Techniques for an Image Space Occlusion Culling Engine

Leandro R. Barbagallo, Matias N. Leone, Mariano M. Banquiero, Diego Agromayor, Andres Bursztyn

Proyecto de Investigación "Explotación de GPUs y Gráficos Por Computadora", GIGC - Grupo de Investigación de Gráficos por Computadora, Departamento de Ingeniería en Sistemas de Información, UTN-FRBA, Argentina

{lbarbagallo, mleone, mbanquiero, dagromayor }@frba.utn.edu.ar
andresb@sistemas.frba.utn.edu.ar

Abstract. In this work we present several techniques applied to implement an Image Space Software Occlusion Culling Engine to increase the speed of rendering general dynamic scenes with high depth complexity. This conservative culling method is based on a tiled Occlusion Map that is updated only when needed, deferring and even avoiding the expensive per pixel rasterization process. We show how the tiles become a useful way to increase the speed of visibility tests. Finally we describe how different parts of the engine were parallelized using OpenMP directives and SIMD instructions.

Keywords: Rasterization, Occlusion Culling, Visibility Algorithms, Hierarchical Occlusion Map, Lazy Grid, Tiles, Depth Buffer, SIMD, OpenMP, GPU.

1 Introduction

In Real-Time Computer Graphics, rendering only the objects that will contribute to the final image is of vital importance especially when rendering large and complex 3D scenes. Visibility algorithms are in charge of determining which objects or parts of them will be visible from a given viewpoint, and in order to address this issue many techniques have been developed since 1970s [1]. Among these techniques is Occlusion Culling which aims to avoid rendering objects that are occluded by other objects in the scene by discarding them at an early stage in the rendering pipeline. The most widely used Occlusion Culling technique nowadays is the Z-Buffer algorithm that discards invisible fragments based on the comparison between the depth information provided in the rasterizing stage and the values previously stored in the Z-Buffer. Despite being implemented in hardware, this method is applied at an advanced stage in the rendering pipeline, where the geometry has already been converted to fragments, potentially becoming a bottleneck in the application.

A conservative and a image space Occlusion Culling technique that best adapts to complex environments with high depth complexity is the Hierarchical Occlusion Map (HOM) proposed by Zhang [2] which chooses a small subset of occluders and rasterizes them in a buffer to store both occluder and depth information in their

respective Occlusion Map and Depth Buffer. Then to perform the visibility test it divides the problem into two phases: a two dimensional overlap test where the occludee screen space projected bounding box is tested against the union of all occluders represented by the Occlusion Map and a second test which is the depth test performed using the Depth Buffer. To avoid the overhead of the per pixel depth comparison in every test, Zhang proposes buffers that are averaged and down-sampled forming a hierarchical pyramid, taking advantage of the bilinear texturing sampling capabilities available in modern GPUs.

One improvement to HOM suggested by Hey et al. [3] is to divide the occlusion buffer in a low resolution grid that is updated only when needed in a lazy manner, consequently avoiding unnecessary rasterization work and also avoiding the expensive pixel level depth comparisons in the overlap and depth test. Conversely, Décoret [4] suggested an alternative to the HOM image pyramid that performs the depth comparison in constant time storing N number of buffers of the same size each one with the maximum depth of all the pixels around a given area.

To speed up the image space occlusion culling process, modern graphics hardware support natively a special rendering mode which the user can query the Z-Buffer to determine whether a simplified version of a complex object such as its bounding volume is visible or not [5] [6] [7]. The GPU hardware rasterizes the query object and compares the fragment depths with the ones stored in the Z-Buffer. If the count of fragments that passed the depth test is zero, then the whole object can be culled, otherwise the whole object is considered visible. However, a direct implementation using this hardware capability does not produce the best results as stated by Bittner and Wimmer [5]: “Although the query itself is processed quickly using the raw power of the graphics processing unit (GPU), its result is not available immediately due to the delay between issuing the query and its actual processing in the graphics pipeline. As a result, a naive application of occlusion queries can even decrease the overall application performance due the associated CPU stalls and GPU starvation.” Techniques that exploit temporal coherence are often applied to mitigate these issues.

Nevertheless, this query delay motivated the development of other different approaches, and one of them is the Software Occlusion Culling Queries processed in the CPU in an attempt to reduce this latency taking advantage of the increasing number of multicore systems and SIMD vector instructions available in modern systems. These techniques were used to speed up video games especially in game console systems as shown in [8] [9] [10].

In this paper we implement a SIMD Optimized Software Occlusion Map Rasterizer that uses a low resolution grid, that divides the screen into cells or tiles to rasterize the image space projected bounding boxes of complex objects as a series of quads. The occluder fusion is performed at image space by rasterizing the quads into the Depth Buffer only when it is absolutely needed.

2 Lazy Occlusion Grid Engine

2.1 Overview

In this section we will do an overview of how our Tiled Occlusion Culling Engine works. The whole purpose of this Occlusion Engine is to perform occlusion queries, which is to determine whether a given occludee is visible or not with respect of a given set of occluders. Therefore the Occlusion Engine has to solve two different things; First it has to accept a limited set of carefully chosen simplified occluders [11], and second it has to perform the overlap test for the occludees to verify if they are visible with respect of the selected occluders.

As the chosen occlusion culling approach belongs to the 2D image space category [1], the occluders need to be transformed from 3D object space to 2D image space. Initially the engine receives the occluders in the form of 2D projected quads[1] each one consisting of four points that form a coplanar four-sided convex hull. These points are already projected in 2D screen space with the depth value normalized between 0.0 and 1.0. The engine assumes that the given occluder quads already passed backface cull test, meaning that only the visible faces of the bounding box are sent to the Engine.

For every occluder quad received, the Occlusion Engine performs a classification of the tiles that are overlapped by the quads. With this new tile classification, the engine decides whether the part of the quad that falls into the tile should be rasterized pixel by pixel whether it just simply needs to update the tile properties.

For the tiles that need to be rasterized, the quad is scan converted using the half-space [12] method and the depth values are linearly interpolated based on the first three vertices values of the quad. On the contrary, for the tiles that are now fully covered by the quad and the depth is less than the tile's previous content, the engine just performs an update to its internal properties updating the depth values at three of its corners.

The result of this occluder rasterization stage is the Occlusion Map and Depth Buffer combined into one single tiled Depth Buffer stored in main memory. The Occlusion Map is represented as all the tiles that have a depth value of 1.0 and unlike Hierarchical Occlusion Map (HOM), our method does not build the full multi resolution Occlusion Map Hierarchy, so it could be considered as a level 0 and a level N (HOM), being the latter the occlusion depth buffer tiled grid.

Finally for visibility determination, the engine receives the query composed of a 2D Axis Aligned Bounding Box that approximate the occludee projected image and a

---

[1] Note: Triangles are accepted primitives as well because quads may require 2D screen clipping.

constant depth calculated as the closest point to the camera of all the bounding volume points.

The occlusion test is done first at a coarse level by checking whether any of the tiles inside the occludee region is in *NotInitialized* status, and if none is found then it proceeds to perform the depth comparison, where first it checks for the tile minimum depth and, as a last resource, the engine performs a pixel level depth comparison.

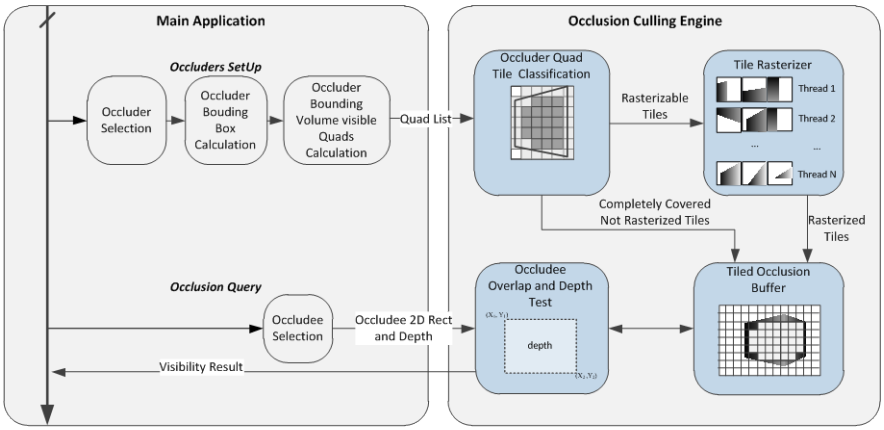

Fig. 1: How the Occlusion Culling Engine communicates with the main application. Occluder setup and Occlusion Queries are the most important functions used.

2.2 Occlusion Map Tiles

In this approach the Occclusion Map and Depth Buffer are combined into a single Occlusion Buffer which is divided into a grid with tiles of equal size, each with its own status data. This data consists of the Tile Status which can be one of the following:

- *NotInitialized*: There are no occluders or parts of them inside the region of that tile.
- *CompletelyCoveredNotRasterized* : A single occluder completely covers the region of the tile and its depth in all its points is less than any possible preexisting occluder point depths.
- *Rasterized*: The tile is covered by one or more occluders or parts of them, having different depth values.

Initially when no occluders are set, the Depth Buffer is composed solely of *NotInitialized* tiles and, as the occluders start to arrive to the engine for classification and processing, the tiles may change their status and their inner properties.

When the user wants to send an occludee to the Engine, the bounding box of the object is calculated and the visible faces of the bounding volume are sent to the

Occlusion Culling Engine as separate quads consisting of their four points and their depth values. Thereafter, based on the convex polygon defined by the quad, the engine classifies the screen tiles that intersect it, using the technique proposed by Greene [13] into three kinds: Outside, Border and Inside tiles. To speed up the process of classifying the tiles, we use the Trivial Reject Corner and Trivial Accept Corner suggested by Abrash [14].

The Outside kinds of tiles are quickly rejected and no changes to the screen tile original status or properties are done. However a different treatment must be done for Border and Inside tiles.

For the tiles that have been classified as Border (i.e partially covers a tile), the rasterizer will have to scan-convert the portion of the quad inside the region of the tile pixel by pixel, performing the depth test similar as what the Z-Buffer does.

Nevertheless there is a case when no rasterization needs to be done at all, and it is when the screen tile is in *CompletelyCoveredNotRasterized* status and the occluder minimum depth is greater than the minimum depth already stored in the tile. In this case, it results the same as performing a Z-Buffer algorithm, but at a coarser level, that is at a tile grid level, avoiding the expensive rasterization process.

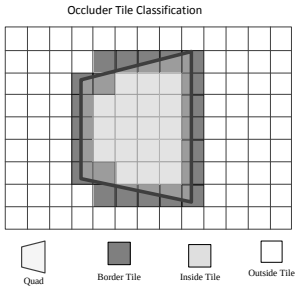

Fig. 2: This Occluder Tile Classification example shows how a quad (black polygon) is tested against the occlusion depth buffer tiles into Border, Inside and Outside tiles.

For the Inside kind of tiles there are more options to consider. For example, if the screen tile status was *NotInitialized* , we defer the rasterization in a lazy manner, since we only need to store the quad plane equation or just three depth points at the tile extreme corners. Taking advantage of the quad coplanarity property, the tile can be later rasterized when needed based on the three previously stored depth values.

On the other hand, if the screen Tile status was *CompletelyCoveredButNot Rasterized*, we need to compare the depth values at the extreme corners. If all the occluder depth values are less than the ones previously stored in the screen tile, then the occluder overlaps the whole tile content; If all depths are greater, then the occluder is occluded and the tile is left with the original depth values and properties. The last case occurs when some values are less and others not, in that case the

occluder needs to be rasterized in order to calculate which fragments have less depth than the ones stored previously in the screen tile.

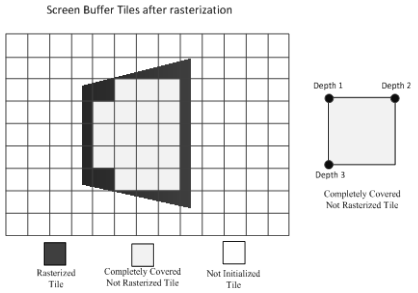

Fig. 3: This example shows how an empty occlusion depth buffer is updated with the content of an occluder quad. In this case only the border tiles are rasterized while the inside tiles becomes *CompletelyCoveredNotRasterized* and only the tile information (i.e. depth values at the tile extreme corners) is updated.

As shown earlier, the tile is only rasterized when it is absolutely needed, that is when the tile is not fully covered and when we need to count with greater depth detail at pixel level. This way of organizing the screen into a low resolution gird of tiles will later be of great use when doing the occludee overlap and depth test. We can say that the advantage of this method is that we gain more precision for occlusion detection when increasing the occlusion map size, but as we are using the tiled grid we get a similar performance as a much lower resolution occlusion map size.

2.3 Tile Rasterization

After the tile classification is done, a set of rasterizable tiles are sent to the rasterizer unit, which depending on the number of available threads in the system, equally distributes the load into a the processing queues [15].

When rasterizing convex quads we take advantage of the fact that we only need to perform a half-space test with four edges in order to check that the point to rasterize is contained inside the primitive. The edge test can be done four at a time utilizing the SIMD instructions available in modern multicore processors.

Another property that we exploit is that the quad is contained inside a plane. Once calculated these plane coefficients, every depth value of every point inside the hull can be obtained by replacing the point position in the plane equation. An optimization to avoid evaluating the plane equation in every pixel is to have accumulator variables so we only have to do additions of the plane's A and B coefficients every time the column or row changes in the X and Y nested rasterization loops [16].

Finally there is a special case in which the occlusion buffer tile is in *Completely CoveredButNotRasterized* status, so the tile doesn't contain pixel level information, only contains the extreme depth values at its corners. In case quad falls into this tile

and needs to be rasterized, the previous content has to be rasterized first. In this case we just make a full tile cover taking advantage that no half-space check has to be done, thus speeding up the fill process significantly, and then as soon as the tile is filled, the new quad portion can be rasterized over it.

2.4 Occludee Visibility Test

After all the occluders have been sent to the engine and the occlusion depth buffer is processed, the application may want to test if a given set of occludees are visible. To perform this, the first step is to get every occludee object's screen aligned bounding box to overestimate its coverage area. The second step is to get the depth by getting the occludee closest point with respect of the view point. Given these two elements, the 2D bounding box and the depth, the occludee is sent to the engine.

The trivial solution to solve the visibility problem would be to compare each point of depth buffer inside the region of the occludee bounding box and check if there is at least one occludee point that is closer to the view point. The downside of this approach is that it involves accessing the depth buffer pixel per pixel which may not be the best option performance wise.

However, since the depth buffer proposed is divided into tiles, we can take advantage of several facts: The first one is that if we find that the occludee bounding box contains at least a tile in *NotInitialized* status, then the test query trivially returns a positive visibility result in an early manner. The second one is that we can utilize the minimum depth tile property to avoid per pixel depth comparisons when possible.

Finally, when all the previous options have been discarded and no positive visibility result is returned, the traditional pixel level test is performed.

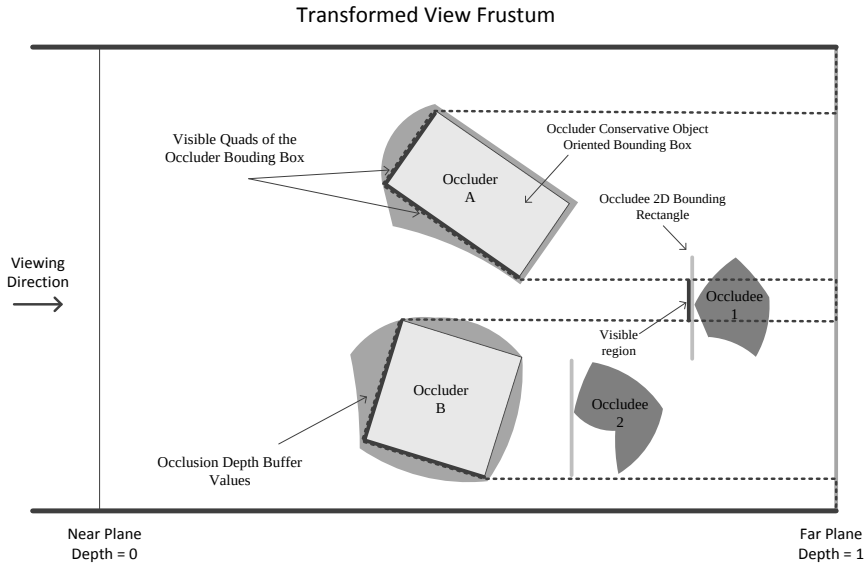

Fig. 4: This example shows a transformed view frustum with a dotted line representing the depth values of the depth buffer when two occluders are rasterized in the Engine. Occludee 1 will result visible since there is a sub region in its rectangle in which the depth of the occludee is less than the value stored in the depth buffer. The Occludee 2 is completely culled by Occluder B. Only the visible faces of the Occluders conservative bounding boxes are sent to the engine as quads.

3 Implementation and Results

We have implemented our Software Occlusion Culling Engine in C++, using Visual Studio 2010 to compile it as a DLL module, and tested different scenarios using a PC with Intel Core 2 Duo 2.40 Ghz with 4GB RAM. The engine was integrated as a module in a 3D interactive application, where it was tested in a city scene with a high number of objects giving satisfactory results.

The occlusion engine consists mainly of the functions that perform the following tasks:

- Initialize the engine: Performs the initial sets up of the occlusion buffer and creates the tiles.
- Set Occluders: Receives and processes a list of occluders.
- Test Occludee Visibility: Determines if a given occludee is visible.

The rasterization of the tiles is parallelized using OpenMP 2.0 [17] by specifying simple directives in the “for” loop construct that iterates the rasterization tile queue. The load balancing between threads is done automatically using the dynamic scheduling clause available in OpenMP, however we found that this mode made the

rasterization time more irregular and did not perform better than static scheduling mode.

At a more fine grained data parallelism, we optimized the intra tile rasterization using the SSE3 and SSE4 SIMD instructions focusing on two main areas:

- Perform the half space test for the four edges of the quad using four 32 bit integer accumulators packed into a single 128bit XMM register. Only additions to the accumulator are done in the rasterization inner loops, reducing significantly the time taken to generate the coverage mask [18] [16].
- SSE4 Blend instruction was used to avoid conditional branching [19] when comparing depth values between occlusion depth buffer and occludee fragment value. This attempts to remediate the possible branch mispredictions caused by the depth comparison conditional.

The occlusion depth buffer grid is composed of an array of equally sized tiles, whose size was defined arbitrarily depending on several factors, such as resolution, system configuration and characteristics of the scene [3]. Since each thread works with one tile at a time, we found wise to turn a linear depth buffer into a nonlinear tiled depth buffer in which every row of a tile is placed contiguously in memory, one row next to the other. This loop blocking technique [19] has two advantages, first it tends to reduce the cache misses since all the pixels in the tile are closer together and the other advantage is that each point in the inner loop can be accessed by increasing a simple pointer variable, avoiding the need to convert from 2D framebuffer coordinates into a memory linear address.

Regarding the performance results, we measured the engine response in different scenarios. We were interested in how the engine responds to different tile sizes and in different scene environments and we found that a tile size of 16 or 32 pixels gave the best performance results in the majority of the cases.

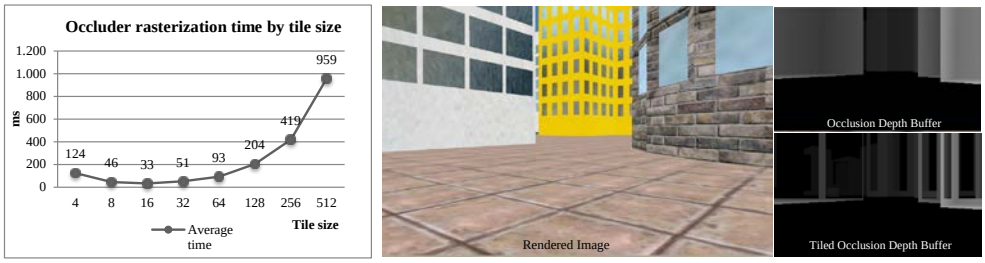

Fig. 5: Left: This line chart shows the average rasterization time of 1000 random occluder quads in a 512x512 pixels occlusion depth buffer when different tile sizes are used. Right: A city rendered image and two versions of the Occlusion Depth Buffer; non tiled and using 16 pixel size tiles.

4 Future work

Even though the average performance of our Software Occlusion Culling Engine is within an acceptable range, there are still pending optimization techniques: Doing

some depth sorting method of the quads to avoid quick tile rejection, applying parallelism in the tile classification stage and reducing the memory footprint required to store the occlusion depth buffer tile grid.

We also found that the rasterization stage could benefit from using the x86 Advanced Vector Extensions (AVX) instructions that could potentially result twice as fast by processing eight pixels at a time by storing its coverage masks and edge accumulators in 256 bit YMM registers [19].

5 Conclusions

We have presented our Software Occlusion Culling Engine and introduced the techniques used to improve its overall effectiveness and efficiency. The lazy update of the low resolution grid helps the occlusion culling process to delay or even avoid expensive operations. Moreover, applying SSE instructions to selected spots of the code enhanced the engine general performance by parallelizing tile rasterization.

The GPU accelerated occlusion queries will still continue to rasterize orders of magnitude faster than any CPU approach, however, as long as the GPU and CPU remain as separate cores, the latency required to transfer the query results will continue being an issue in performance.

When hardware accelerated occlusion queries are not available, software approaches have proven to be very effective and many production level applications were shipped using this technology. As result of this we estimate that as the multicore CPUs keep increasing their number of cores and expanding their SIMD instruction sets, software image-based occlusion culling is going to become a more utilized technique in the following years.

6 References

1. Cohen-Or, D., Chrysanthou, Y. L., Silva, C. T., Durand, F.: A survey of visibility for walkthrough applications. Visualization and Computer Graphics, IEEE Transactions on Visualization and Computer Graphics 9(3), 412-431 (2003)
2. Zhang, H., Manocha, D., Hudson, T., Hoff, K.: Visibility Culling Using Hierarchical Occlusion Maps. In : In Computer Graphics (Proceedings of SIGGRAPH ´97), Los Angeles, CA, pp.77-88 (August 1997)
3. Hey, H., Tobler, R., Purgathofer, W.: Real-Time Occlusion Culling with a Lazy Occlusion Grid. In : Proceedings of the 12th Eurographics Workshop on Rendering Techniques, London, UK, UK, pp.217-222 (2001)
4. Decoret, X.: N-Buffers for efficient depth map query. Computer Graphics Forum 24(3), 393-400 (2005)
5. Bittner, J., Wimmer, M., Piringer, H., Purgathofer, W.: Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. Computer Graphics Forum 23(3), 615-624 (2004)
6. Akenine-Moller, T., Haines, E., Hoffman, N.: Real-Time Rendering 3rd Edition. A. K. Peters, Ltd., Natick, MA, USA (2008)

7. NVIDIA Corporation: NV_occlusion_query. (Accessed February 6, 2002) Available at: http://www.opengl.org/registry/specs/NV/occlusion_query.txt
8. Vale, W.: Practical occlusion culling in KILLZONE 3. In : SIGGRAPH Talks, p.49 (2011)
9. Andersson, J.: Parallel Graphics in Frostbite-Current & Future. SIGGRAPH Course: Beyond Programmable Shading (2009)
10. Narkowicz, K.: Software occlusion culling. (Accessed April 2012) Available at: http://kriscg.blogspot.com/2010/09/software-occlusion-culling.html
11. Germs, R., Jansen, F.: Geometric Simplification For Efficient Occlusion Culling In Urban Scenes. In : Proc. of WSCG 2001, pp.291-298 (2001)
12. Olano, M., Greer, T.: Triangle scan conversion using 2D homogeneous coordinates. In : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, New York, NY, USA, pp.89-95 (1997)
13. Greene, N.: Hierarchical polygon tiling with coverage masks. In : Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, New York, NY, USA, pp.65-74 (1996)
14. Abrash, M.: Rasterization on Larrabee. (Accessed June 2012) Available at: http://www.drdobbs.com/parallel/217200602
15. Bethel, Z.: A Modern Approach to Software Rasterization. (Accessed 2011) Available at: http://cse.taylor.edu/~zbethel/MSR/ModernApproachToSR.pdf
16. Capens, N.: Advanced Rasterization. In: Advanced Rasterization. (Accessed November 2004) Available at: http://devmaster.net/forums/topic/1145-advanced-rasterization/
17. The OpenMP® API specification for parallel programming. (Accessed April 2012) Available at: http://openmp.org/
18. McCormack, J., McNamara, R.: Tiled polygon traversal using half-plane edge functions. In : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, New York, NY, USA, pp.15-21 (2000)
19. Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual. (Accessed April 2012) Available at: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html