptions on the TLB operation are made, seven cases are identified and another set of formulas are given.
The Calibrator microbenchmark described in [9] varies stride and array size in a small loop that executes a million memory reads in order to determine cache characteristics including cache size, cache line size, TLB size, page size and miss latencies.

The above approaches fail because the timing results are fairly complex. The microbenchmark considers all levels of the memory hierarchy simultaneously. Therefore, these results are usually interpreted manually to obtain the memory hierarchy parameters. Although tools like Calibrator, Imbench and X-ray can determine some cache parameters automatically from these timing results, none of them measure TLB associativity, for example. Moreover, optimizations performed by modern compilers when compiling the C code can confuse the timing measurements. Also, the optimizations performed by modern processors such as pre-fetching of reverse strided accesses. The major drawback of approaches such as X-ray and calibrator is the inability to distinguish between interfering entities. Multilevel Cache and TLB organization of modern day processors return timing results that are too complex and interfering that these tools failed to produce accurate results. Other tools use hardware performance counters to probe the memory hierarchy [5], but these tools have portability problems.

The key difference between our approach and previous approaches is that our analytical model is designed to account for interference between entities and we have established an algorithm to distinguish and disambiguate between them.
Chapter 3: METHODOLOGY

3.1 Calibrator framework

The idea underlying the calibrator tool is to have a micro-benchmark whose performance only depends on the frequency of cache misses that occur. The calibrator is a simple C program, mainly a small loop that executes a million memory reads. By changing the stride and the size of the memory area, we force varying cache miss rates.

In principle, the occurrence of cache misses is determined by the array size. Array sizes that fit into the L1 cache do not generate any cache misses once the data is loaded into the cache. Analogously, arrays that exceed the L1 cache size but still fit into L2, will cause L1 misses but no L2 misses. Finally, arrays larger than the last level of cache causes misses in all levels of cache.

The framework implements features such as the reverse-point chain that aims to defeat processor prefetching. The pointer chain essentially involves each memory location holding the address of the next access and the addresses are stored in reverse stride. This makes address speculation difficult and each location needs to be accessed to get the address to the next location and since this is done in reverse, it is found that this is enough to defeat the prefetching algorithms of most modern processors. We improve on this framework by setting up a completely randomized pointer chain that does not follow any specific stride and hence cannot be prefetched.
In order to avoid loop overhead and also minimize the instruction cache usage, the pointer chain and the timing mechanisms are implemented as simple code and the traversal of the pointer chain is done a few thousand times before loop is involved. In order to smoothen the timing variations and also to prevent the effects of cold cache creeping into timing measurements, these reads are performed more than a million times.

3.2 Naive approach

We are interested in characterizing the following properties of TLB/Cache:

- The physical characteristics: Capacity (C), Associativity (A) and Block Size (B)
- Hit time
- Miss penalty

The naive way to try and determine the size of the TLB-L1 would be to decide on a fixed stride and increase the number of spots accessed by powers of 2. The point at which we see the jump should help us determine the TLB size. This is quantified by the equation (3.1)

\[
S = \begin{cases} 
2^e \times A \times 2^p / H & \text{if } H < 2^{e+p} \\
A & \text{if } H \geq 2^{e+p}
\end{cases}
\]

(3.1)

where, \( S \) is the value we are looking for, \( H \) is the stride being used, \( E = 2^e \) is the number of sets in the TLB, \( A \) is its associativity and \( P = 2^p \) is the page size. Therefore, if we double the stride, then the number of spots at which the jump occurs reduces by half. In the limit, for very large values of \( H \), the spots being accessed would be in pages which map to the same set in the TLB. At this limit, accessing more than \( A \) spots would generate a miss and result in a jump. This effect can be used to get the associativity of the TLB, but
without knowing the page size of the system, we would not be able to calculate the number of sets, hence the size of the TLB.

### 3.3 Robust Approach

The approach that we have taken is to first characterize the physical characteristics of the TLB and Cache. Having achieved that we would be in a position to generate misses in a way we can ensure that the miss is happening due to the entity that we are interested in. This way we will be able to remove interference between different entities.

We now highlight the approach that we use to determine the physical characteristics of the different hardware entities. We will use the example of trying to find the TLB-L1 size on Intel Core-I7 Nehalem architecture, to highlight the salient features of our approach.

One of the major drawbacks of previous microbenchmark approaches is the issue of conflict and interference. Multilevel cache and TLB wreaks havoc on timing measurements. This issue is addressed by breaking down our approach into two stages.

- Entity Detection (Size and Associativity)
- Entity Disambiguation (Block Size)

#### 3.3.1 Detection

While previous approaches attempt to look at timing charts to find jumps in timing measurements and plot them as entities, our approach begins by ironing out these interferences. We begin with a small stride in the order of 16 bytes and progressively increase this stride to large values. When the stride $H$ is doubled to $2H$, it is observed that the spots $S$ at which jumps occurred is halved to $S/2$. While observing the timing chart in figure 3.1 with spots and stride, the movement in jump from $S$ to $S/2$ is observed for every doubling
of the stride till we hit upon the associativity of the entity causing that jump. We can say that these jumps *gravitate* towards the associativity of the entity. We continue to double the stride till these jumps stop moving and come to a rest at the associativity of each entity. This is due to the fact that the large stride means we access only Set 0 from each entity and the number of spots after which we miss in each entity is essentially equal to the associativity at this point. In figure 3.1, there is movement till the stride is doubled until 2M and 4M. At these high strides, jumps are observed at 4 spots and 8 spots which indicates that there are multiple entities that have the associativity 4 and 8. Also, by removing movement from the chart, we have got rid of interference at this stage.

\[ \text{Capacity}(C) = \text{Associativity}(A) \times \text{Stride of Entity} \]

(3.2)

After achieving the *large stride* at which no movement of jumps occur, we gather the size of each entity by observing the *Stride of Entity* i.e. the stride at which we hit the associativity of the specific entity. In the figure, the movement at stride 512K to 256K from 4 spots to 8 spots shows that the entity was 4-way associative and the stride of the entity is 512K. Applying (3.2), we get the size of this entity to be \(512K \times 4 = 2MB\).

Considering the movement, we are also able to identify the miss penalty of the entity by measuring the magnitude of the jump. In this specific case from 512K to 256K from 4 spots to 8 spots, the timing changes from 9.36ns to 5.99ns which gives a difference of 3.37 nanoseconds.

At this point, we are not aware of the block size of the entity and hence we cannot differentiate between cache and TLB.
Figure 3.1: Spots vs Stride graph showing the movement of jumps. Measurements given in $10^{-11}$ seconds
3.3.2 Algorithm for Detection

**Algorithm 1: Detection of Entity Capacity and Associativity**

**Data:** H,S,CurrentTime,PrevTime,Jump

**Result:** Capacity,Associativity

**Input:** Max Memory(Z), Max Associativity(N) and Max Stride(M)

**Output:** Stride of the entity and associativity

Initialization $H = 16$, $S = 1$;

while $H 	imes N$ is less than $Z$ do

while $S$ is less than $N$ do

$CurrentTime \leftarrow \text{Time}(H,S)$;

$Jump \leftarrow \text{DeltaDiff}(CurrentTime, PrevTime)$;

if $Jump$ then

$RecordJump(H,S)$;

end

$S \leftarrow S + 1$;

$PrevTime = CurrentTime$ ;

end

if isMovement($H, H/2$) then

$H \leftarrow H \times 2$;

continue;

end

else

break;

end

DetectEntity($H$)

- The algorithm takes in the upperbound values for memory, stride and associativity and returns the size, associativity and miss penalty of entities.

- $H$ is initialized to the lowest value of line size that is realistic for existing processor architectures.
• $S$ goes from 1 to max associativity

• The `time()` subroutine returns the timing measurement for a million reads for a specified higher stride and number of spots

• The `DeltaDiff()` subroutine returns true if there is a jump(or a difference in timing measurements) between the current time and previous time

• The `RecordJump()` subroutine records the position of the jump in stride and spots

• The `isMovement()` subroutine returns true if there is a movement in the position of the jumps for a current stride when compared to a previous stride

• The `DetectEntity` subroutine starts decreasing the stride from specified value of $H$ and records the stride at which there is movement in jumps for each entity. This gives each entity and its stride of entity.

### 3.3.3 Disambiguation

The issue of conflict is half addressed when most of the interference is ironed out and entity size and associativity are found. It is important to find the block size of these entities to disambiguate between cache and TLB and also to find the number of sets. In some cases, Cache and TLB might have the same size and can cause jumps in measurements at the same number of spots.

\[
\text{Stride}(S) = \text{HighStride}(H) + \text{LowStride}(L) \tag{3.3}
\]

Our strategy to disambiguate involves using the sum of two strides. During detection, only a single value of stride which is a power of 2 is used. During disambiguation, the
stride is a sum of a higher stride and a lower stride, both a power of 2, as given in the equation (3.3)

There are two clear patterns that emerged from using the sum of a higher stride and lower stride. These patterns are explained below.

**Pattern I :** \( H \leq 2^p \)

When the value of the higher stride is lesser than the page size (which we don’t know), the access pattern would be such that we would end up filling the TLB. For example, on Intel Core i7, the TLB-L1 size is 64 entries and it is 4-way associative. The page size is 4096Kb. With a stride of 2048, two accesses would map to the same page, but a third access would map to a new page. This is shown in the table 3.2, where the shaded region, represents the page number in binary representation. Therefore, we would expect the jump to happen at 128 spots, as given by Eq (3.1). The actual timing values are shown in table 3.1 which indicates a jump of \( 12.83ns - 9.77ns = 3.06ns \) at 128 spots as expected.

Now consider a lower stride of 1024. Looking at the actual memory addresses accessed and comparing it to the original, it can be seen that once in every two spots, we would “affect” the higher stride and therefore change the pattern in which we would access the

<table>
<thead>
<tr>
<th>Spots</th>
<th>AMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>977</td>
</tr>
<tr>
<td>160</td>
<td>1283</td>
</tr>
</tbody>
</table>

Table 3.1: Jump from 128 to 160 spots at stride 2048 in Intel Core i7
Table 3.2: Normal stride of 1024 without lower stride added

different sets in TLB. What would have been 3 accesses earlier, would now be just 2 accesses.

For example, with a stride of just 2048, we would access addresses say, 0, 2048, 4096, 6144 . . .
. With a modified stride of 2048 + 1024, we would access addresses, 0, 3072, 6144 as shown in table 3.3. The highlighted region indicates the page number in bits and it is seen that the accesses to pages is not uniform and some of the sets(pages) are skipped due to the effect of the lower stride. Therefore the number of spots at which the jump occurs, \( A \), would reduce. We can quantify it with the following set of equations:

\[
N = \frac{H}{L}; \quad A = S \times \frac{N}{N+1} \tag{3.4}
\]

where, \( S \) is given from Eq (3.1). For example, with \( H = 2048 \) and \( L = 1024 \); from Eq (3.1) we get \( S = 128 \) and the number of spots at which the jump occurs would be \( A \approx 86 \). This jump is shown in table 3.4. The key point is that we see a reduction in the number of


<table>
<thead>
<tr>
<th>Stride (in decimal)</th>
<th>Stride (Bit-pattern)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>3072</td>
<td>0000110000000000</td>
</tr>
<tr>
<td>6144</td>
<td>0001100000000000</td>
</tr>
<tr>
<td>9216</td>
<td>0010010000000000</td>
</tr>
<tr>
<td>12288</td>
<td>0011000000000000</td>
</tr>
<tr>
<td>15360</td>
<td>0011110000000000</td>
</tr>
<tr>
<td>18432</td>
<td>0100100000000000</td>
</tr>
<tr>
<td>21504</td>
<td>0101010000000000</td>
</tr>
<tr>
<td>24576</td>
<td>0110000000000000</td>
</tr>
<tr>
<td>27648</td>
<td>0110110000000000</td>
</tr>
</tbody>
</table>

Table 3.3: Pattern 1: Stride of 2048 with added lower stride 1024

<table>
<thead>
<tr>
<th>Spots</th>
<th>AMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>86</td>
<td>305</td>
</tr>
<tr>
<td>140</td>
<td>602</td>
</tr>
</tbody>
</table>

Table 3.4: Jump from 86 to 140 spots at stride 2048 + 1024 in Intel Core i7

spots at which the jump occurs. As another example, if the lower stride was 512, the value of $A$ would be $128 \times 4/5 \approx 102$.

**Pattern II**: $H > 2^p$

When the higher stride being used is more than the page size, the accesses pattern would be such that, it would not fill up the TLB. i.e. there would be sets in the TLB, which are not accessed at all. For example, on Intel Core i7, with a stride of 8192, only 8 of the 16 sets would be accessed, i.e., the spots accessed would be say, 0, 8192, 16284, 32468, ... 

The pages accessed would be 0, 2, 4, 6 ... This is shown in figure 3.6, the shaded region...
Table 3.5: Jump from 32 to 40 spots at stride 8192 in Intel Core i7

<table>
<thead>
<tr>
<th>Stride(in decimal)</th>
<th>Stride(Bit-pattern)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>8192</td>
<td>0010000000000000</td>
</tr>
<tr>
<td>16384</td>
<td>0100000000000000</td>
</tr>
<tr>
<td>24576</td>
<td>0110000000000000</td>
</tr>
<tr>
<td>32768</td>
<td>1000000000000000</td>
</tr>
<tr>
<td>40960</td>
<td>1010000000000000</td>
</tr>
<tr>
<td>49152</td>
<td>1100000000000000</td>
</tr>
<tr>
<td>57344</td>
<td>1110000000000000</td>
</tr>
<tr>
<td>65536</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>73728</td>
<td>0010000000000000</td>
</tr>
</tbody>
</table>

Table 3.6: Normal stride of 8192 without lower stride added

representing the page number in binary representation. Therefore the jump would be seen at 32 spots (since it is 4-way associative). The actual values from table 3.5 show that this expectation is true and there is a jump at 32 spots of magnitude 1.98ns. On introducing a lower stride, when sufficient number of spots are accessed, the lower stride will change the access pattern, to access pages which map to the unaccessed sets before. For example, for a stride of 8192 + 4096, the access pattern would be 0, 12286, 24572, 36858... which would map to pages 0, 3, 6, 9... From figure 3.8, it can be seen that all the previously missed sets (the odd numbered sets) will be now accessed, thus increasing the number of spots at which the jump occurs to 64 in this example.

Some key points to note are:

16
Table 3.7: Jump from 64 to 80 spots at stride 8192 + 4096 in Intel Core i7

<table>
<thead>
<tr>
<th>Stride(in decimal)</th>
<th>Stride(Bit-pattern)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>12288</td>
<td>0011000000000000</td>
</tr>
<tr>
<td>24576</td>
<td>0110000000000000</td>
</tr>
<tr>
<td>36864</td>
<td>1001000000000000</td>
</tr>
<tr>
<td>49152</td>
<td>1100000000000000</td>
</tr>
<tr>
<td>61440</td>
<td>1110000000000000</td>
</tr>
<tr>
<td>73728</td>
<td>0010000000000000</td>
</tr>
<tr>
<td>86016</td>
<td>0101000000000000</td>
</tr>
<tr>
<td>98304</td>
<td>1000000000000000</td>
</tr>
<tr>
<td>110592</td>
<td>1011000000000000</td>
</tr>
</tbody>
</table>

Table 3.8: Pattern 2: 8192 stride with added lower stride 4096

- For the previous example, if we decreased the lower stride to 2048, the jump would still be seen at the 64 spots itself, since we would still be filling up the TLB. There is a subtle point to keep track of here. If the lower stride is too small, we would not change the accesses pattern with respect to pages at all. This is quantified later on.

- If instead we had a higher stride of 16384 and had a lower stride of 8192, we would now access only half the TLB size. So the number of spots at which the jump occurs would change from 16 (for the case of just 16384) to 32 (for the case of 16384+8192). In effect, the number of spots needed would keep doubling and stagnate at the TLB size and this stagnation would happen when the lower stride is same as the page size. At this lower stride we would access all the sets in the TLB. This could be used
as a mechanism to deduce the page size, but interference from other entities might
change the point at which we see the stagnation. We can only use this to increase the
confidence in the final deduction.

Coming back to a point we were making earlier, if the lower stride was too small, say
64, when the higher stride is 8192; it would take (Page-size /64) spots for the lower stride
to affect the pattern of pages being accessed. Therefore for such strides, there would be no
increase in the number of spots before the jump. On accessing 32 spots, we would still see
a jump in spite of having a lower stride. To quantify this:

\[ n = \frac{2^p}{L} \]  

(3.5)

where, \( n \) is the minimum number of spots that need to be accessed to see a change in
the number of spots. Therefore if the configuration of page sizes and TLB happened to be
such that, the number of spots at which the higher stride starts repeating accesses to sets
( hence conflict misses ) is lesser than the number of spots needed for the lower stride to
“affect” the page size, there would be no increase in the number of spots.

A summary of the above strategy would be, that for the case where the higher stride is
lesser than or equal to the page size, introducing a lower stride would reduce the number
of spots to see a jump. If on the other hand the higher stride is greater than the page size,
introducing a lower stride ( and accessing a sufficient number of spots ) would increase the
number of spots. This switch in behaviour as shown in figure 3.2, gives us the page size.
In the figure, the downward facing arrows indicates decrease in spots(Pattern 1) and the
upward facing arrow indicates increase in spots (Pattern 2). The stride at which the pattern
switches is indicated as the block size of the entity. The number of sets in the TLB can also
be determined from Equation (3.1). Thus we can fully characterize the TLB.

18
3.3.4 Algorithm for Disambiguation

**Algorithm 2**: Detection of Block Size

**Data**: H, L, S, Trend[]

**Result**: Page Size or Line Size

for each Entity detected do

  initialization H = 16, L = 1, S=1;

  while $H \times S$ is within Maximum Memory Range do

    Observe for Jump with added Lower Stride L;

    if Jump Encountered then

      if Decrease $(H, S)$ then

        Temp ← 'S';

      else if Increase $(H, S)$ then

        Temp ← 'D';

      else if FullyAssociative $(H, S)$ then

        Temp ← 'F';

      else

        Temp ← 'Z';

      end

      Trend[$\log_2 H$] ← Temp;

    end

    S ← S * 2;

  end

end

Page Size ← DetectSize (Trend[]);
• The algorithm takes Higher Stride (H), Lower Stride (L), No. of Spots (S) and uses an array Trend[] to calculate the Page Size (P) or Line Size (L) as the result.

• H is initialized to the lowest value of line size that is realistic for existing processor architectures.

• L is initialized to any small value lesser than the smallest line size value. This serves to map the spots into different sets in cache and TLB and maximize the number of spots accessed. This is done to minimize the interference due to misses due to different entities occurring close to each other.

• The Maximum Memory Range sets the limit for the amount of memory that can be used by the microbenchmarks.

• Observing for jump at any stride would require a call to the DetectJump() subroutine.

• 'S', 'D', 'F' and 'Z' are flags that record the trend for every H in Trend[] array and DetectSize() subroutine uses this array to observe a change in trend and record that as the Page Size or Line Size.

3.3.4.1 Decrease()

This subroutine tests for properties of Pattern 1 as described previously and uses the input parameters H,S to determine the spots at which a jump due to decrease in number of spots is expected. The subroutine runs over multiple values of lower stride L for each H and S and hence returns a confidence value i.e. the number of occurrences of pattern 1 when tested for each combination of H,L,S
3.3.4.2  *Increase()*

This subroutine tests for properties of Pattern 2 as described previously and uses the input parameters H,S to determine the spots at which a jump due to increase in number of spots is expected. The subroutine runs over multiple values of lower stride L for each H and S and hence returns a confidence value i.e. the number of occurrences of pattern 2 when tested for each combination of H,L,S

3.3.4.3  *FullyAssociative()*

A fully associative entity does not exhibit the properties of either Pattern 1 or Pattern 2. This subroutine tests for the property of 'neither increase nor decrease' in spots at which the jump occurs, even with added lower stride. It returns a confidence value based on the number of occurrences of no change when tested for each combination of H,L,S

3.3.4.4  *DetectSize()*

This subroutine runs over the recorded trend and returns the stride at which there is switch from pattern 1 to pattern 2. This stride value is the block size of the entity.

3.4  *Similarity between TLB and Cache*

The approach described above worked with the example of TLB. The same approach can be used to determine the cache properties. The page size of TLB is analogous to the line size, $B(=2^B)$ in Cache, and the same equations/methodology can be used to determine the cache properties.
Case I: \( H \leq 2^b \)

When the value of the higher stride is lesser than the line size, the access pattern would be such that we would end up filling the Cache. For example, on Nehalem, the L1-Cache size is 32KB, 8-way associative and 64 sets. The line size is 64 bytes. With a stride of 32, two accesses would map to the same line, but a third access would map to a new line. This line in-turn maps to a new set in the Cache, since we are accessing the immediate next line. Therefore, we would expect the jump to happen at 1024 spots, as given by Eq (3.1). Now consider a lower stride of 16. It can be seen that once in every two spots, we would “affect” the higher stride and therefore change the pattern in which we would access the different sets in Cache. What would have been 3 accesses earlier, would now be just 2 accesses. For example, with a stride of just 32, we would access addresses, 0, 32, 64, 96, 128… The equivalent sets are 0, 0, 1, 1…. With a modified stride of 32 + 16, we would access addresses, 0, 48, 96, 144, 192…. The equivalent sets are 0, 0, 1, 2…. It is seen that the second access to set 1 is missed. The access pattern is varied such that we would not fill up the Cache, since we would miss a certain number of sets. Therefore the number of spots at which the jump occurs, \( A \), would reduce. We can quantify it with the set of equations given in (3.4).

\( S \) is given from Eq (3.1). For example, with \( H = 32 \) and \( L = 16 \); from Eq (3.1) we get \( S = 1024 \) and the number of spots at which the jump occurs would be \( A \approx 683 \). As another example, if the lower stride was 8, the value of \( A \) would be \( 1024 \times 4/5 \approx 819 \) spots.
Case II : $H > 2^h$

When the higher stride being used is more than the line size, the accesses pattern would be such that, it would not fill up the Cache. That is there would be sets in the Cache, which are not accessed at all. For example, on Intel Core i7, with a stride of 128, only 32 of the 64 sets would be accessed, i.e., the spots accessed would be say, 0, 128, 256, 384, 512, . . . . The sets accessed would be 0, 2, 4, 6, 8 . . . . Therefore the jump would be seen at 256 spots (since it is 8-way associative). On introducing a lower stride, when sufficient number of spots are accessed, the lower stride will change the access pattern, to access pages which map to the unaccessed sets before. For example, for a stride of $128 + 64$, the access pattern would be $0, 192, 384, 576, 768, . . . .$, which would map to sets $0, 3, 6, 9, 12, . . . .$. On writing out all the sets accessed, it can be seen that all the previously missed sets (the odd numbered ones) will be now accessed, thus increasing the number of spots at which the jump occurs to 512 in this example. Some key points given in the previous section for TLB is applicable here as well.

3.5 Determining the Miss Penalties

Once we have the sizes of all entities, we can now go ahead and try to quantify the miss penalties. Lets work with an example of trying to quantify the miss penalty of the L1 Cache size on Intel core i7. The Cache size on Intel Core i7 is 32 Kb, the line size is 64 bytes, the number of sets is 64 and it is 8-way associative. Suppose we go with a higher stride of 64, the number of spots at which we would fill up the Cache would be 512. On increasing the number of spots by 1, we would have 2 conflict misses, one for the last memory access
which would map to the same set as the first memory access, and one due to the access to
the first memory location on the next run. Therefore, we can quantify the jump by,

\[ T = \frac{n \cdot T_H + m \cdot T_m}{n + m} \]  \hspace{1cm} (3.6)

where \( n \) and \( m \) is the number of hits and number misses respectively; and \( T_H \) and \( T_M \)
is the hit time and miss time respectively. The value of \( T_H \) would be same as the Average
Memory Access Time for 512 spots ( when all accesses were hits ). For the example used
earlier, \( n = 511 \) and \( m = 2 \). On increasing the number of spots by 1, the value of \( n \) and
\( m \) would be 4 and 512, thus increasing the AMAT by a small amount. This incremen-
tal increase would happen upto the point were we read \( 2^e \) number of spots. After this, the
AMAT would not increase due to the miss in L1 Cache since there would be conflict misses
in all the sets in Cache. Here we assume that the cache employs a LRU replacement policy,
which results in associativity not being a factor. For example, for the 8-way associative L1
Cache on Nehalem, on accessing 513 spots, the 513\(^{th}\) memory access would replace the
first memory access, since that was least recently used. On the next run when we access the
first memory location again, we would replace the 65\(^{th}\) memory access ( since that would
map to the same set and was least recently used ) and so on.

This characterization of step-wise increase in AMAT helps us determine the miss penalty
for every increase in number of spots. Since there can be noise in the measurement of time,
we get a number of sample points to get confidence in our measurements. Since we can
also calculate what the AMAT should be based on previous measurements, if we dont see
a value close to this, we can conclude that there is an interference with some other entity,
and further measurements are not possible.
**TLB Example**

The latency measurements given by Calibrator are in Nanoseconds(ns).

- $H = 4096$, $L = 32$. At $S = 63$, $T = 353\text{ns}$.
- At $S = 64$, $T = 353\text{ns}$. This shows there is no jump from 63 to 64. TLB is still being filled.
- At $S = 65$, $T = 373\text{ns}$.
- At $S = 66$, $T = 393\text{ns}$
- At $S = 67$, $T = 411\text{ns}$ The AMAT resembles a step function that increases for every set missed. The function will reach a peak at $S + E$, Where $E$ is the number of sets in TLB.
- At $S = 80$, $T = 600\text{ns}$
- At $S = 81$, $T = 600\text{ns}$ From this data, We can characterize Miss Penalty ($M_P$). $T_H = 353\text{ns}$, $T_M = 600\text{ns}$. $M_P = T_M - T_H = 600\text{ns} - 353\text{ns} = 246\text{ns}$

Using (3.6), Taking the case of $S = 65$, there are 2 misses due to 1 set mapped twice. When $n = 63$, $m = 2$, $T_H = 353\text{ns}$, $T = 373\text{ns}$; On solving, we get $T_M = 650\text{ns}$. Taking the case of $S = 73$, there are 18 misses due to 9 sets mapped twice. When $n = 55$, $m = 18$, $T_H = 353\text{ns}$, $T = 512\text{ns}$; On solving, we get $T_M = 644\text{ns}$. On using multiple samples and taking average, We will get a good estimate of the Miss Penalty of the entity.
Cache Example

- \( H = 64, \ L = 0. \) At \( S = 511, \ T = 346\text{ns}. \)

- At \( S = 512, \ T = 346\text{ns}. \) This shows there is no jump from 511 to 512. Cache is still being filled.

- At \( S = 514, \ T = 354\text{ns}. \)

- At \( S = 523, \ T = 390\text{ns} \)

- At \( S = 540, \ T = 449\text{ns} \) The AMAT resembles a step function that increases for every set missed. The function will reach a peak at \( S + E, \) Where \( E \) is the number of sets in Cache. Here \( E = 64, \) Hence function peaks at \( S = 512 + 64 = 576. \)

- At \( S = 576, \ T = 565\text{ns} \)
At $S = 577$, $T = 565$ns From this data, We can characterize Miss Penalty ($M_P$). $T_H = 346$ns, $T_M = 565$ns. $M_P = T_M - T_H = 565\text{ns} - 346\text{ns} = 219\text{ns}$

Figure 3.4: Step Function of Miss Time for Cache
Chapter 4: RESULTS

In this section we compare the results from running the *lmbench v3.0a9* and *Calibrator v0.9e* with the results obtained by running a modified version of calibrator that implements the approach described in previous sections. We also list the parameters obtained from the Intel [1] and AMD [2] processor manuals and software optimization documentation that contain the hardware information.

4.1 Intel Core i7 - Nehalem architecture

<table>
<thead>
<tr>
<th>Approach</th>
<th>Capacity</th>
<th>Associativity</th>
<th>Line Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>32 KB</td>
<td>8</td>
<td>64 byte</td>
</tr>
<tr>
<td>Lmbench</td>
<td>32 KB</td>
<td>-</td>
<td>64 byte</td>
</tr>
<tr>
<td>Calibrator</td>
<td>32 KB</td>
<td>-</td>
<td>64 byte</td>
</tr>
<tr>
<td>Our approach</td>
<td>32 KB</td>
<td>8</td>
<td>64 byte</td>
</tr>
</tbody>
</table>

Table 4.1: L1 Data cache comparison with other approaches for Intel Core i7 920 @ 2.67Ghz
<table>
<thead>
<tr>
<th>Approach</th>
<th>Capacity</th>
<th>Associativity</th>
<th>Line Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>256 KB</td>
<td>8</td>
<td>64 byte</td>
</tr>
<tr>
<td>Lmbench</td>
<td>256-384 KB</td>
<td>-</td>
<td>256 byte</td>
</tr>
<tr>
<td>Calibrator</td>
<td>320 KB</td>
<td>-</td>
<td>256 byte</td>
</tr>
<tr>
<td>Our approach</td>
<td>256 KB</td>
<td>8</td>
<td>64 byte</td>
</tr>
</tbody>
</table>

Table 4.2: L2 Data cache comparison with other approaches for Intel Core i7 920 @ 2.67Ghz

<table>
<thead>
<tr>
<th>Approach</th>
<th>No. of Entries</th>
<th>Associativity</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>64</td>
<td>4</td>
<td>4KB</td>
</tr>
<tr>
<td>Lmbench</td>
<td>32</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Calibrator</td>
<td>80</td>
<td>-</td>
<td>4KB</td>
</tr>
<tr>
<td>Our approach</td>
<td>64</td>
<td>4</td>
<td>4KB</td>
</tr>
</tbody>
</table>

Table 4.3: L1 TLB comparison with other approaches for Intel Core i7 920 @ 2.67Ghz

<table>
<thead>
<tr>
<th>Approach</th>
<th>No. of Entries</th>
<th>Associativity</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>512</td>
<td>4</td>
<td>4KB</td>
</tr>
<tr>
<td>Lmbench</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Calibrator</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Our approach</td>
<td>512</td>
<td>4</td>
<td>4KB</td>
</tr>
</tbody>
</table>

Table 4.4: L2 TLB comparison with other approaches for Intel Core i7 920 @ 2.67Ghz
### 4.2 Intel Core 2 Quad

<table>
<thead>
<tr>
<th>Approach</th>
<th>Capacity</th>
<th>Associativity</th>
<th>Line Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>32 KB</td>
<td>8</td>
<td>64 byte</td>
</tr>
<tr>
<td>Lmbench</td>
<td>32 KB</td>
<td>-</td>
<td>64 byte</td>
</tr>
<tr>
<td>Calibrator</td>
<td>32 KB</td>
<td>-</td>
<td>64 byte</td>
</tr>
<tr>
<td>Our approach</td>
<td>32 KB</td>
<td>8</td>
<td>64 byte</td>
</tr>
</tbody>
</table>

Table 4.5: L1 Data cache comparison with other approaches for Intel Core2 Quad - Q6670

<table>
<thead>
<tr>
<th>Approach</th>
<th>No. of Entries</th>
<th>Associativity</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>16</td>
<td>4</td>
<td>4KB</td>
</tr>
<tr>
<td>Lmbench</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Calibrator</td>
<td>320</td>
<td>-</td>
<td>4KB</td>
</tr>
<tr>
<td>Our approach</td>
<td>16</td>
<td>4</td>
<td>4KB</td>
</tr>
</tbody>
</table>

Table 4.6: L1 TLB comparison with other approaches for Intel Core2 Quad - Q6670

<table>
<thead>
<tr>
<th>Approach</th>
<th>No. of Entries</th>
<th>Associativity</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>256</td>
<td>4</td>
<td>4KB</td>
</tr>
<tr>
<td>Lmbench</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Calibrator</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Our approach</td>
<td>256</td>
<td>4</td>
<td>4KB</td>
</tr>
</tbody>
</table>

Table 4.7: L2 TLB comparison with other approaches for Intel Core2 Quad - Q6670
4.3 Intel Core2 Duo P8700

<table>
<thead>
<tr>
<th>Approach</th>
<th>Capacity</th>
<th>Associativity</th>
<th>Line Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>32 KB</td>
<td>8</td>
<td>64 byte</td>
</tr>
<tr>
<td>Lmbench</td>
<td>32 KB</td>
<td>-</td>
<td>64 byte</td>
</tr>
<tr>
<td>Calibrator</td>
<td>32 KB</td>
<td>-</td>
<td>64 byte</td>
</tr>
<tr>
<td>Our approach</td>
<td>32 KB</td>
<td>8</td>
<td>64 byte</td>
</tr>
</tbody>
</table>

Table 4.8: L1 Data cache comparison with other approaches for Intel Core2 Duo - P8700

<table>
<thead>
<tr>
<th>Approach</th>
<th>No. of Entries</th>
<th>Associativity</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>16</td>
<td>4</td>
<td>4KB</td>
</tr>
<tr>
<td>Lmbench</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Calibrator</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Our approach</td>
<td>16</td>
<td>4</td>
<td>4KB</td>
</tr>
</tbody>
</table>

Table 4.9: L1 TLB comparison with other approaches for Intel Core2 Duo - P8700

<table>
<thead>
<tr>
<th>Approach</th>
<th>No. of Entries</th>
<th>Associativity</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>256</td>
<td>4</td>
<td>4KB</td>
</tr>
<tr>
<td>Lmbench</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Calibrator</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Our approach</td>
<td>256</td>
<td>4</td>
<td>4KB</td>
</tr>
</tbody>
</table>

Table 4.10: L2 TLB comparison with other approaches for Intel Core2 Duo - P8700
### 4.4 Intel Xeon 3070

<table>
<thead>
<tr>
<th>Approach</th>
<th>Capacity</th>
<th>Associativity</th>
<th>Line Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>32 KB</td>
<td>8</td>
<td>64 byte</td>
</tr>
<tr>
<td>Lmbench</td>
<td>32 KB</td>
<td>-</td>
<td>64 byte</td>
</tr>
<tr>
<td>Calibrator</td>
<td>32 KB</td>
<td>-</td>
<td>64 byte</td>
</tr>
<tr>
<td>Our approach</td>
<td>32 KB</td>
<td>8</td>
<td>64 byte</td>
</tr>
</tbody>
</table>

Table 4.11: L1 Data cache comparison with other approaches for Intel Xeon - 3070

<table>
<thead>
<tr>
<th>Approach</th>
<th>No. of Entries</th>
<th>Associativity</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>16</td>
<td>4</td>
<td>4KB</td>
</tr>
<tr>
<td>Lmbench</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Calibrator</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Our approach</td>
<td>16</td>
<td>4</td>
<td>4KB</td>
</tr>
</tbody>
</table>

Table 4.12: L1 TLB comparison with other approaches for Intel Xeon - 3070

<table>
<thead>
<tr>
<th>Approach</th>
<th>No. of Entries</th>
<th>Associativity</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>256</td>
<td>4</td>
<td>4KB</td>
</tr>
<tr>
<td>Lmbench</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Calibrator</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Our approach</td>
<td>256</td>
<td>4</td>
<td>4KB</td>
</tr>
</tbody>
</table>

Table 4.13: L2 TLB comparison with other approaches for Intel Xeon - 3070
4.5 AMD Phenom

<table>
<thead>
<tr>
<th>Approach</th>
<th>Capacity</th>
<th>Associativity</th>
<th>Line Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Manual</td>
<td>64 KB</td>
<td>2</td>
<td>64 byte</td>
</tr>
<tr>
<td>Lmbench</td>
<td>64 KB</td>
<td>-</td>
<td>64 byte</td>
</tr>
<tr>
<td>Calibrator</td>
<td>64 KB</td>
<td>-</td>
<td>64 byte</td>
</tr>
<tr>
<td>Our approach</td>
<td>64 KB</td>
<td>2</td>
<td>64 byte</td>
</tr>
</tbody>
</table>

Table 4.14: L1 Data cache comparison with other approaches for AMD Phenom 9850
Chapter 5: CURRENT LIMITATIONS AND FUTURE WORK

- The last level of cache is physically addressed. Our approach with sums of strides would work only with entities that are virtually addressed since the stride property needs to be maintained. For Ex: A stride of 64 bytes would be meaningful with a virtually addressed entity since the virtual addresses are accessed at a stride of 64. If the entity were physically addressed and we generate virtual addresses at stride 64, the actual stride can vary arbitrarily depending on the mapping between physical and virtual addresses. It is possible that the 64th byte is actually 1000 bytes away in physical memory, which makes the strided access meaningless.

- This issue can be possibly addressed by forcing the allocation of physically contiguous memory. This is done using the kmalloc() system call which allocates physically contiguous kernel memory. The system call can be invoked by inserting the code as a kernel module.

- AMD processors exhibit behavior that makes it hard to iron out the interferences by reaching a higher stride. There is possibly a replacement-buffer that exists between L1 and L2 cache that is repeatedly accessed such that it throws off measurements.
• This limitation can be subverted by not attempting to reach the higher stride and
detect the entities as much as possible to let the disambiguation help with character-
ization.

• The current approach does not account for Huge pages or Large pages i.e. pages that
hold more than 1MB of data. Modern architectures have inbuilt support for large
pages and have dedicated entries in the TLB for large pages.

• The framework needs to be extended for variable page sizes as well since large pages
can have multiple page sizes.
Bibliography


