PBench: A Benchmark Suite for Characterizing 3D Printing Prefabrication

Fan Yang†, Feng Lin†, Chen Song†, Chi Zhou ‡, Zhanpeng Jin§, Wenyao Xu†
† Computer Science and Engineering, University at Buffalo (SUNY), New York 14260
‡ Industrial and System Engineering, University at Buffalo (SUNY), New York 14214
§ Electrical and Computer Engineering, Binghamton University (SUNY), New York 13902
Email: {fyang24, flin28, csong5, chizhou, wenyaoxu}@buffalo.edu, zjin@binghamton.edu

***Abstract*—3D printing is revolutionizing the next-generation manufacturing industry. With increasing design complexity, computation in the prefabrication process is becoming the bottleneck of 3D printing. For example, a multi-scale, multi-material 3D design (e.g., a bionic bone) takes a few hours or even days to complete the prefabrication computation. Therefore, it is prudent to improve the performance of 3D printing prefabrication. In this paper, we investigate the computational challenges in 3D printing. First, we develop the *PBench*, an open-source standard benchmark suite for 3D printing prefabrication. To the best of our knowledge, this is the first benchmark suite for 3D printing prefabrication. Second, we study the properties of PBench using *TotalChar*, a proposed benchmark characterization framework analyzing PBench from three complementary dimensions, i.e., (1) microarchitecture independent analysis, (2) architecture bottleneck analysis and (3) functional performance analysis. Experiments show that 3D printing prefabrication can be potentially optimized through specified function accelerator design or parallelism exploration. Overall, this work establishes the potential for accelerating 3D printing prefabrication.**

Keywords: 3D printing, Prefabrication algorithms, Benchmark.

I. Introduction

3D printing, also known as additive manufacturing, refers to a process which builds a three-dimensional (3D) object layer by layer from digital data [11]. As an advanced manufacturing technique, 3D printing holds the merit of affordability and customizability. It has been changing the market trends, and resulting in an efficient, responsive, robust and sustainable production paradigm in a wide range of applications including aerospace, automobile, defense, biomedical, health and energy industries [20].

There are two steps in the standard 3D printing process. The process input usually is a 3D design model, which is usually designated by 3D design tools (e.g., Solidworks) or produced by a 3D scanner. The 3D design is represented as a triangle mesh in the STereoLithography (STL) format. There are two main steps in the fabrication process. The first step is a computing process that transforms a 3D design file into a set of instruction codes, G-code, which a 3D printer can execute. This process for a 3D printer is similar to a compiler for a CPU, or a synthesizer for an Field-Programmable Gate Array (FPGA). This step is also known as $prefabrication$ or process planning [16]. The second step is the manufacturing process, in which the 3D printer fabricates the physical object according to the instruction codes.

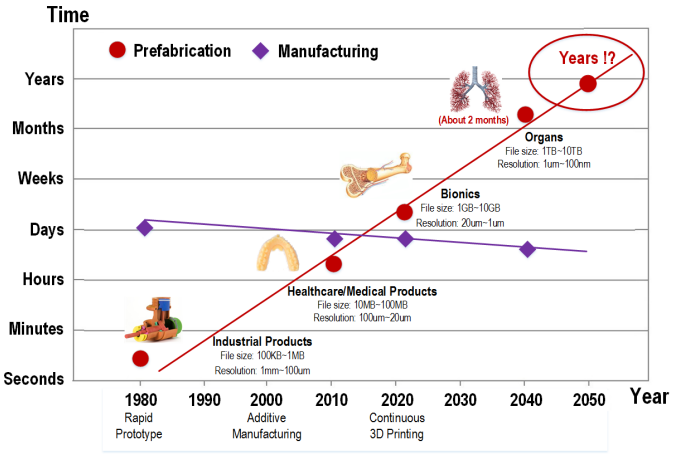

Fig. 1. The process time changes as 3D printing technology advances.

The primary fabrication time in 3D printing contains two parts, i.e., computing time in prefabrication and processing time in manufacturing. Our preliminary study has surveyed the fabrication time evolution of 3D printing in the past three decades [24], [32], [9]. Figure 1 depicts our survey results, indicating the tremendous changes in terms of fabrication resolution, design complexity and application domains. The advances in material, process and machine development has moved the 3D printing process from rapid prototyping to rapid manufacturing with significantly improved manufacturing speed and production throughput [2]. The recent breakthrough continuous 3D printing technology decreases the manufacturing time by orders of magnitudes [24]. However, the unprecedented design complexity introduced by the expanded design freedom poses a significant challenge in terms of prefabrication computation [8]. Nowadays, prefabrication time is increasingly significant and becomes the dominating factor in the entire 3D printing process. According to our literature review, there are few in-depth studies focusing on the acceleration of 3D Printing prefabrication.

The purpose of this study is to introduce the 3D printing computation challenge within the computer society. The first step is to understand the computational characteristics in 3D printing prefabrication. To this end, we develop the *PBench*, an open-source standard benchmark suite of 3D printing prefab-

rication. PBench contains a set of prefabrication algorithms, including slicing, path planning and support generation in 3D printing, and a group of 3D design STL files. Specifically, both coverage and representativeness are considered in the benchmark design, which establishes a baseline in this application.

To comprehensively understand the PBench characteristic and provide intuition for acceleration in hardware design, we propose *TotalChar*, an end-to-end performance characterization framework. TotalChar has three analytic dimensions: (1) microarchitecture independent analysis, (2) architecture bottleneck analysis and (3) functional performance analysis. Microarchitecture independent analysis provides the fundamental characteristics of 3D printing prefabrication, including computing, memory and control behaviors. Architecture bottleneck analysis is to analyze the efficiency at the application level. We find that an average 66.24% of the CPU pipeline slots are used for useful computation, and only 3.74% slots are dedicated to resolving branch misprediction. This result indicates that 3D printing prefabrication can be potentially optimized through specified function accelerator design or parallelism exploration. In functional performance analysis, we divide the PBench benchmarks into multiple common function blocks. Experiments implies that the intersection and distance computing blocks are dominant computational bottlenecks.

The contribution of this work are summarized as follows:

- We develop the *PBench*, a benchmark suite for the 3D printing prefabrication. It encompasses *six* prefabrication algorithms and *six* 3D design sets.
- We propose a comprehensive characterization framework, *TotalChar*, which evaluates PBench from three dimensions, including the invariant property, architecture character and functional behavior. TotalChar can provide comprehensive evaluation of a benchmark and potential methods for hardware improvement.
- We analyze the characteristics of 3D printing manufacturing, discussing the pitfalls, challenges and direction of accelerating 3D printing prefabrication.

II. Background and Motivation

In the section, we introduce the background of 3D printing prefabrication and highlight the motivation of this work.

A. Acceleration of 3D Printing Prefabrication

Prefabrication is a mandatory step in the 3D printing process, and its research has a long history since the birth of additive manufacturing technologies. In recent years, a few studies have emerged on designing time-efficient prefabrication algorithms in the 3D printing process. For example, Vatani *et al.* enhanced the *slicing* algorithm using nearest distance analysis [26]. Muller *et al.* presented a method to accelerate *slicing* in prefabrication through downsampling of design resolution [18]. Gregori *et al.* proposed an asymptotically optimal algorithm for *slicing* [12]. Zhou *et al.* developed an efficient *path planning* algorithm to decrease the overhead in prefabrication [34]. These proposed methods all claims that they can improve the time-efficiency of 3D printing prefabrication. However, in demonstrating performance of their algorithms, these studies employed different 3D design input files. There is a lack of a common standard benchmark to compare the work with a baseline. In addition, all these works study acceleration of 3D printing fabrication from the algorithm perspective. There is no study to address this challenge through the design of advanced architecture and hardware.

B. Benchmark Characterization

In recent years, domain-specific acceleration has become a research trend in computer science [7]. A few studies have investigated benchmark design and characterization on specific applications [19], [14], [23], [28]. For instance, Reagen *et al.* proposed MachSuite for accelerator design and customized architectures [19]. This work analyzes the microarchitecture-dependent character of MachSuite, but it lacks direct insight on architectural optimization. Kanev *et al.* profiled the character of a warehouse-scale computer [14]. This work presents the detailed architecture information of a warehouse-scale computer and analyzes the potential methods to improve a specific architecture. However, this method cannot be generously applied to other benchmark designs because it only focuses on the optimization strategy towards a special type of architecture. Therefore, to provide insights on both the invariant characters and the methods for improving architectures, it needs a comprehensive benchmark characterization framework.

III. Benchmark Suite Design

In this study, the benchmark suite consists of two parts. One is a set of prefabrication algorithms, and the other is a group of appropriate input datasets of 3D design. In 3D printing prefabrication, the input data sets are in the STL format. In the remaining part of this section, we will discuss in detail the design of the benchmark suite and input testbench.

A. Prefabrication Algorithm Selection

Design Space: 3D printing prefabrication contains $slicing$, $path\ planning$, $support\ generation$, $orientation$, $repairing$, $packaging$ etc. $Slicing$ is a computing process converting a 3D design model to a set of 2D planar layers. It intersects all the triangles with each slice plane and connects the resulting segments with a group of properly oriented closed contours. $Path\ planning$ transforms a 2D sliced contour into 1D paths, which directs the printing tools to form the desired shape. $Support\ generation$ adds physical support to the regions that need to form the shape during the printing process. $Orientation$ changes the direction of the 3D object to form a more robust and concrete printing result. $Repairing$ corrects the input data if they are not complete or have fault. $Packaging$ wraps up multiple 3D design files when multiple objects are manufactured. PBench's design is based on two objectives: coverage and representativeness.

Rational Design: We design PBench with respect to tow objectives, i.e., coverage and representativeness. Coverage requires including main computing patterns in 3D printing prefabrication [19]. In general, there are two types of processes

in 3D printing prefabrication workloads: basic operations and auxiliary operations. Basic operations include $slicing$ and $path\ planning$, which are required in all 3D printing processes. Auxiliary operations include $support\ generation$, $orientation$, $repairing$, $packaging$, etc., which are required for the specific printing object. In this study, PBench is designed to contain all the basic operations, as well as the most frequently used stage in auxiliary operations, $support\ generation$ [5], [25], [10]. In total, PBench contains three types of operations, $slicing$, $path\ planning$ and $support\ generation$. Representativeness requires the chosen benchmark implementation to be able to substitute other similar workloads [3]. Therefore, we choose mainstream implementation [31], [34], [24], [4] of three operations in PBench. Specifically, PBench includes three types of slicing algorithm implementation, two types of path planning algorithm implementation and one support generation algorithm implementation. The details are described in Section 4.

B. 3D Design Testbench

The other part of the PBench design is to include a group of 3D design files (in STL format) to test the prefabrication algorithms. These design testbenches are important because different inputs can lead to significantly diverse computing behaviors and demands. According to our literature review, many studies use existing design testbenches. For the sake of efficiency, we use a in total six testbenches in this study.

The inputs are selected according to the following procedure: First, we collect a diverse existing 3D testbench as an input STL pool. To form the STL pool, we search through the internet for open-source STL files and develop a base STL pool with 65 different STL files [15]. Second, we organize the input STL pool with selection criteria. In this study, the criteria are coverage, diversity and representativeness. The chosen STLs should be able to represent the diversity of the entire STL pool. We achieve this by constructing a two-dimension matrix for each STL, one representing the size and the other indicating the domain. Finally, we select these criteria to choose a predefined number of STL files. The clustering strategy is adopted in the selection process. The implementation and selected results will be elaborated on the next section.

IV. Benchmark Description

In this section, we describe the PBench implementation in detail through the design approach in Section 3. Specifically, in this study we implement the prefabrication algorithm as single thread benchmarks considering it is the most commonly used case in practice. For 3D design testbench implementation, the K-means algorithm [13] is employed to categorize these STL files, and the chosen STLs are the centers of each group. We execute K-means multiple times and choose the groups with highest frequency.

A. Algorithm Description

Closest distance slicing (cds): $Slicing$ cuts a 3D model layer by layer and generates the 2D contour information [26]. The input for $slicing$ is $STereoLithography$ (STL). STL contains triangle mesh that describes the 3D object [30]. The output of $slicing$ is a $Common\ Layer\ Interface$ (CLI) file, which depicts the contour information of each layer [30]. $Closest\ distance\ slicing$ is the most fundamental and popular algorithm for slicing [31], [26], [17]. This algorithm iterates through all layers by computing the intersection segments of STL with the current layer, which forms an intersection queue. Then these segments are connected to contours using the closest distance search through the intersection queue. The core computation of this algorithm is the comparison with all other segments to find the neighbor segment.

Marching slicing (ms): $Slicing$ can also be computed through marching, which comprises geometry information to shorten the time for the neighboring search [21], [22]. It is implemented by first constructing a half-edge data structure (HEDS) to represent the geometry information of STL. Similar to the $closest\ distance\ slicing$, it also slices the STL layer by layer and connects these intersection segments to closed contours. The difference is the method for finding the neighbor segment. Instead of comparing through the intersection queue, $marching\ slicing$ checkes HEDS and locates the neighbor segment.

Sampling-based slicing (sbs): $Sampling\text{-}based\ slicing$ is especially designed for continuous printing, which is an evolving technique in 3D printing that can print one entire layer at a time [24], [33], [6]. Most 3D printers need to know the location of points as well as the movement of printing heads for manufacturing. This specific printer for continuous printing only requires the information of points. $Sampling\text{-}based\ slicing$ reads the data from the STL file. First, this algorithm rasterizes the data, changing the format from triangle mesh to point cloud. The X and Y coordinates of each point are defined by pixel centers (position). Second, this algorithm converts the sampling point cloud to the image profile (point location). It is achieved by sorting all the sampling points according to Z-coordinates. For each position, all the sampling points below the current slicing plane and on or above the previous slicing plane are considered and operate on the pixel matrix for previous layers according to the norm of each point. If the facing down position, we enter the object, and there should be a point at this position. If facing up, we leave the object and there should be no point at this location. Using this method we can produce the point location of each layer.

Rasterization path planning (rpp): $Path\ planning$ specifies the movement of the printing head based on the contours computed by $slicing$ [16], [34]. The input for $path\ planning$ is the CLI file, and the outputs are the points indicating the moving location of the 3D printer head. One common method to implement $path\ planning$ is $rasterization$. It applies precomputed contours to a fine-grained mesh and computes the intersections, and then connects the intersecting points to fill the inside of contours. The coordinates of these connecting lines represent the position of the printer head.

Contour offsetting path planning (copp): Another method to implement $path\ planning$ is contour offsetting [27], [4].

Fig. 2. Design testbench profile. From left to right, these images are cylinder, club, bull, ring, cam and impeller, respectively.

TABLE I
DESIGN TESTBENCH DESCRIPTION.

Name	Size	Triangle	Layer	Description
Cylinder	54 KB	1088	100	Simple geometry
Club	165 KB	3290	20	Daily supply
Bull	620 KB	12400	26	Animal
Ring	1.2 MB	23254	204	Daily supply
Cam	4.3 MB	86360	183	Daily supply
Impeller	28.5 MB	570982	175	Industry part

Contours are constructed by segments. Moving all segments of a contour inward a certain distance will result in a smaller contour of the same shape. The $Contour\ offsetting$ method repeats these steps until reaching the inner contour, or all the regions constrained by the contour have been filled up. These offsetting contours indicate the locations of the printer head.

Support generation (sg): $Support\ generation$ computes the positions requiring support during the manufacturing process [5]. The input is the CLI file, and the output is a set of points showing the support position. $Support\ generation$ comprises three steps. First, calculating the self-support regions, then computing the regions requiring support for each layer, and finally the complete support structure of the 3D object is computed. In PBench, it is implemented by first augmenting the contours of the current layer a certain range representing self-support, then computing the difference between the current and the previous layer as the regions requiring support. Finally adding support from the top to the bottom to derive the complete support structure.

B. Testbench Description

The selected testbenches are shown in Figure 2, and characterized in Table I. It can be observed that the chosen design of the testbenches range from kilobyte to megabyte in size, as well as cover four different domains. These testbenches meet the requirements of coverage, diversity and representativeness. $Cylinder$ is a simple geometry product, the height of which is 10 centimeters. $Club$ is a daily tool model. It is relatively small in size, and the height is 2 centimeters. $Bull$ is an animal design model. The size of $bull$ is 2.6 centimeters. $Ring$ and Cam are two kinds of ring knots, and cam is relatively more complex in structure. Their heights are 20.4 and 18.3 centimeters, respectively. $Radial\ impeller$ is an industry part which is prevalent in industry engineering, and its height is 17.5 centimeters.

V. BENCHMARK CHARACTERIZATION

To explore the characteristics of PBench, and provide insights into improving hardware architecture for 3D printing prefabrication domain, we evaluate PBench using a new characterization method, TotalChar, which profiles benchmarks in three dimensions.

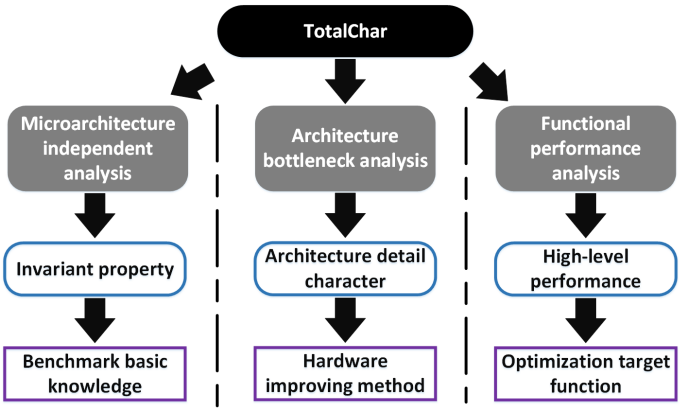

Fig. 3. TotalChar characterization method.

The diagram of TotalChar is shown in Figure 3. Each dimension of analysis in TotalChar have different focuses on the properties of the PBench. Microarchitecture independent analysis aims to seek the character which is invariant on different architectures; architecture bottleneck analysis inspects the architecture in detail to detect where the time is spent; functional performance analysis finds the performance of each function block. From each of these three analysis, we gain different insights for hardware improving. Microarchitecture independent analysis forms a solid base understanding of PBench; architecture bottleneck analysis detects which method is appropriate for enhancing performance; functional performance analysis reveals the target function block which plays the most significant role in system performance. In this paper, we first perform microarchitecture independent analysis, then architecture bottleneck analysis and finally the functional performance analysis.

A. Microarchitecture Independent Analysis

The microarchitecture independent characters of a benchmark depict the invariant properties that remains constant under different architectures. Analyzing these characters is not only a good first-order measurement that provides insight into the benchmark itself, but also forms a solid basis for future architecture bottleneck analysis. The matrix we use for analysis is focused on three main categories: compute, memory and control [23].

We evaluate these properties using gem5. The gem5 simulator is an open-source simulation platform for computer-system architecture research, encompassing system-level architecture as well as processor micro-architecture [1]. With gem5 we can trace the flow of every instruction as well as the movement of data, which provides the opportunity to inspect the properties of the benchmark. Because the target properties in this section are microarchitecture independent characters, it makes no difference no matter which CPU configuration we use. So we choose the default out-of-order CPU with X86 ISA in gem5.

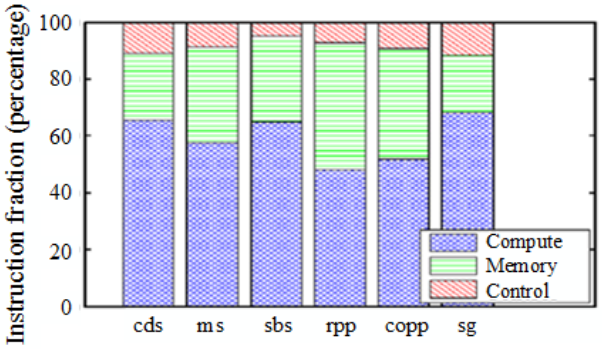

Fig. 4. Instruction mix of PBench.

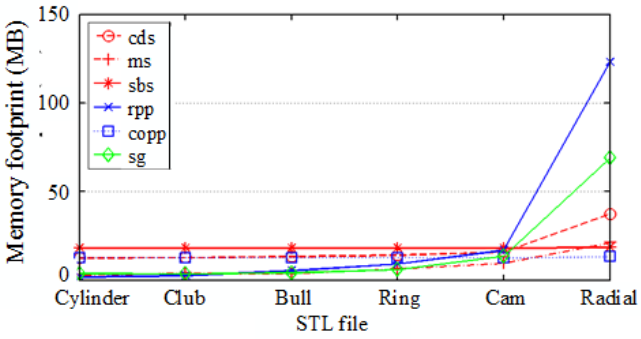

Fig. 5. Memory footprint of PBench.

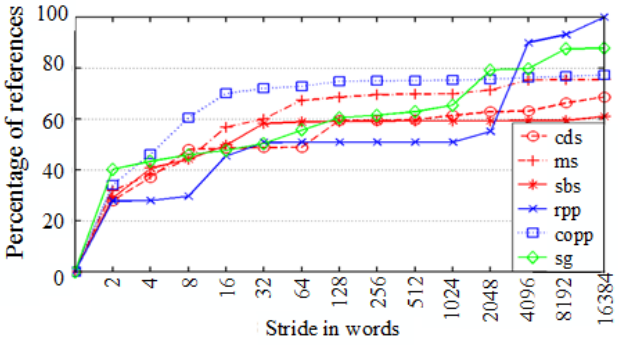

Fig. 6. Spatial locality of PBench.

1) Compute: The first category we want to study is the compute behaviors, which analyze the instruction sequences of programs and explore their execution patterns. In this category, we will focus on instruction mix which break down the executed instructions into compute, memory and control categories [23]. We average the instruction mix of the six input STL files for each benchmark in PBench. As shown in Figure 4, each bar represents the average instruction mix of each benchmark. In general, PBench encompasses a relatively high portion of compute instructions, and a low portion of control instructions. Specifically, the overall average percentage of compute, memory and control instructions of PBench are 59.51%, 31.81% and 8.68%, respectively. Since the control percentage is small, it is likely that PBench is highly predictable, which will be confirmed in the analysis of control behavior.

The difference of instruction mix between benchmarks indicates their difference in computing behavior. For example, *marching slicing* has a relatively larger percentage of memory instructions than *closest distance slicing*, specifically, 33.61% for *marching slicing* and 22.33% for *closest distance slicing*. This results from the fact that *marching slicing* needs to create a half edge data structure to store all the input data, but *closest distance slicing* does not. Referencing and searching through this large structure requires multiple memory access operations. This results in more memory operations for *marching slicing*.

2) Memory: The invariant memory property of a program has significant impact on architecture performance including cache miss and memory operation latency. Benchmarking the memory behavior will not only provide insight into the memory constrain of a benchmark itself, but also establish foundations for future architecture bottleneck analysis. We will discuss three memory metrics in this section, memory footprint, spatial locality and temporal locality.

a) Memory footprint: Memory footprint measures the total memory space a program requires for execution. We execute all PBench benchmarks with all input data sets, and the memory footprint is shown in Figure 5.

This figure shows the scalability of the memory footprint of PBench. Each line represents the memory footprint required for running this benchmark given the specific input STL file. Since the STL files are arranged according to the size, it can be observed that as the input data size increases, the memory required by each benchmark also increases. The average memory footprint required for executing the six PBench benchmarks for each STL file is 8.754, 9.046, 9.638, 11.144, 11.536 and 47.171 megabytes, respectively. Moreover, there is a turning point in this figure after which the memory footprint grows dramatically. This is because the memory footprint of a benchmark is determined by two factors. The first is the space required for processing the input data, and the second is the routine operations the program execute independent of the size of data. When the size of STL file is small, the second factor dominates the memory footprint, but as the size of STL file gets larger, the first factor becomes more important, and these result in the turning point in figure 5.

b) Spatial locality: Spatial locality measures how far away the benchmark's memory access occurs relative to the previous memory access. This matrix is important for hardware optimization because once we understand the strides between each memory access, we can design caches with lower miss rate. We show the spatial locality of PBench by averaging the execution results of the six STL files in PBench testbenches. The experimental result is indicated in Figure 6.

This figure shows the cumulative percentage of memory access of each benchmark. The X coordinate indicates the distance between each memory access in number of words (32

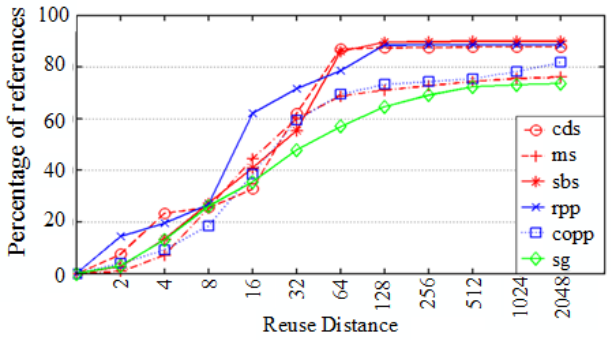

Fig. 7. Temporal locality of PBench.

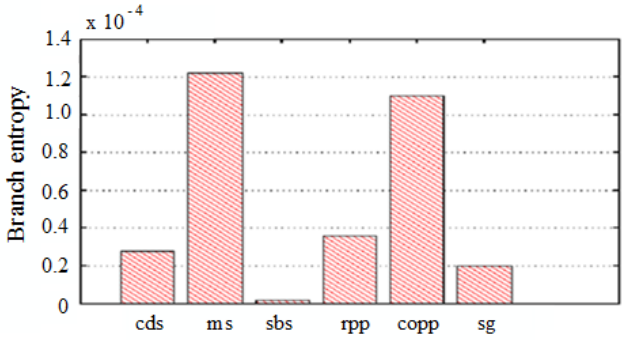

Fig. 8. Branch entropy of PBench.

bits). It can be observed that these benchmarks have an initial jump, then rising with different slopes. These are resulted from the multiple loops of these benchmarks. In the outer loop, the benchmarks traverse through the input data set with stride-one access pattern, which resulted in the initial jump. In the inner loop, these benchmarks operate on the data choosing from the outer loop, which caused the rising of spatial locality after the initial jump.

Take the benchmark of *closest distance slicing* for example. In the outer loop, it traverses through point arrays to choose the closest points. In the inner loop, it connects the closest point pairs to closed contours. The traversing in the outer loop caused the initial jump. In the inner loop, it operates on the choosing point pairs, and the memory location of where the choosing pairs are influences the stride of spatial locality. In our experience, the choosing pairs can be adjacent, or separating far away in the points array. This makes the memory stride can be any number from 1 to the size of the point array, which explains the continuous raising of spatial locality.

c) Temporal locality: Temporal locality measures the interval distance of memory accessing the same address. This is also a very important matrix for hardware architectures since it explores the degree of reusability. Typically, a benchmark with a high portion of memory access that have small reuse distance usually has a low miss rate. We perform temporal locality analysis on PBench and the result is shown in Figure 7.

Temporal locality presents an overall trend of ascending and saturating at some thresholds, which are also resulted from multiple loops in PBench. In the outer loops, these benchmarks will traverse through the data and select some target points for further operations. This operation has little re-usability. In the inner loops, these benchmarks perform operations on the selected data, which causes the data reuse. Using *rasterization path planning* as an example, in the outer loop, this algorithm traverses through the input contours to compute the intersection points with the rasterization nets. In the inner loop, it sorts and traverses these intersection points. The re-usability comes in the operations on these intersection points, and the size of the intersection points influences the reuse distance. In our experience, the size of the intersection points varies from several points to dozens of points, which causes the reuse distance continuously ascending from 2 to 128 as indicated in Figure 7.

Both spatial locality and temporal locality analysis provide the hint that PBench contains multiple loops, which are many fixed function blocks. Following this hint we observe interesting property in architecture bottleneck analysis and we show that there are opportunities for fixed function accelerator design for hardware improvement of PBench.

3) Control: A modern superscalar CPU always speculatively execute instructions to achieve higher performance, which requires predicting the control flow of a benchmark. The control flow measurement quantifies the degree of predictability of a program. We use branch history entropy to describe the predictability of PBench. This method was proposed by Yokota *et al* in 2008, which combines Shannon's entropy with branch prediction [29]. In general, branch predictions largely depend on the randomness of the pattern of branches. The more regularity of a serial of branch results, the more likely branch predictors produce the correct prediction result.

We measure the randomness of control patterns in every 10 consecutive branches, and the result is shown in Figure 8. Each bar represents the average entropy of executing six input testbenches of Pbench. It is observed that all benchmarks have a very low branch entropy. This confirms our assumption in the section of compute analysis that the predictability of PBench is high.

B. Architecture Bottleneck Analysis

Microarchitecture independent analysis forms a solid base understanding of PBench. Only utilizing this analysis, however, does not reveal architecture detail performance. In order to provide more knowledge on architecture properties and insights for architecture improvement, we perform architecture bottleneck analysis on PBench.

Architecture bottleneck analysis focuses on one specific CPU. Without loss of generality, we choose the most prevalent and representative CPU in PCs, Intel i7 processor. We run this experiment in real machine, and the detailed system information is shown in Table II. The method we use for inspecting architecture properties is Top-Down analysis. Top-Down analysis is a method developed by Yasin A in Intel

TABLE II
REAL MACHINE SYSTEM CONFIGURATION.

Component	Configuration
Processor	Intel Core I7-4790 CPU @ 3.60GHz
Memory	7.7 GB
Operating System	Ubuntu 14.04 LTS

which can quickly identify true bottlenecks in out-of-order processors [28]. It utilizes the counters in the hardware performance counter unit available in Intel processors to measure the performance of benchmark. The approach is to first categorize the execution time into different domains, and then drill down to the domains of interest. There are four domains in top level: Frontend Bound, Bad Speculation, Retiring and Backend Bound, which are as follows:

- *Frontend Bound*: Slots when the pipeline's fornt end under-supplies the back end. Front end is the portion of the pipeline responsible for delivering operations to be executed later by the back end.
- *Bad Speculation*: Slots wasted due to all aspects of incorrect speculations.
- *Retiring*: Slots utilized by useful operations.
- *Backend Bound*: Remaining stalled slots due to lack of required back-end resources to accept new operations.

We apply all benchmarks in PBench with each input STL file for architecture bottleneck analysis, and the top level result is shown in Figure 9. This figure shows the diversity of PBench in the architecture performance. The average percentage of frontend bound, bad speculation, backend bound and retiring for PBench is 5.23%, 3.74% 24.79% and 66.24%, respectively. We can observe that both the frontend and bad speculation take relatively small percentages in the total pipeline slots. The reason for the small frontend bound is because all these benchmarks have multiple repetitive functions like loop, which is indicated in the memory section of microarchitecture independent analysis. These repetitive functions, with their code size within the range of L1 code cache, result in large portion of L1 cache hits in instruction fetch, which makes the frontend bound not a bottleneck in the architecture performance. A small percentage of bad speculation indicates that the control flow of these benchmarks are predictable. Hence bad speculation is negligible in architecture performance, which is also consistent with our points in the control section of microarchitecture independent analysis. The backend bound of these benchmarks consumes a relatively large portion of pipeline slots. This is reasonable because backend bound represents slots stalled due to lack of computing resources or the delay in memory operation. So this stage should consume a considerable amount of pipeline slots. The interesting observation is that most of the benchmarks in PBench spend a majority of slots in retiring. Retiring represents the useful work committed by this processor. The higher ratio of retiring compared to the total slots is, the higher instructions per cycle the processor achieves. This indicates how efficient a processor executes a program.

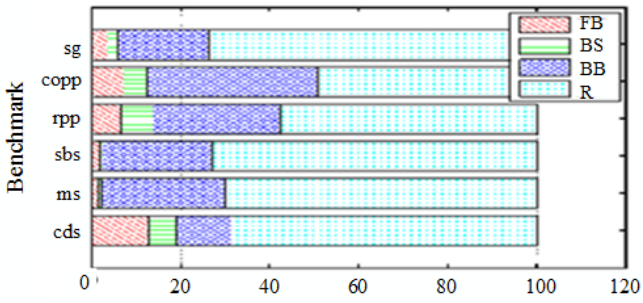

Fig. 9. Accumulative run time of PBench. Each benchmark is divided to four pipeline slots: frontend bound (FB), bad speculation (BS), backend bound (BB) and retiring (R).

The top level analysis of PBench reveals multiple ways for performance improvements. The first is the utilization of specialized function hardware. The low portion of bad speculation shows that the control predictability of PBench is high. The low frontend bound ratio infers the instruction of PBench has a relatively small size and large re-usability. These benchmarks have a high percentage of retiring, which means the processor issuers short instructions intensively, with less wait time. A predictable control flow, multiple iterative functions, as well as compact and intensive instruction execution, all these properties provide potential possibility of using fixed function hardware, e.g., instruction set extension or specialized processing unit. Another method for potential performance enhancement is to explore the parallelism. There are multiple levels of parallelism, and PBench utilizes most of instruction-level parallelism and thread-level parallelism. The reasons are as follows: the low frontend bound shows these programs have multiple loops, as indicated in their algorithm design, each benchmark utilizes loops in different levels. An outer loop for the operation of different layers, and many inner loops for iterative operations of the data of the current layer. On the other hand, the high retiring rate reveals that most of the time the processor can execute instructions without waiting for data, which indicates a low data dependency. Given the low data dependency, it is promising to use different levels of parallelism to improve the architecture performance of PBench. One strategy is that we can apply thread-level parallelism to the outer loop, and instruction-level parallelism to the inner loop.

C. Functional Performance Analysis

From microarchitecture independent analysis and architecture bottleneck analysis, we establish a concrete understanding of the intrinsic behaviors and architecture performance of PBench, and we know which methods are appropriate to improve 3D printing prefabrication. But architects would beneficial more if they could know which function of a program is the bottleneck by designing hardware accelerating this bottleneck. In order to provide this convenience, we perform functional performance analysis on PBench.

TABLE III
MAPPING OF FUNCTION BLOCKS TO BENCHMARKS

Benchmark	Function blocks
cds	FileIO, intersection, distance
ms	FileIO, intersection, sorting, traverse
sbs	FileIO, intersection, sorting, traverse
rpp	FileIO, intersection, sorting
copp	FileIO, contour offset
sg	FileIO, traverse

Functional performance analysis defines multiple common function blocks for PBench, and divides each benchmark into these blocks. Then the information of runtime and energy consumption of each function block of each program can be collected. The function blocks are defined as follows:

- *File I/O* represents the process of read the input data and store the output result.
- *Intersection* is a ubiquitous operation in PBench. It denotes the operation of computing the intersection between two geometry object, e.g. the intersection of triangle and line, or the intersection of contour with mesh.
- *Distance* computes the Euclidean distance of two vertex. This function is only executed in $shortest\ distance\ slicing$ algorithm.
- *Sorting* is a method of computing the sequence of an array. In PBench, the element of the array could be integers, or different data structures.
- *ContourOffset* represents the function of moving the contour inward certain distances. It is uniquely used in $contour offset\ path\ planning$.
- *Traverse* is an operation regarding a specific segment or the whole regions of a data block. It traverse through the data block, meanwhile executing simple arithmetic and logic operations, e.g., add, minus, or, etc.

Each benchmark in PBench can be regarded as the combination of multiple function blocks in certain sequence. The mapping of each benchmark to these function blocks is shown in Table III.

1) Runtime analysis: In order to provide accurate result, we evaluate the run time of PBench on real machines. The system configuration is the same as the experimental setup in architecture bottleneck analysis (as shown in Table II). On one hand, we want PBench to have diversity, so that each benchmark will explore something new. On the other hand, we eager to find the common behavior of PBench, which will provide insight into the methods for improving 3D printing prefabrication. So we perform runtime analysis in two dimensions. The first is dissimilarity, which characterizes the difference among benchmarks. The other is commonality, which shows the common behavior and the scalability of PBench.

To characterize the property of dissimilarity, we apply each benchmark of PBench with each design file into this system. We average the run time over each benchmark, and split the time fraction to each function block. The performance result is shown in Figure 10. Each bar represents the average run

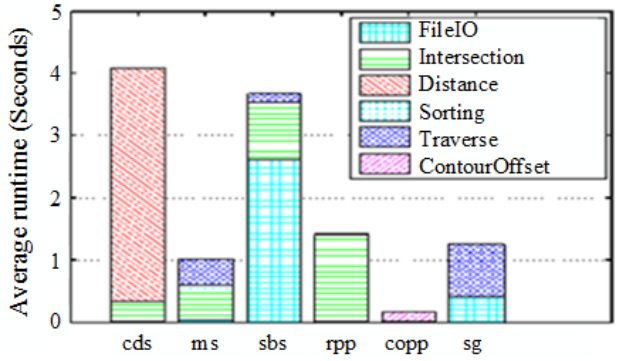

Fig. 10. Average runtime split for each function block.

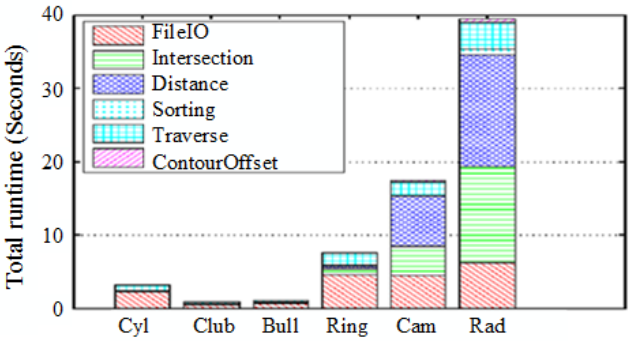

Fig. 11. Accumulative runtime of PBench.

time a benchmark takes for executing each design STL in the input set of PBench. The average run time for executing the six testbenches are 4.09, 1.02, 3.67, 1.43, 2.19, 1.42 $seconds$ for $closest\ distance\ slicing$, $marching\ slicing$, $sampling\ based\ slicing$, $rasterization\ path\ planning$, $contour\ offset\ path\ planning$ and $support\ generation$, respectively. Different regions of the bar indicate the portion of time each function block consumes. This figure shows the diversity of computing patterns for PBench by exploring the difference of function blocks in each benchmark. Although different benchmarks may share the same function block, the combinations of blocks and percentage of the run time for each function block are different, which results in different bottlenecks for each benchmark. $Closest\ distance\ slicing$ is dominated by $distance$ computing block. There are two dominant function blocks in $marching\ slicing$, $traverse$ and $intersection$ computing. In $sampling\ based\ slicing$, most of the time is spent in $FileIO$. The bottleneck function of $rasterization\ path\ planning$ and $contour\ offset\ path\ planning$ are $intersection$ processing and $contour\ offsetting$ block each. $Support\ generation$ is mainly influenced by $traversing$. It can also be inferred that among these six benchmarks, $closest\ distance\ slicing$ and $sampling\ based\ slicing$ are the two most time consuming algorithms, and these two benchmarks are all in the stage of $slicing$. This indicates that $slicing$ is the bottleneck among the three stages in prefabrication.

To analyze the characteristics of commonality and scalability, we apply all programs in PBench with each design document file, and compute the aggregated run time of each function block. The results are shown in Figure 11. Each bar indicates the total time spent executing PBench for each STL file, and the segments of the bar reflect the aggregated run time for each function block. The total run time for performing the six benchmarks are 3.31, 0.89, 1.24, 8.51, 22.56, 46.45 *seconds* for the STL file of *cylinder*, *club*, *bull*, *ring*, *cam* and *radial impeller*, respectively. It is observed that the run time of certain function blocks are constant or have little variation for different STL file, e.g., *FileIO* changes slightly for the STL file of *ring*, *cam* and *radial*. This is because of two reasons. Firstly, although *FileIO* is directly affected by the size of STL file, it is also influenced by the number of layers cut for the design STL file. The size of *radial* is larger than *cam* and *ring*, but the layers of *ring* and *cam* is more than *radial*. Secondly, there are many routine operations for *FileIO* such as the standard input and output format processing. These operations are constant when applied with different STL files. These two reasons makes the *FileIO* of these three STL files virtually the same. We also observe some interesting behavior from Figure 11. When STL size is small, the run time is dominated by *FileIO*. As STL grows larger and more complex, the function block of *intersection* and *distance* consume an increasingly significant portion of total run time. The reasons are as follows: Because *FileIO* has many routine operations, when STL is small, these operations become important and make *FileIO* the bottleneck. When STL grows larger, the data processing blocks become dominant. The data processing blocks are mainly *intersection* and *distance* computing in 3D printing prefabrication. *Intersection* is a popular function in PBench, e.g., *closest distance slicing*, *marching slicing*, *sampling based slicing*, *rasterization path planning*. *Distance* is only utilized by *closest distance slicing*, but as shown in Figure 10, it consumes a large percentage of time. So both the ratios of *intersection* and *distance* grow tremendously as STL size becomes large. The trend of STL design file is that it will become larger, more complex and more refinement. It can be predicted that in the future, 3D printing prefabrication will be dominated by *intersection* and *distance* computing blocks.

The dissimilarity and commonality analysis of PBench provide insights for hardware optimization. For the task of dealing with tongs of small size STL files, hardware improvement will be significant if architects focus on the optimization of *FileIO* function. On the other hand, when the goal is to increase the speed for executing large design documents, or to design a hardware benefiting the next-generation 3D printing prefabrication, we should pay more attention to *intersection* and *distance* function blocks.

2) Energy analysis: In order to provide solid hardware comparison baseline, we also provide energy analysis for PBench. We again perform this experiment on real machine of the same system configuration. The result is shown in Figure 12. Each line represents the energy consumption of each benchmark for the given input STL file. The first observation is that the energy consumption has an overall increasing trend as STL size grows larger. Specifically, the total energy consumption of PBench is 34.12, 9.20, 12.17, 84.20, 300.25, 769.81 *Joules* for *cylinder*, *club*, *bull*, *ring*, *cam*, *radial*, respectively. The overall increasing trend is resulted from the relationship between energy and runtime. Comparing Figure 11 and Figure 12, energy consumption is proportional to the runtime, which is affected by the size, complexity and number of layers of the design file. As STL evolves larger, the overall energy consumption also increases. We also observe that the energy dominating benchmark among PBench varies as STL changes. This is because *sampling based slicing* has more data independent operations than *closest distance slicing* and *marching slicing*. When STL size is small, these routine operations dominant, which make *sampling based slicing* dominate energy consumption. When STL grows larger, data dependent operations become dominant and this makes the *closest distance slicing* and *marching slicing* spend more energy. We also observe that there are some exception points in this figure. The STL files are arranged based on their size, but the energy consumption of *club* and *bull* is less than *cylinder*. This is because the energy consumption is not only affected by the size, but also by the number of layers of a STL file. The layers of *club* and *bull* is significantly smaller than *cylinder*, which result in the energy consumption of these two STL files smaller.

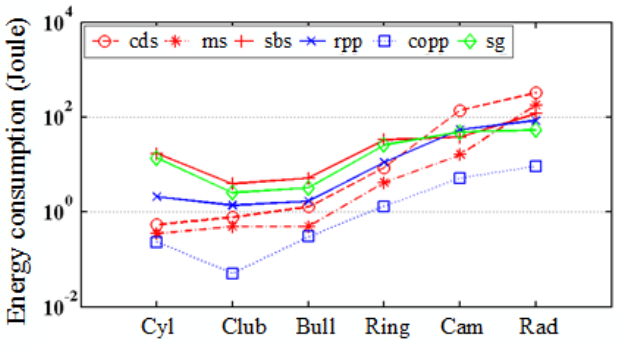

Fig. 12. Energy consumption of PBench.

D. Summary

The TotalChar framework reveals many interesting properties of this benchmark suite. Specifically, microarchitecture independent analysis shows the invariant characters of PBench in instruction mix, memory behavior and branch entropy. Architecture bottleneck analysis presents PBench to a popular modern computer system and discusses the architecture bottleneck of PBench. On the basis of this analysis, we conclude that 3D printing prefabrication has great potential to improve through accelerator design and parallelism exploration. Functional performance analysis divides PBench into several function blocks and examines the high level character of each block. This analysis provides the insight that we should

focus on $intersection$ and $distance$ function blocks for next-generation 3D printing prefabrication performance optimization. In sum, microarchitecture independent analysis forms a base understanding of PBench, architecture bottleneck analysis provides the method for hardware performance improvement, and functional performance analysis indicates the object for architecture design. These three dimension analysis approaches work collaboratively to characterize PBench and identify the potential on accelerating 3D printing prefabrication.

VI. Conclusion

In this paper, we investigated the computational challenge in the 3D printing prefabrication. We developed PBench, an open-source benchmark suite to establish the standard evaluation on 3D printing prefabrication. Also, we characterized PBench using TotalChar, which study PBench from three dimensions: microarchitecture independent analysis, architecture bottleneck analysis and functional performance analysis. The experimental results showed that there is a significant potential to accelerate 3D printing prefabrication by specified function accelerator design and parallelism exploration.

Acknowledgment

We thank our shepherd Dr. Brandon Lucia and the anonymous reviewers for their insightful comments on this paper. This work was in part supported by NSF grant CNS-1547167.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[2] D. L. Bourell, M. C. Leu, and D. W. Rosen. Roadmap for additive manufacturing: identifying the future of freeform processing. *The University of Texas at Austin, Austin, TX*, 2009.

[3] M. B. Breughe and L. Eeckhout. Selecting representative benchmark inputs for exploring microprocessor design spaces. *ACM Trans. Architecture and Code Optimization*, 10(4):37, 2013.

[4] X. Chen and S. McMains. Polygon offsetting by computing winding numbers. In *ASME Int'l Design Eng. Technical Conf. and Computers and Information in Eng. Conf.*, pages 565–575, 2005.

[5] Y. Chen, K. Li, and X. Qian. Direct geometry processing for telefabrication. *J. Computing and Information Science in Eng.*, 13(4):041002, 2013.

[6] Y. Chen and C. C. Wang. Layer depth-normal images for complex geometries: Part oneaccurate modeling and adaptive sampling. In *ASME 2008 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 717–728, 2008.

[7] J. Cong, V. Sarkar, G. Reinman, and A. Bui. Customizable domain-specific computing. *IEEE Design & Test of Computers*, (2):6–15, 2010.

[8] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. *Computational geometry*. Springer, 2000.

[9] A.-V. Do, B. Khorsand, S. M. Geary, and A. K. Salem. 3d printing of scaffolds for tissue regeneration applications. *Advanced healthcare materials*, 4(12):1742–1762, 2015.

[10] J. Dumas, J. Hergel, and S. Lefebvre. Bridging the gap: Automated steady scaffoldings for 3d printing. *ACM Transactions on Graphics*, 33(4):98, 2014.

[11] I. Gibson, D. W. Rosen, B. Stucker, et al. *Additive manufacturing technologies*. Springer, 2010.

[12] R. M. Gregori, N. Volpato, R. Minetto, and M. V. Da Silva. Slicing triangle meshes: An asymptotically optimal algorithm. In *2014 14th International Conference on Computational Science and Its Applications (ICCSA)*, pages 252–255. IEEE, 2014.

[13] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[14] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ACM/IEEE 42nd Int'l Symp. Computer Architecture (ISCA)*, pages 158–169. IEEE, 2015.

[15] H.-J. Kim, K.-H. Wie, S.-H. Ahn, H.-S. Choo, and C.-S. Jun. Slicing algorithm for polyhedral models based on vertex shifting. *International Journal of Precision Engineering and Manufacturing*, 11(5):803–807, 2010.

[16] P. Kulkarni, A. Marsan, and D. Dutta. A review of process planning techniques in layered manufacturing. *Rapid Prototyping Journal*, 6(1):18–35, 2000.

[17] B. Mueller. Additive manufacturing technologies–rapid prototyping to direct digital manufacturing. *Assembly Automation*, 32(2), 2012.

[18] S. Mueller, S. Im, S. Gurevich, A. Teibrich, L. Pfisterer, F. Guimbretière, and P. Baudisch. Wireprint: 3d printed previews for fast prototyping. In *Proc. 27th ACM Symp. User Interface Software and Technology*, pages 273–280, 2014.

[19] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 110–119. IEEE, 2014.

[20] F. Rengier, A. Mehndiratta, H. von Tengg-Kobligk, C. M. Zechmann, R. Unterhinninghofen, H.-U. Kauczor, and F. L. Giesel. 3d printing based on imaging data: review of medical applications. *International journal of computer assisted radiology and surgery*, 5(4):335–341, 2010.

[21] S. J. Rock and M. J. Wozny. Utilizing topological information to increase scan vector generation efficiency. In *Proc. Solid Freeform Fabrication Symp.*, pages 3–5, 1991.

[22] S. J. Rock and M. J. Wozny. Generating topological information from a bucket of facets. In *Proceedings of Solid Freeform Fabrication Symposium, Austin, TX, Aug*, pages 3–5. Citeseer, 1992.

[23] Y. S. Shao and D. Brooks. Isa-independent workload characterization and its implications for specialized architectures. In *IEEE Int'l Symp. Performance Analysis of Systems and Software*, pages 245–255, 2013.

[24] J. R. e. a. Tumbleston. Continuous liquid interface production of 3d objects. *Science*, 347(6228):1349–1352, 2015.

[25] J. Vanek, J. A. Galicia, and B. Benes. Clever support: Efficient support structure generation for digital fabrication. In *Computer graphics forum*, volume 33, pages 117–125. Wiley Online Library, 2014.

[26] M. Vatani, A. Rahimi, F. Brazandeh, and A. S. Nezhad. An enhanced slicing algorithm using nearest distance analysis for layer manufacturing. In *Proceedings of World Academy of Science, Engineering and Technology*, volume 37, pages 721–726, 2009.

[27] B. R. Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63, 1992.

[28] A. Yasin. A top-down method for performance analysis and counters architecture. In *IEEE Int'l Symp. Performance Analysis of Systems and Software*, pages 35–44, 2014.

[29] T. Yokota, K. Ootsu, and T. Baba. Potentials of branch predictors: From entropy viewpoints. In *Architecture of Computing Systems–ARCS 2008*, pages 273–285. Springer, 2008.

[30] K. Zeng, N. Patil, H. Gu, H. Gong, D. Pal, T. Starr, and B. Stucker. Layer by layer validation of geometrical accuracy in additive manufacturing processes. In *Proceedings of the Solid Freeform Fabrication Symposium, Austin, TX, Aug*, pages 12–14, 2013.

[31] Z. Zhang and S. Joshi. An improved slicing algorithm with efficient contour construction using stl files. *The International Journal of Advanced Manufacturing Technology*, 80(5-8):1347–1362, 2015.

[32] H. Zhao, C. C. Wang, Y. Chen, and X. Jin. Parallel and efficient boolean on polygonal solids. *The Visual Computer*, 27(6-8):507–517, 2011.

[33] C. Zhou, Y. Chen, Z. Yang, and B. Khoshnevis. Digital material fabrication using mask-image-projection-based stereolithography. *Rapid Prototyping Journal*, 19(3):153–165, 2013.

[34] C. Zhou, H. Ye, and F. Zhang. A novel low-cost stereolithography process based on vector scanning and mask projection for high-accuracy, high-speed, high-throughput, and large-area fabrication. *Journal of Computing and Information Science in Engineering*, 15(1):011003, 2015.