Performance Study of Catmull-Clark Subdivision Surfaces Algorithm

Ali J. Ghandour
National Council for Scientific Research
Beirut, Lebanon
aghandour@cnrs.edu.lb

Walid J. Ghandour
Lebanese University
Beirut, Lebanon
ghandour@utexas.edu

Hassan Diab
Prime Minister
Beirut, Lebanon
diab@aub.edu.lb

Ahmad Nasri
National Council for Scientific Research
Beirut, Lebanon
anasri@cnrs.edu.lb

Abstract—**Generating smooth surfaces of arbitrary topology is a major challenge in geometric modeling, computer graphics, and scientific visualization. Subdivision surfaces have emerged as a powerful and useful technique in modeling free-form surfaces. Recursive subdivision techniques generate visually pleasing smooth surfaces through the repeated application of a fixed set of subdivision rules. Given a control mesh, a Catmull-Clark Subdivision Surface (CCSS) is generated by iteratively refining (subdividing) the control mesh to form new control meshes. CCSS implementations do not perform well and suffer from low CPU utilization because they are often waiting for data to be transferred from memory due to the repeated pointer indirections. Rapid Evaluation of CCSS is an approach that focuses on achieving maximum performance by carefully addressing caching issues. However, this approach is designed with primary target being single core machines. In this paper, we implement the Rapid Evaluation of Catmull-Clark Subdivision Surfaces (RECCSS) algorithm in C++ and measure its achieved performance. We then redesign RECCSS to leverage the potential of parallel computation on widely available multicores machines and show that Parallelized RECCSS (PRECCSS) achieves noticeable speedup when run on diverse set of machines compared to the original RECCSS implementation. In addition, we conduct a study about the data value predictability of PRECCSS load instructions using *Pin* dynamic binary instrumentation tool, and conclude that 93.25% prediction accuracy can be achieved with existing value prediction techniques.**

Index Terms—**Subdivision, Catmull-Clark, Parallel Computing, Value Prediction, Performance Analysis.**

I. Introduction

Subdivision is used as a basic primitive for the construction of arbitrary topology free-form surfaces. It provides powerful compression and detailed rendering algorithms. Subdivision surfaces have been integrated in all standard and modeling packages such as 3DMax, Maya, Softimage, Mirai, and Lightwave. Due to the intensive computations required by most existing subdivision algorithms for their evaluation, subdivision surfaces were not generally used in real time applications. Subdivision engines implementation uses a recursive process that inserts new vertices into the mesh, refines existing point positions, and updates the connectivity [6]. Four different approaches are suggested in the literature to evaluate and render subdivision surfaces, namely: (*i*) recursive evaluation [5] [4] [3], (*ii*) direct evaluation [7] [8], (*iii*) reduction to the regular setting [9] [10] and (*iv*) pre-tabulated basis function composition [1].

The scope of this work mainly focuses on the pre-tabulated basis function composition technique introduced by Catmull and Clark in [2], though it can be applied with little changes to other subdivisions methods. Catmull-Clark Subdivision Surfaces (CCSS) is widely used in commercial computer graphics software packages. Several attempts have been made to enhance its implementation, mainly the work proposed in [1] that focuses on achieving maximum performance of CCSS on standard graphics hardware using a general purpose CPU. The suggested algorithm is known by Rapid Evaluation of Catmull-Clark Subdivision Surfaces (RECCSS). RECCSS uses a table driven evaluation strategy for subdivision surfaces and presents a compromise between cache size and memory bus usage. The proposed algorithm emulates memory latency delays and takes advantage of caching and Single-Instruction Multiple Data (SIMD) instructions. Careful attention is made to caching since on-chip computation is much cheaper than memory references [1].

RECCSS is designed having in mind a single core machine. With the state of the art machines offering multiple cores and parallelism, we revisited RECCSS to exploit such opportunities. We first implement it in C++ and run it sequentially on various machines to be used as a reference. We then modify the algorithm to leverage the potential of the state of the art architectures and allow its parallel execution on multicores machines, and run it on several distinct machines with various processor capabilities. We compare the computation time required for sequential versus parallelized implementations and report results showing the speedup gained using parallelism.

Moreover, we study the predictability of the parallelized RECCSS load instructions, and show that they are highly predictable. Accurately predicting load instructions that miss the cache has the potential to significantly enhance the performance.

The rest of this paper is organized as follows. Section II reviews related work. Section III describes the implementation of the parallelized Rapid Evaluation of Catmull-Clark Subdivision Surfaces and shows performance results. Section IV studies the predictabilities of parallelized RECCSS load instructions. Finally, we conclude in Section V.

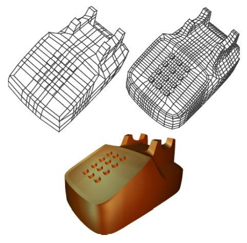
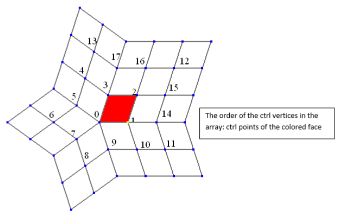


Fig. 1. A control mesh of a phone, its first subdivision, and limit surface [1].



0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
The order of the ctrl vertices in the array: ctrl points of the colored face

Fig. 2. The control structure around an extraordinary point.

II. Related Work

A. Catmull-Clark subdivision

The Catmull-Clark subdivision algorithm is one of the popular algorithms used to generate cubic free-form surfaces defined by arbitrary topological meshes. One of their main advantages is to overcome the regularity's restriction imposed by the traditional B-spline approach. It is currently widely used in software packages such as 3D max, 3D-Coat, AC3D, Anim8or, Blender, Carrara and many others. This algorithm was devised by Edwin Catmull and Jim Clark [2] as a generalization of bi-cubic uniform B-spline surfaces to arbitrary topology. The algorithm takes as input a topological 2-manifold bounded control mesh. It can consist of faces with arbitrary degree (number of bounding edges) and vertices with arbitrary valence (number of incident faces). For simplicity, quad faces are assumed. A "dart" vertex is a vertex that has an incident crease edge. A vertex that has two incident creases can be either a smooth crease or a corner. A vertex with more than two incident creases is a corner. Corner creases partition the neighborhood into convex or concave sectors. A sector is independent of the topology of another sector. Similarly, each side of a crease can be treated independently. Subdivision quadric-sects each face and assigns point position to each vertex as shown in Figure 1.

The following refinement scheme is used to recursively define Catmull-Clark surfaces:

- Start with a mesh of an arbitrary polyhedron. All the vertices in the mesh are called original points.
- Add a face point for each face.
- Set each face point to be the centroid of all original points for the respective face.
- Add an edge point for each edge.
- Set each edge point to be the average of the two neighboring face points and its two original endpoints.
- For each face point, add an edge for every edge of the face, connecting the face point to each edge point.
- For each original point P, take the average F of all n face points for faces touching P, and take the average R of all n edge midpoints for edges touching P, where each edge midpoint is the average of its two endpoint vertices. Move each original point to the point $\frac{F+2R+(n-3)P}{n}$. The new mesh will consist only of quadrilaterals, which will not be in general planar.

  In addition, the new mesh will generally look smoother than the old mesh. Repeated subdivision results in smoother meshes.

B. Rapid Evaluation of Catmull-Clark Subdivision Surfaces

Bolz and Schroder observe in [1] that the subdivision process is linear in nature. And thus, the final surface can be considered as a linear combination of basis functions with the original control points as weights as shown in Equation1.

$$s(u, v) = \sum_i B^i(u, v)p_i \tag{1}$$

where p_i are the control points that carry (x,y,z) positions. p_i might also carry texture coordinates and colors. B^i are the basis functions and they only depend on the connectivity of the mesh and the presence of tags.

In the following, we elaborate on the Rapid Evaluation of Catmull-Clark Subdivision Surfaces algorithm implementation and generation of basis function and control points.

1) Control Points and Basis Function Generation: The implementation of the Rapid Evaluation of Catmull-Clark Subdivision Surfaces algorithm constitutes of two main components: (*i*) Control point gathering for each face and (*ii*) Basis function tables generation.

The process of gathering control points for a specific face depicted in Figure 2 can be summarized as follows:

- Start with the point next to the extraordinary vertex in the face.
- Add the vertices in the one ring neighborhood of the extraordinary in an anticlockwise order.
- Add vertices of type one; these vertices will be at the following indices: 2*valence + 1, 2*valence + 2, 2*valence + 3.
- Add vertices of type two; these vertices will be at the following indices: 2*valence + 4, 2*valence + 5, 2*valence + 6, and 2*valence + 7.

Basis Functions tables are then pre-computed by generating base meshes that include only one basis function in a certain coordinate, and subdividing those base meshes using an existing recursive implementation.

2) Algorithm overview: Once the control points have been gathered for each face and the basis function tables are generated, we can proceed with the last step of the RECCSS algorithm which requires heavily computation resources and constitutes the main bottleneck for the algorithm execution time.

Each limit patch is evaluated uniformly to a user specified depth from the control points using pre-computed arrays that contain basis function tables. An initial subdivision step is performed to get to a situation where each first level quad contains at most one irregular vertex. This results in the basis function being a function only of the valence of the irregular vertex of the patch. According to [1], Pseudo-code for the algorithm is as follows:

```
// N = number of control points in 1-ring
// C = number of channels
float sample[C][33*33];
float bases[N][33*33];
float control[N][C];
for( k = 0; k < C; ++k )
 for( j = 0; j < N; ++j )
  for( i = 0; i < 33*33; ++i )
   sample[k][i] += bases[j][i]*control[j][k]
```

We observe that the loops in the pseudo-code are totally independent; and thus, the internal loop can be easily parallelized since the evaluation of one patch has no effect on the evaluation of any other patch. This observation will be harvested in Section III to allow for parallel computation of RECCSS on the available architectures.

III. Parallelization of RECCSS

A. Open Multi-Processing

Open Multi-Processing (OpenMP) [22] is an Application Programming Interface (API) that supports multi-platform shared memory multiprocessing programming. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI).

B. Amdahl's law

Speedup is defined as the time it takes a program to execute in serial (with one processor) divided by the time it takes to execute in parallel (with more than one processor).

The formula for speedup is shown in Equation 2:

$$S = \frac{T(1)}{T(j)} \tag{2}$$

where $T(j)$ is the time it takes to execute the program when using j processors.

Several factors show up as overhead for parallel execution which can limit the achieved speedup, notably:

1) Time when one or more of the processors used is idle.
2) Extra computations needed for parallel execution.
3) Communication overhead between processors.

Amdahl's Law [23] formulates the potential speedup of program using multiple processors as shown in Equation 3.

$$Speedup = \frac{1}{(1-f)+\frac{f}{N}} \tag{3}$$

where f is the fraction of code infinitely parallelizable with no scheduling overhead, $(1-f)$ is the fraction of code inherently serial and N is the number of processors.

We note that:

- When f is small, parallel processing has little effect. For $f = 0$, $Speedup = 1$, i.e. the time required to execute the program on a single processor is equal to the time required to executed it on N processors.
- When $f = 1$, $Speedup = N$. This is maximum achievable speedup.

C. Performance Evaluation

We note in Section II-B2 that the bulk of the RECCSS algorithm can be parallelized using simple tweaking. We claim that this parallelization would result in great enhancements in term of speedup by leveraging the capabilities of the available state of the art architectures. For this aim, we first implement RECCSS in C++ using Visual Studio 10, execute it for 100 different runs and collect the average execution time. We refer to this as **RECCSS** in the rest of this paper. Second, by following the directions discussed in Section III-A, we alter the RECCSS algorithm implementation to allow for parallel execution to take place. We refer to this as Parallelized RECCSS (**PRECCSS**).

For the sake of performance evaluation of the parallelization of RECCSS, we use the following three machines:

- Machine I: 1.66 GHz Intel Core Duo, 1 GB Memory.
- Machine II: 2.66 GHz Intel Quad Core, 2 GB Memory.
- Machine III: 3 GHz Intel Core i7-5500U, 16 GB Memory.

1) Machine I: Figure 3 shows in red RECCSS reported execution time (1 process, 1 thread) on Machine I. The average run time is 4839.78 ms.

Figure 3 shows in blue PRECCSS measured execution time (1 process, 2 threads) on Machine I. The average run time is 2467.16 ms. Speedup achieved is 1.96 which is a great improvement for an algorithm used in real-time by commercial applications.

Machine I uses Intel Core Duo which has two cores. The ideal speedup that can be achieved according to Amdahl's law is 2. The 1.96 reported speedup indicates that the applications is highly parallelizable.

2) Machine II: Figure 4 shows in red RECCSS reported execution time (1 process, 1 thread) on Machine II. The average run time is 2108.46 ms.

Figure 4 shows in blue PRECCSS measured execution time (1 process, 4 threads) on Machine II. The average run time is 564.76 ms. Speedup achieved is again very noticeable of value equal to 3.73.

Machine II uses Intel Quad Core which has four cores. The ideal speedup that can be achieved according to Amdahl's law

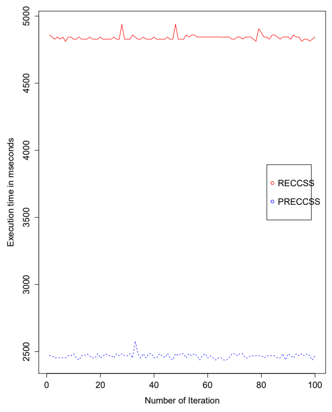


Fig. 3. RECCSS and PRECCSS execution time on Machine I.

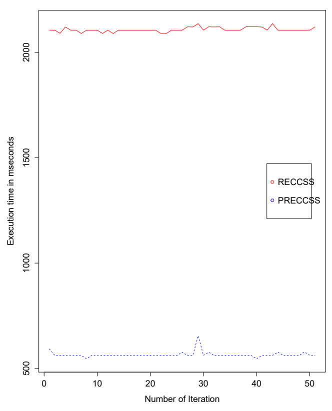


Fig. 4. RECCSS and PRECCSS execution time on Machine II.

is 4. The 3.73 reported speedup is another proof about the highly parallelizable characteristic of the application.

Table I reports RECCSS and PRECCSS average run time in milliseconds and the achieved speedup using PRECCSS for machines I and II.

3) Machine III: We repeat the same scenario using Machine III with:

- Hyper-threading disabled.
- Hyper-threading enabled.

Intel Hyper-Threading Technology (HTT) is a form of Simultaneous Multi-Threading (SMT) technology [24]. SMT is a pipeline design and implementation which allows more than one hardware threads to execute simultaneously within each core and share its resources. Each thread can be individually halted or interrupted, independently from the other threads sharing the same physical core. Intel HTT can be enabled or disabled.

With hyper-threading disabled, we witness a speedup factor of 1.85. When hyper-threading is enabled, we obtain a speedup factor of 2.14.

Machine III uses Intel Core i7-5500U Processor which has two cores and four threads, with Intel Hyper-Threading Technology. The ideal speedup that can be achieved according to Amdahl's law, when hyper-threading is disabled, is 2. Amdahl's law does not apply for hyper-threaded multicore processors [25].

We study in the following section the predictability of PRECCSS load instructions.

IV. Load Values Predictability

Cache miss occurs when a program accesses a memory location that is not in the cache. The processor has to wait for the data to be fetched from the next memory location or from main memory; thus, cache misses directly impact the processor performance. The miss ratio is the fraction of accesses which are a miss. The cache miss ratio of an application depends on the size and organization of the cache. In general, the larger the cache size, the smaller the miss ratio. The performance impact of a cache miss depends on the latency of fetching the data from the next cache level or main memory.

Catmull-Clark Subdivision Surface (CCSS) implementations do not perform well and suffer from low CPU utilization because it is often waiting for data to be transferred from memory due to the repeated pointer indirections. Rapid Evaluation of CCSS intend to achieve better performance by carefully addressing caching issues. However, this does not eliminate cache misses completely.

Pin [19] is a dynamic binary instrumentation tool. Instrumentation is a technique that inserts code into a program to collect run-time information. *Pin* does not need the source code, and does not require recompilation and post-linking. It provides rich APIs that allow the user to write instrumentation tools called pintools in C, C++ or assembly.

We use *Pin* to characterize cache performance of RECCSS. We run RECCSS with a pintool that models the following cache hierarchy:

- First level instruction cache: 32 kB, 32 B lines, 32-way associative.
- First level data cache: 32 kB, 32 B lines, 32-way associative.
- Second level unified cache: 2 MB, 64 B lines, direct mapped.
- Third level unified cache: 16 MB, 64 B lines, direct mapped.

We show in Table II load and store instructions hits, misses and miss rates for all cache levels. The reported load miss rates are 0.91%, 50.13%, and 100.00% for L1 data cache, L2 unified cache and L3 unified cache respectively. Also, we measure 15.63%, 0.66%, and 100.00% store miss rates for L1 data cache, L2 unified cache and L3 unified cache respectively. The obtained miss rates reflect the fact that RECCSS suffers from considerable cache misses, which negatively impact its performance. Data value prediction, described in the following section, is a technique intended to overcome cache misses issue.

	Machine I	Machine II
RECCSS Average Run Time	4840	2108
PRECCSS Average Run Time	2467	565
Speedup using PRECCSS	1.96	3.73

TABLE I
AVERAGE RUN TIME OF RECCSS AND PRECCSS IN MSECONDS, AND THE SPEEDUP ACHIEVED ON MACHINES I AND II.

	L1 Instruction Cache	L1 Data Cache	L2 Unified Cache	L3 Unified Cache
Load Hits	137594	21755	384	0
Load Misses	570	200	386	386
Load Miss Rate	0.41%	0.91%	50.13%	100.00%
Store Hits	0	78504	14446	0
Store Misses	0	14542	96	96
Store Miss Rate	NA	15.63%	0.66%	100.00%

TABLE II
STATISTICS OF CACHE ACCESSES AND PERFORMANCE IN RECCSS.

A. Data Value Prediction

Long latency instructions can be either of variable latency (e.g. LD, DIV, and SQRT) or fixed latency (e.g. MUL and FP ADD). Data dependent instructions will stall behind long latency instructions, potentially creating one or more critical paths in the program. This may cause instruction buffers to fill up, which results in stalling the instruction fetch unit and hence, preventing new instructions from flowing into the pipeline. Data value prediction (DVP) is a technique to increase instruction level parallelism by attempting to overcome serialization constraints caused by true data dependences. By predicting instruction results early and passing them to dependent instructions, DVP reduces the length of critical paths through a program [11] [21] [20]. When a predicted instruction executes, the correct result is compared with the predicted one. If a mismatch occurs, the correct result is passed to the dependent instructions, which execute again with the correct result as input [12]. If the benefit from increased parallelism outweighs the misprediction recovery penalty, overall performance could be improved. Enhancing performance with value prediction therefore requires highly accurate prediction methods.

Prediction mechanisms can be either computational or context based [11]. A computational predictor yields a predicted next value by performing an operation on previous values such as *last value* (LV) predictor [15] and *stride value* (STR) predictor [14]. Last value predictor performs the identity operation on the previous value. In a stride sequence, an element is obtained from the previous one by adding to it a constant delta called stride. A *2-delta stride* predictor keeps track of two stride values (s_1 and s_2). S_1 and the last value are used to compute a prediction of the next value. The newly computed stride is stored in s_2. S_1 is only updated with s_2 when s_2 occurs twice in a row [16].

A context is a finite ordered sequence of values. A context based predictor makes value prediction based on its observation of previous patterns. An order k *Finite Context Method* (FCM) predictor uses k preceding values [13]. A hybrid predictor combines distinct types of predictors, such as (STR, LV) [17] and (STR, FCM) [18] hybrid predictors.

B. Predictability of CCSS Algorithm Load Instructions

We show in this Section the predictability of the load instructions in the parallelized Rapid Evaluation of Catmull-Clark Subdivision Surfaces algorithm. We collect data using a machine that has 2.9 GHz Intel Core i7-3520M Processor. The processor has 2 physical cores, 4 threads, 128 KB L1 cache, 512 KB L2 cache, and 4 MB L3 cache.

We use *Pin* to implement a tool that predicts all load instructions using *LV*, *FCM*, *2-delta stride* and hybrid *(LV, FCM)* predictors. We employ the tool to study the predictability of the load instructions in the parallelized rapid evaluation of Catmull-Clark subdivision surfaces algorithm.

Table III shows the total number of static and dynamic instructions, the total number of static and dynamic load instructions, and the percentage of load instructions with respect to the total number of instructions in RECCSS and PRECCSS. For RECCSS and PRECCSS, static load instructions account respectively for 5.366% and 12.72% of the total number of static instructions, while dynamic load instructions account respectively for 5.77% and 5.78% of the total number of dynamically executed instructions.

We use the following metric in measuring the predictability of load instructions:

Prediction accuracy = total number of successful loads predictions / total number of attempted loads predictions.

Figure 5 shows the result of the predictability of the load instructions using *LV*, *FCM*, *2-delta stride* and hybrid *(LV, FCM)* predictors. *LV*, *FCM*, *2-delta stride* and *hybrid(LV, FCM)* achieve 61.95%, 92.2%, 49.03% and 93.25% prediction accuracy respectively.

We can conclude from the high prediction accuracy obtained using FCM and hybrid(LV, FCM) predictors, in addition to the high load miss rates for L2 and L3 unified caches reflected in Table II, that significant performance improvement can potentially be obtained using value prediction.

	Static Instructions	Static Load Instructions	Dynamic Instructions	Dynamic Load Instructions	%Static Load / Static Instructions	%Dynamic Load/ Dynamic Instructions
RECCSS	62964	3379	138164	7984	5.366	5.77
PRECCSS	24891	3167	138167	7984	12.72	5.78

TABLE III
COUNT OF STATIC AND DYNAMIC INSTRUCTIONS IN RECCSS AND PRECCSS.

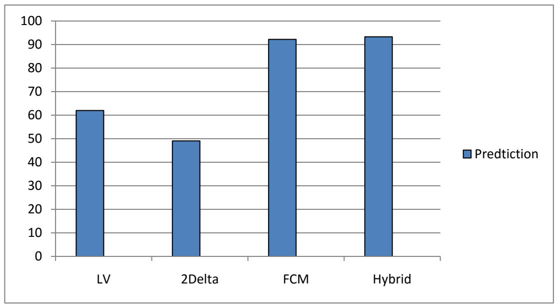


Fig. 5. Predictability of load instructions for PRECCSS.

V. CONCLUSION

Recursive subdivision is a powerful and useful technique in surface modeling. Their main advantages is to overcome the restriction of using rectangular meshes imposed by the traditional B-spline surface modeling approach. A subdivision surface is defined by a polyhedral control mesh where the vertices could be n-valent, and a set of rules for refining this mesh. By applying these rules to the generated mesh, a limit surface is generated which could be multi-sided. For these reasons and others, it had been integrated in most standard modeling packages. Among the set of rules, those developed for the Catmull-Clark surfaces (CCSS) are the most popular, as they present natural extension of the tensor product cubic B-spline. Rapid Evaluation of CCSS (RECCSS) focuses on achieving maximum performance on standard graphics hardware together with a modern, general purpose CPU, by carefully addressing caching issues. We implement RECCSS, run it sequentially on three different machines and monitor the needed computation time. We then introduce Parallelized RECCSS (PRECCSS) to allow parallel computation and analyze the achieved performance where significant speedup is observed. In addition, we study the predictability of PRECCSS load instructions and show that they are highly predictable.

As for future work, we intend to apply similar techniques to other types of subdivision surfaces, and to incorporate various interpolation constraints such as those mentioned in [26].

REFERENCES

[1] J. Bolz and P. Schroder. Rapid Evaluation of Catmull-Clark Subdivision Surfaces. *ACM Special Interest Group on Computer Graphics and Interactive Techniques*, 2002.

[2] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350-355, 1978.

[3] S. Havemann. Interactive Rendering of Catmull/Clark Surfaces with Crease Edges. Tech. Rep. TUBSCG-2001-01, TU Braunschweig, 2001.

[4] S. Junkins. Fast Triangle Neighbor Finding for Subdivision Surfaces. Tech. Rep. Intel Architecture Labs, September 1999.

[5] D. Zorin, P. Schroder and W. Sewldens. Interactive Multiresolution Mesh Editing. Proceedings of SIGGRAPH 97 (1997), 259-268.

[6] D. Zorin, P. Schroder and W. Sewldens. Subdivision for Modeling and Animation. Course Notes. ACM Siggraph, 2000.

[7] J. Stam. Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values. Proceedings of SIGGRAPH 98 (1998), 395-404.

[8] D. Zorin and D. Kristjansson. Evaluation of Piecewise Smooth Subdivision Surfaces. Visual Computer (2002).

[9] S. Bischoff, L. Kobbelt and H. Seidel. Towards Hardware Implementation Of Loop Subdivision. 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware (2000), 41-50.

[10] T. Derose, M. Kass and T. Truong. Subdivision Surfaces in Character Animation. Proceedings of SIGGRAPH 98 (1998), 85-94.

[11] Y. Sazeides and J. E. Smith., The predictability of Data Values. Micro-30, December, 1997.

[12] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit via Value Prediction. MICRO'29, 226–237, December, 1996.

[13] Y. Sazeides and J.E. Smith. Implementations of Context-Based Value Predictors. ECE-TR-97-8, University of Wisconsin-Madison, 1997.

[14] Y. Sazeides, S. Vassiliadis and J.E. Smith. The Performance Potential of Data Dependence Speculation and Collapsing. Proc. of the 29th. Int. Symp on Microarchitecture, December, 1996.

[15] M.H. Lipasti, C.B. Wilkerson and J.P. Shen. Value Locality and Load Value Prediction. Proc. of the 7th. Conf. on Architectural Support for Programming Languages and Operating Systems, 138–147, October, 1996.

[16] R. J. Eickemeyer and S. Vassiliadis. A load instruction unit for pipelined processors. IBM Journal of Research and Development, 37(4),547-564, July, 1993.

[17] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, 281–290, 1997.

[18] B. Rychlik, J. Faistl, B. Krug and J. P. Shen. Efficacy and Performance Impact of Value Prediction. International Conference on Parallel Architectures and Compiler, October, 1998.

[19] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. PLDI '05, 2005.

[20] W. J. Ghandour, H. Akkary and W. Masri. Leveraging Strength-Based Dynamic Information Flow Analysis to Enhance Data Value Prediction. ACM Trans. Archit. Code Optim., 9(1), March 2012.

[21] W. J. Ghandour, H. Akkary and W. Masri. The Potential of Using Dynamic Information Flow Analysis in Data Value Prediction. The Nineteenth International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2010.

[22] The OpenMP API specification for parallel programming. http://openmp.org/wp/.

[23] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67, 483–485.

[24] D. Tullsen, S. Eggers and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. ISCA 1995.

[25] H. Che and M. Nguyen. Amdahl's law for multithreaded multicore processors. J. Parallel Distrib. Comput. 74 (2014) 3056–3069.

[26] A. Nasri and M. Sabin. Taxonomy of interpolation constraints on recursive subdivision surfaces. Visual Comp 18, 382–403 (2002).