Fidelity and Scaling of the PARSEC Benchmark Inputs

Christian Bienia and Kai Li, Princeton University

***Abstract*—A good benchmark suite should provide users with inputs that have multiple levels of fidelity for different use cases such as running on real machines, register level simulations, or gate-level simulations. Although input reduction has been explored in the past, there is a lack of understanding how to systematically scale input sets for a benchmark suite. This paper presents a framework that takes the novel view that benchmark inputs should be considered approximations of their original, full-sized inputs. It formulates the input selection problem for a benchmark as an optimization problem that maximizes the accuracy of the benchmark subject to a time constraint. The paper demonstrates how to use the proposed methodology to create several simulation input sets for the PARSEC benchmarks and how to quantify and measure their approximation error. The paper also shows which parts of the inputs are more likely to distort their original characteristics. Finally, the paper provides guidelines for users to create their own customized input sets.**

I. Introduction

Computer architects need detailed simulations for their microarchitecture designs. However, simulators are typically many orders of magnitude slower than a real machine. Running a set of benchmark applications with realistic inputs on a simulator will far exceed the design time limits, even using thousands of computers.

To reduce the number of simulated instructions of a benchmark, a commonly used method is to take a subset of the instructions with statistical sampling, which requires a sophisticated mechanism in the simulator [4], [5], [20], [25]. Another method is to reduce the size of the input for a benchmark [14], [22]. Reduced inputs are easy to use with any simulator, but there is no systematic framework for scaling input sets in a continuous way. Furthermore, there is a lack of understanding of the tradeoffs between accuracy and cost involved when reducing inputs for a given benchmark.

We believe a good benchmark suite should provide users with inputs of multiple levels of fidelity, scaling from realistic inputs for execution on real machines down to small inputs for detailed simulations. The key question is how to scale such inputs for a benchmark so that its execution with a scaled-down input produces meaningful performance predictions for computer architects.

This paper presents a framework for the input-scaling problem of a benchmark. We view scaled inputs as approximations of the original, full-sized inputs. As the inputs are scaled down, the benchmark becomes increasingly inaccurate. The question of how to create many derivatives of real inputs with varying size and accuracy motivates the following optimization problem: *Given a time budget, what is the optimal selection of inputs for a set of benchmark programs?* We will show that this optimization problem is the classical *multiple-choice knapsack problem* (MCKP), which is NP-hard. The formulation of the problem allows us to derive the general guidelines for designing multiple inputs with different levels of fidelity for a benchmark suite.

The paper proposes the methodology and implementation of scaled inputs for the Princeton Application Repository for Shared-Memory Computers (PARSEC) [3]. The PARSEC benchmark suite is designed to represent emerging workloads. The current release (version 2.1) consists of 13 multithreaded programs in computer vision, video encoding, physical modeling, financial analytics, content-based search, and data deduplication. We have used the methodology to design six sets of inputs with different levels of fidelity. The first version of PARSEC was released only two years ago. It has been well adopted by the computer architecture community to evaluate multicore designs and multiprocessor systems.

To evaluate the impact of input scaling, we have defined a measure called *approximation error*. Using this measure, we have analyzed several input sets of PARSEC and shown which reduced inputs are more likely to distort the original characteristics of the PARSEC benchmarks. We furthermore analytically derive a scope for each input which defines the range of architectures for which the input can be expected to be reasonably accurate. Such results are helpful for PARSEC users to understand the implications when creating customized inputs for their simulation time budgets.

This paper makes several contributions. First, it presents a novel methodology to analyze how scaled inputs affect the accuracy of a benchmark suite. Second, it formulates the input selection problem for a benchmark as an optimization problem that maximizes the accuracy of a benchmark subject to a time constraint. Third, it describes how the PARSEC inputs have been scaled and shows which parts of the inputs are more likely to distort the original program characteristics. More importantly, it discusses in which situations small input sets might produce highly misleading program behavior. Finally, the paper provides guidelines for users to create their own customized input sets for PARSEC.

II. Input Fidelity

This section introduces the conceptual foundation and terminology to discuss the inaccuracies in benchmark inputs. Our concepts and definitions follow the terms commonly used in the fields of mathematical approximation theory and scientific modeling [1].

The main insight is that inputs for benchmark programs are almost never real program inputs. An example is a server program whose behavior is input dependent, but its input data contains confidential user information that cannot be made publicly available. Also, computationally intensive applications require too much time to complete. For that reason benchmark inputs are typically derived from real program inputs by reducing them in a suitable way. We call the process of creating a range of reduced inputs *input scaling*. This process can be compared to stratified sampling. The idea of stratified sampling is to take samples in a way that leverages knowledge about the sampling population so that the overall characteristics of the population are preserved as much as possible.

Benchmark inputs can be thought of as models of real inputs. Their purpose is to approximate real program behavior as closely as possible, but even if the benchmark program itself is identical to the actual application, some deviations from its real-world behavior must be expected. We call these deviations the *approximation error* because they are unintended side effects which can distort performance measurements in subtle ways. Sometimes very noticeable errors can be identified. We will refer to these errors as *scaling artifacts*. A scaling artifact can increase the inaccuracy of a benchmark and sometimes it may lead to highly misleading results. In such cases, scaling artifacts are good indicators whether an input set can accurately

approximate the real workload inputs. Identifying such artifacts is essential to determine the constraints of a benchmark input set.

We define *fidelity* as the degree to which a benchmark input set produces the same program behavior as the real input set. An input set with high fidelity has a small approximation error with few or no identifiable scaling artifacts, whereas an input set with low fidelity has a large approximation error and possibly many identifiable scaling artifacts. Input fidelity can be measured by quantifying the approximation error. It is common that approximations are optimized to achieve a high degree of fidelity in a limited target area, possibly at the expense of reduced accuracy in other areas. We call this target area of high fidelity the *scope* of an input. The scope can be thought of as a set of restrictions for a reduced input beyond which the input becomes very inaccurate. An example is an overall reduction of work units in an input set, which will greatly reduce execution time, but limit the amount of CPUs that the program can stress simultaneously. Running the benchmark on CMPs with more cores than the input includes may lead to many noticeable scaling artifacts in the form of idle CPUs.

A. Optimal Input Selection

Input scaling can be used to create a continuum of approximations of a real-world input set with decreasing fidelity and instruction count. This allows benchmark users to trade benefit in the form of benchmarking accuracy for lower benchmarking cost as measured by execution or simulation time. This motivates the question what the optimal combination of inputs from a given selection of input approximations is that maximizes the accuracy of the benchmark subject to a time budget.

If the time budget is fixed and an optimal solution is desired, the problem assumes the structure of the multiple-choice knapsack problem (MCKP), which is a version of the binary knapsack problem with the addition of disjoint multiple-choice constraints [15], [21]. For a given selection of m benchmark programs with disjoint sets of inputs N_i, $i = 1,\ldots,m$, we will refer to the approximation error of each respective input $j \in N_i$ with the variable $a_{ij} \geqq 0$. If we define a solution variable $x_{ij} \in \{0,1\}$ for each input that specifies whether an input has been selected then we can formally define the goal of the optimization problem as follows:

$$\min \sum_{i=1}^{m} \sum_{j \in N_i} a_{ij} x_{ij} \tag{1}$$

The solution of the problem is the set of all x_{ij} that describes which inputs are optimal to take. If we refer to the time each input takes to execute with $t_{ij} \geqq 0$ and to the total amount of time available for benchmark runs with $T \geqq 0$ then we can describe which solutions are acceptable as follows:

$$\sum_{i=1}^{m} \sum_{j \in N_i} t_{ij} x_{ij} \leqq T \tag{2}$$

Lastly, we need to state that exactly one input must be selected for each benchmark program and that we do not allow partial or negative inputs:

$$\sum_{j \in N_i} x_{ij} = 1 \quad i = 1,\ldots,m \tag{3a}$$

$$x_{ij} \in \{0,1\} \quad j \in N_i, \quad i = 1,\ldots,m \tag{3b}$$

MCKP is an NP-hard problem, but it can be solved in pseudo-polynomial time through dynamic programming [6], [19]. The most efficient algorithms currently known first solve the linear version of MCKP (LMCKP) that is obtained if the integrality constraint $x_{ij} \in \{0,1\}$ is relaxed to $0 \leqq x_{ij} \leqq 1$. LMCKP is a variant of of the fractional knapsack problem and can be solved in $O(n)$ time by a greedy algorithm. The initial feasible solution that is obtained that way is used as a starting point to solve the more restrictive MCKP version of the problem with a dynamic programming approach. It keeps refining the current solution in an enumerative fashion by adding new classes until an optimal solution has been found. Such an algorithm can solve even very large data instances within a fraction of a second in practice, as long as the input data is not strongly correlated [19].

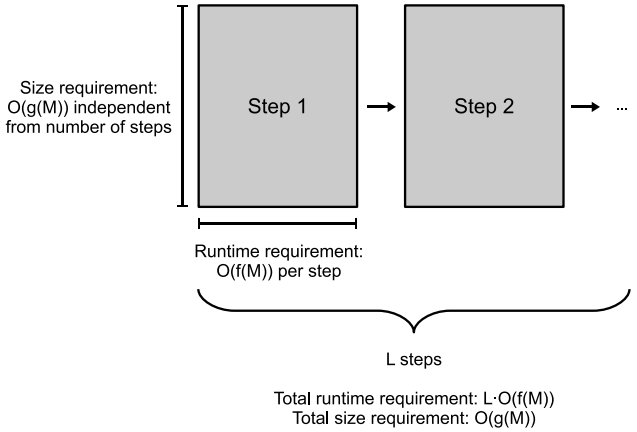

Fig. 1. The impact that linear and complex input size scaling typically have on a program. Complex scaling of M usually affects the execution time $f(M)$ and the memory requirement $g(M)$. Linear scaling typically only changes the number of repetitions L.

B. Scaling Model

The most common way to express the asymptotic runtime behavior of a program is to use a function of a single parameter N, which is the total size of the input. This is too simplistic for real-world programs, which usually have inputs that are described by several parameters, each of which may significantly affect the behavior of the programs. This section presents a simple scaling model for reasoning about the effect of input parameters on program behavior.

In our simple model, we group the inputs of a program into two components: the linear component and the complex component. The linear component includes the parts that have a linear effect on the execution time of the program such as streamed data or the number of iterations of the outermost loop of the workload. The complex component includes the remaining parts of the input which can have any effect on the program. For example, a fully defined input for a program that operates on a video would be composed of a complex part that determines how exactly to process each frame and a linear part that determines how many frames to process.

Our motivation for this distinction is twofold: First, real-world workloads frequently apply complex computational steps which are difficult to analyze in a repetitive fashion that is easy to count and analyze. Second, the two components have different qualitative scaling characteristics because the complex

TABLE I
THE SIX STANDARDIZED INPUT SETS OFFERED BY PARSEC LISTED IN ORDER OF INCREASING SIZE. LARGER INPUT SETS GUARANTEE THE SAME PROPERTIES OF ALL SMALLER INPUT SETS. TIME IS APPROXIMATE SERIAL EXECUTION TIME ON REAL MACHINES.

Input Set	Description	Time Budget T	Purpose
`test`	Minimal execution time	N/A	Test & Development
`simdev`	Best-effort code coverage of real inputs	N/A	
`simsmall`	Small-scale experiments	$\lessapprox$ 1 s	Simulations
`simmedium`	Medium-scale experiments	$\lessapprox$ 4 s	
`simlarge`	Large-scale experiments	$\lessapprox$ 15 s	
`native`	Real-world behavior	$\lessapprox$ 15 min	Native execution

input part often has to remain in a fairly narrow range whereas the linear component can often be chosen rather freely. In the example of the video processing program, the complex input part determining the work for each single frame is limited by the algorithms that are available and need to be within the range of video frame sizes that make sense in practice, which means there is a tight upper bound on the number of processing steps as well as lower and upper bounds on the frame resolution that make sense. These bounds significantly limit the scaling range, and the possible choice among different algorithms make this part of the input difficult to analyze. The number of frames, however, is unconstrained and can practically reach any number.

These two components of an input affect the input scaling behavior of the program in different ways, as summarized in Figure 1. The impact of the complex input parts is typically highly dependent on the workload so that few general guidelines can be given. It is common that the asymptotic runtime of the program increases superlinearly but its memory requirements often grow only linearly because most programs only keep the input data in a parsed but essentially unmodified form in memory. This property makes it hard to scale an input using its complex components without skewing it. Complex scaling almost always reduces working set sizes.

The linear input components typically do not affect the memory requirements or working set sizes of the program that much because they involve a form of repetition of previous steps. Their strong and direct impact on the execution time of the program make them suitable for input size scaling. Usually, there is no upper limit for linear input scaling, but reducing the input to the point where it includes few if any repetitions can often result in strong scaling artifacts because the individual steps are not exactly the same or not completely independent from each other. For example, if a program takes the output of its previous iteration as the input for its next iteration, it usually results in significant communication between threads. The underlying communication patterns may vary significantly from one iteration to the next. In this case, the number of repetitions included in the input must be large enough to stabilize what type of tasks the program performs on average.

III. PARSEC INPUTS

This section describes the inputs of PARSEC and how they are scaled. PARSEC offers six input sets. Each set contains exactly one fully defined input for each PARSEC benchmark. An input is composed of all input files required by the program and a predetermined way to invoke the binary. Input sets can be distinguished by the amount of work their inputs contain. This determines what an input set can be used for. Table I gives an overview of the six PARSEC input sets ordered in ascending order by the allowed time budget.

The `native` input set is the closest approximations to realistic inputs, even though it is not authoritative. Only benchmarks with real-world programs using real-world inputs are authoritative [12]. Smaller input sets can be considered increasingly inaccurate approximations of real-world inputs. Users of the benchmark suite should therefore generally use the largest input set possible. The smallest input set which we consider acceptable for at least some performance experiments is `simsmall`.

A. Scaling of PARSEC Inputs

PARSEC inputs predominantly use linear input scaling to achieve the large size reduction from real inputs to `native` and `simlarge` and a combination of linear and complex scaling to derive the `simmedium` and `simsmall` input sets from `simlarge`. For that reason the differences between real inputs, `native` and `simlarge` should be relatively small. The input sets `simdev` and `test` were created in a completely different way and should not be used for performance experiments at all. The various inputs suitable for performance measurements are summarized in Table II.

Most parts of the complex input components are identical between the input sets `simlarge` and `native`. In seven cases at least one part of the complex input component is not identical between the two input sets: `Canneal`, `ferret`, `fluidanimate`, `freqmine`, `raytrace`, `streamcluster` and `x264` all have at least one input component that exhibits complex scaling behavior which might affect the program noticeably. However, only in the case of `streamcluster` could we measure a noticeable and strong impact of that property on the program characteristics. This is because there is a direct, linear relationship between the working set of the program and the selected block size, which can be freely chosen as part of the input and which has also been scaled between input sets. For `simlarge` the working set corresponding to the block size is 8 MB, which means a working set size between 64 MB and 128 MB can be expected for the `native` input. In all other cases we expect the differences between `simlarge` and `native` to be negligible on contemporary machines.

`Blackscholes`, `dedup`, `swaptions` and `vips` all break their input into small chunks which are processed one after another. Their inputs are the easiest to scale and generally should show little variation. In the case of `blackscholes` and `dedup` some impact on the working set sizes should be expected because each input unit can be accessed more than once by the program.

The most difficult inputs to scale are the ones of `freqmine`. They exhibit no linear component, which means that any form of input scaling might alter the characteristics of the workload significantly. Moreover, `freqmine` parses and stores its input data internally as a frequent-pattern tree (FP-tree) that will be mined during program execution. An FP-tree is a compressed form of the transaction database that can be traversed in multiple ways. This makes the program behavior highly dependent on the exact properties of the input data, which might further amplify the problem.

TABLE II
OVERVIEW OF PARSEC INPUTS AND HOW THEY WERE SCALED. WE POINT OUT IN WHICH CASES THE EXACT CONTENTS OF THE INPUT DATA CAN HAVE A STRONG IMPACT ON THE CODE PATH OR THE CHARACTERISTICS OF THE PROGRAM.

Program	Input Set	Problem Size		Comments
		Complex Component	Linear Component	
blackscholes	simsmall		4,096 options	
	simmedium		16,384 options	
	simlarge		65,536 options	
	native		10,000,000 options	
bodytrack	simsmall	4 cameras, 1,000 particles, 5 layers	1 frame	
	simmedium	4 cameras, 2,000 particles, 5 layers	2 frames	
	simlarge	4 cameras, 4,000 particles, 5 layers	4 frames	
	native	4 cameras, 4,000 particles, 5 layers	261 frames	
canneal	simsmall	100,000 elements	10,000 swaps per step, 32 steps	
	simmedium	200,000 elements	15,000 swaps per step, 64 steps	
	simlarge	400,000 elements	15,000 swaps per step, 128 steps	
	native	2,500,000 elements	15,000 swaps per step, 6,000 steps	
dedup	simsmall		10 MB data	Data affects behavior
	simmedium		31 MB data	
	simlarge		184 MB data	
	native		672 MB data	
facesim	simsmall	80,598 particles, 372,126 tetrahedra	1 frame	Complex scaling challenging
	simmedium	80,598 particles, 372,126 tetrahedra	1 frame	
	simlarge	80,598 particles, 372,126 tetrahedra	1 frame	
	native	80,598 particles, 372,126 tetrahedra	100 frames	
ferret	simsmall	3,544 images, find top 10 images	16 queries	
	simmedium	13,787 images, find top 10 images	64 queries	
	simlarge	34,793 images, find top 10 images	256 queries	
	native	59,695 images, find top 50 images	3,500 queries	
fluidanimate	simsmall	35,000 particles	5 frames	
	simmedium	100,000 particles	5 frames	
	simlarge	300,000 particles	5 frames	
	native	500,000 particles	500 frames	
freqmine	simsmall	250,000 transactions, min support 220		Data affects behavior
	simmedium	500,000 transactions, min support 410		
	simlarge	990,000 transactions, min support 790		
	native	250,000 transactions, min support 11,000		
raytrace	simsmall	480×270 pixels, 1 million polygons	3 frames	Data affects behavior
	simmedium	960×540 pixels, 1 million polygons	3 frames	
	simlarge	$1,920 \times 1,080$ pixels, 1 million polygons	3 frames	
	native	$1,920 \times 1,080$ pixels, 10 million polygons	200 frames	
streamcluster	simsmall	4,096 points per block, 32 dimensions	1 block	
	simmedium	8,192 points per block, 64 dimensions	1 block	
	simlarge	16,384 points per block, 128 dimensions	1 block	
	native	200,000 points per block, 128 dimensions	5 blocks	
swaptions	simsmall		16 swaptions, 5,000 simulations	
	simmedium		32 swaptions, 10,000 simulations	
	simlarge		64 swaptions, 20,000 simulations	
	native		128 swaptions, 1,000,000 simulations	
vips	simsmall		$1,600 \times 1,200$ pixels	
	simmedium		$2,336 \times 2,336$ pixels	
	simlarge		$2,662 \times 5,500$ pixels	
	native		$18,000 \times 18,000$ pixels	
x264	simsmall	640×360 pixels	8 frames	Data affects behavior
	simmedium	640×360 pixels	32 frames	
	simlarge	640×360 pixels	128 frames	
	native	$1,920 \times 1,080$ pixels	512 frames	

The complex input components of the facesim inputs have not been scaled at all. Doing so would require generating a new face mesh, which is a challenging process. Significant reductions of the mesh resolution can also cause numerical instabilities. The three simulation inputs of facesim are therefore identical and should be considered as belonging to the simlarge input set.

Besides freqmine three more programs significantly alter their behavior depending on the data received. Dedup builds a database of all unique chunks that are encountered in the input stream. Less redundancy in the input stream will cause larger working sets. Raytrace follows the path of light rays through a scene. Small alterations of the scene or the movement of the camera might cause noticeable changes of the execution or working set sizes. However, in natural scenes with realistic camera movement this effect is likely to be small if the number of light rays is sufficiently large because their fluctuations will average out due to the law of large numbers. Finally, x264 uses a significantly larger frame size for its native input. Just like vips the program breaks an input frame into smaller chunks of fixed size and processes them one at a time. However, x264 must keep some frames in memory after they have been processed because it references them to encode subsequent frames. This property increases working set sizes and the amount of shared data with the frame size and is the reason why we classify the image resolution of the input as a complex input component.

B. General Scaling Artifacts

One common scaling artifact caused by linear input scaling is an exaggerated warmup effect because the startup cost has to be amortized within a shorter amount of time. Furthermore, the serial startup and shutdown phases of the program will also appear inflated in relation to the parallel phase. This is an inevitable consequence if workloads are to use inputs with working sets comparable real-world inputs but with a significantly reduced execution time - the programs will have

to initialize and write back a comparable amount of data but will do less work with it.

Consequently, all characteristics will be skewed towards the initialization and shutdown phases. In particular the maximum achievable speedup is limited due to Amdahl's Law if the whole execution of the program is taken into consideration. It is important to remember that this does not reflect real program behavior. The skew should be compensated for by either excluding the serial initialization and shutdown phases and limiting all measurements to the Region-of-Interest (ROI) of the program, which was defined to include only the representative parallel phase, or by measuring the phases of the program separately and manually weighing them correctly. We believe it is safe to assume that the serial initialization and shutdown phases are negligible in the real inputs, which allows us to completely ignore them for experiments. Benchmark users who do not wish to correct measurements in such a way should limit themselves to the `native` input set, which is a much more realistic description of real program behavior that exhibits these scaling artifacts to a much lesser extent.

C. Scope of PARSEC Inputs

In this section we will briefly describe what the constraints of the PARSEC simulation inputs are and under which circumstances we can expect to see additional, noticeable scaling artifacts. The most severe limitations are the amount of parallelism and the size of the working sets.

The PARSEC simulation inputs were scaled for machines with up to 64 cores and with up to tens of megabytes of cache. These limitations define the scope of these inputs, with smaller inputs having even tighter bounds. If the inputs are used beyond these restrictions noticeable scaling artifacts such as idle CPUs caused by limited parallelism must be expected. The `native` input set should be suitable for machines far exceeding these limitations.

TABLE III
WORK UNITS CONTAINED IN THE SIMULATION INPUTS. THE NUMBER OF WORK UNITS PROVIDED BY THE INPUTS IS AN UPPER BOUND ON THE NUMBER OF THREADS THAT CAN WORK CONCURRENTLY. ANY OTHER BOUNDS ON PARALLELISM THAT ARE LOWER ARE GIVEN IN PARENTHESES.

Program	Input Set		
	simsmall	**simmedium**	**simlarge**
blackscholes	4,096	16,384	65,536
bodytrack	60	60	60
canneal	$\leq 5.0 \cdot 10^4$	$\leq 1.0 \cdot 10^5$	$\leq 2.0 \cdot 10^5$
dedup	2,841	8,108	94,130
facesim	80,598	80,598	80,598
ferret	16	64	256
fluidanimate	$\leq 3.5 \cdot 10^4$	$\leq 1.0 \cdot 10^5$	$\leq 3.0 \cdot 10^5$
freqmine	23	46	91
raytrace	1,980	8,040	32,400
streamcluster	4,096	8,192	16,384
swaptions	16	32	64
vips	475 (50)	1369 (74)	3612 (84)
x264	8	32	128

Reducing the size of an input requires a reduction of the amount of work contained in it which typically affects the amount of parallelism in the input. We have summarized the number of work units in each simulation input set in Table III. This is an upper bound on the number of cores that the input can stress simultaneously.

Workloads with noticeably low amounts of work units are `bodytrack`, `ferret`, `freqmine`, `swaptions` and `x264`. `Ferret` processes image queries in parallel, which means that the number of queries in the input limits the amount of cores it can use. This can be as little as 16 for `simsmall`. The amount of parallelism in the simulation input sets of `swaptions` is comparable. The smallest work unit for the program is a single swaption, only 16 of which are contained in `simsmall`. `X264` uses coarse-grain parallelism that assigns whole frames to individual threads. The number of possible cores the program can use is thus restricted to the number of images in the input, which is only eight in the case of `simsmall`. The number of work units that `vips` can simultaneously process is technically limited to the cumulative size of the output buffers, which can be noticeable on larger CMPs. This limitation has been removed in later versions of `vips` and will probably disappear in the next version of PARSEC. The amount of parallelism contained in the `bodytrack` inputs is limited by a vertical image pass during the image processing phase. It was not artificially introduced by scaling, real-world inputs exhibit the same limitation. It is therefore valid to use the simulation inputs on CMPs with more cores than the given limit. The upper bound introduced by input scaling is given by the number of particles, which is nearly two orders of magnitude larger. The natural limitation of parallelism during the image processing phase should only become noticeable on CMPs with hundreds of cores because image processing takes up only a minor part of the total execution time. The bound on parallelism for `canneal` and `fluidanimate` is probabilistic and fluctuates during runtime. It is guaranteed to be lower than the one given in Table III but should always be high enough even for extremely large CMPs.

Input scaling can also have a noticeable effect on the working sets of a workload and some reduction should be expected in most cases. However, the impact is significant in the cases of 'unbounded' workloads [3], which are `canneal`, `dedup`, `ferret`, `freqmine` and `raytrace`. A workload is unbounded if it has the qualitative property that its demand for memory and thus working sets is not limited in practice. It should therefore never fully fit into a conventional cache. Any type of input that requires less than all of main memory must be considered scaled down. For example, `raytrace` moves a virtual camera through a scene which is then visualized on the screen. Scenes can reach any size, and given enough time each part of it can be displayed multiple times and thus create significant reuse. This means the entire scene forms a single, large working set which can easily reach a size of many gigabytes.

A scaled-down working set should generally not fit into a cache unless its unscaled equivalent would also fit. Unfortunately larger working sets also affect program behavior on machines with smaller caches because the miss rate for a given cache keeps growing with the working set if the cache cannot fully contain it. We therefore do not give exact bounds on cache sizes as we did with parallelism, an impact on cache miss rates must be expected for all cache sizes. Instead we will account for this effect by including the cache behavior for a range of cache sizes in the aproximation error.

IV. VALIDATION OF PARSEC INPUTS

This section addresses the issue of accuracy of smaller input sets. We will first describe our methodology and then report the approximation error of the input sets relative to the entire PARSEC benchmark suite.

A. Methodology

An ideal benchmark suite should consist of a diverse selection of representative real-world programs with realistic inputs. As described in Section II, an optimal selection of inputs should minimize the deviation of the program behavior for a target time

limit while maintaining the diversity of the entire benchmark suite. Thus, analyzing and quantifying program behavior is the fundamental challenge in measuring differences between benchmarks and their inputs.

We view program behavior as an abstract, high-dimensional feature space of potentially unlimited size. It manifests itself in a specific way such that it can be measured in the form of characteristics when the program is executed on a given architecture. We can think of this process as taking samples from the behavior space at specific points defined by a particular architecture-characteristic pair. Given enough samples an image of the program behavior emerges.

We chose a set of characteristics and measured them for the PARSEC simulation inputs on a particular architecture. We then processed the data with Principal Component Analysis (PCA) to automatically eliminate highly correlated data. The result is a description of the program and input behavior that is free of redundancy.

We define *approximation error* as the dissimilarity between different inputs for the same program, as mentioned in Section II. One can measure it by computing the pairwise distances of the data points in PCA space. We will use approximation error as the basic unit to measure how accurate a scaled input is for its program. To visualize the approximation error of all benchmark inputs we use a dendrogram which shows the similarity or dissimilarity of the various inputs with respect to each other.

This methodology to analyze program characteristics is the common method for similarity analysis, but its application to analyze approximation errors is new. Measuring characteristics on an ideal architecture is frequently used to focus on program properties that are inherent to the algorithm implementation and not the architecture [2], [3], [24]. PCA and hierarchical clustering have been in use for years as an objective way to quantify similarity [9], [10], [13], [18], [23].

1) Program Characteristics: For our analysis of the program behavior we chose a total of 73 characteristics that were measured for each of the 39 simulation inputs of PARSEC 2.1, yielding a total of 2,847 sample values that were considered. Our study focuses on the parallel behavior of the multithreaded programs relevant for studies of CMPs. The characteristics we chose encode information about the instruction mix, working sets and sharing behavior of each program as follows:

- **Instruction Mix** 25 characteristics that describe the breakdown of instruction types relative to the total amount of instructions executed by the program
- **Working Sets** 8 characteristics encoding the working set sizes of the program by giving the miss rate for different cache sizes
- **Sharing** 40 characteristics describing how many lines of the total cache are shared and how intensely the program reads or writes shared data

The working set and sharing characteristics were measured for a total of 8 different cache sizes ranging from 1 MBytes to 128 MBytes to include information about a range of possible cache architectures. This approach guarantees that unusual changes in the data reuse behavior due to varying cache sizes or input scaling are captured by the data. The range of cache sizes that we considered has been limited to realistic sizes to make sure that the results of our analysis will not be skewed towards unrealistic architectures.

2) Experimental Setup: To collect the characteristics of the input sets we simulate an ideal machine that can complete all instructions within one cycle using `Simics`. We chose an ideal machine architecture because we are interested in properties inherent to the program, not in characteristics of the underlying architecture. The binaries which we used are the official precompiled PARSEC 2.1 binaries that are publicly available on the PARSEC website. The compiler used to generate the precompiled binaries was `gcc 4.4.0`.

We simulated an 8-way CMP with a single cache hierarchy level that is shared between all threads. The cache is 4-way associative with 64 byte lines. The capacity of the cache was varied from 1 MB to 128 MB to obtain information about the working set sizes with the corresponding sharing behavior. Only the Region-of-Interest (ROI) of the workloads was characterized.

3) Principal Component Analysis: Principal Component Analysis (PCA) is a mathematical method to transform a number of possibly correlated input vectors into a smaller number of uncorrelated vectors. These uncorrelated vectors are called the principal components (PC). We employ PCA in our analysis because PCA is considered the simplest way to reveal the variance of high-dimensional data in a low dimensional form.

To compute the principal components of the program characteristics, the data is first mean-centered and normalized so it is comparable with each other. PCA is then used to reduce the number of dimensions of the data. The resulting principal components have decreasing variance, with the first PC containing the most amount of information and the last one containing the least amount. We use the Kaiser's Criterion to eliminate PCs which do not contain any significant amount of information in an objective way. Only the top PCs with eigenvalues greater than one are kept, which means that the resulting data is guaranteed to be uncorrelated but to still contain most of the original information.

4) Approximation Error: All PARSEC inputs are scaled-down versions of a single real-world input, which means the more similar an inputs is to a bigger reference input for the same program the smaller is its approximation error. After the data has been cleaned up with PCA, this similarity between any two PARSEC inputs can be measured in a straightforward manner by calculating the Euclidean distance (or L_2) between them. Inputs with similar characteristics can furthermore be grouped into increasingly bigger clusters with hierarchical clustering. This method assigns each input set to an initial cluster. It then merges the two most similar clusters repeatedly until all input sets are contained in a single cluster. The resulting structure can be visualized with a dendrogram.

Inputs which merge early in the dendrogram are very similar. The later inputs merge the more dissimilar they are. Inputs for the same workload which preserve its characteristics with respect to other programs in the suite should fully merge before they join with inputs from any other benchmarks. Ideally all inputs for the same workload form their own complete clusters early on before they merge with clusters formed by inputs of any other workloads.

5) Limitations: We express all approximation errors relative to `simlarge`, the largest simulation input. While it would be interesting to know how `simlarge` compares to `native` or even real-world inputs, this information is of limited practical value because it is infeasible to use even bigger inputs for most computer architecture studies. For the most part, benchmark users are stuck with inputs of fairly limited size no matter how big their approximation error is. Time constraints also limit the scope of this study because we believe the more detailed behavior space exploration which becomes possible with simulation

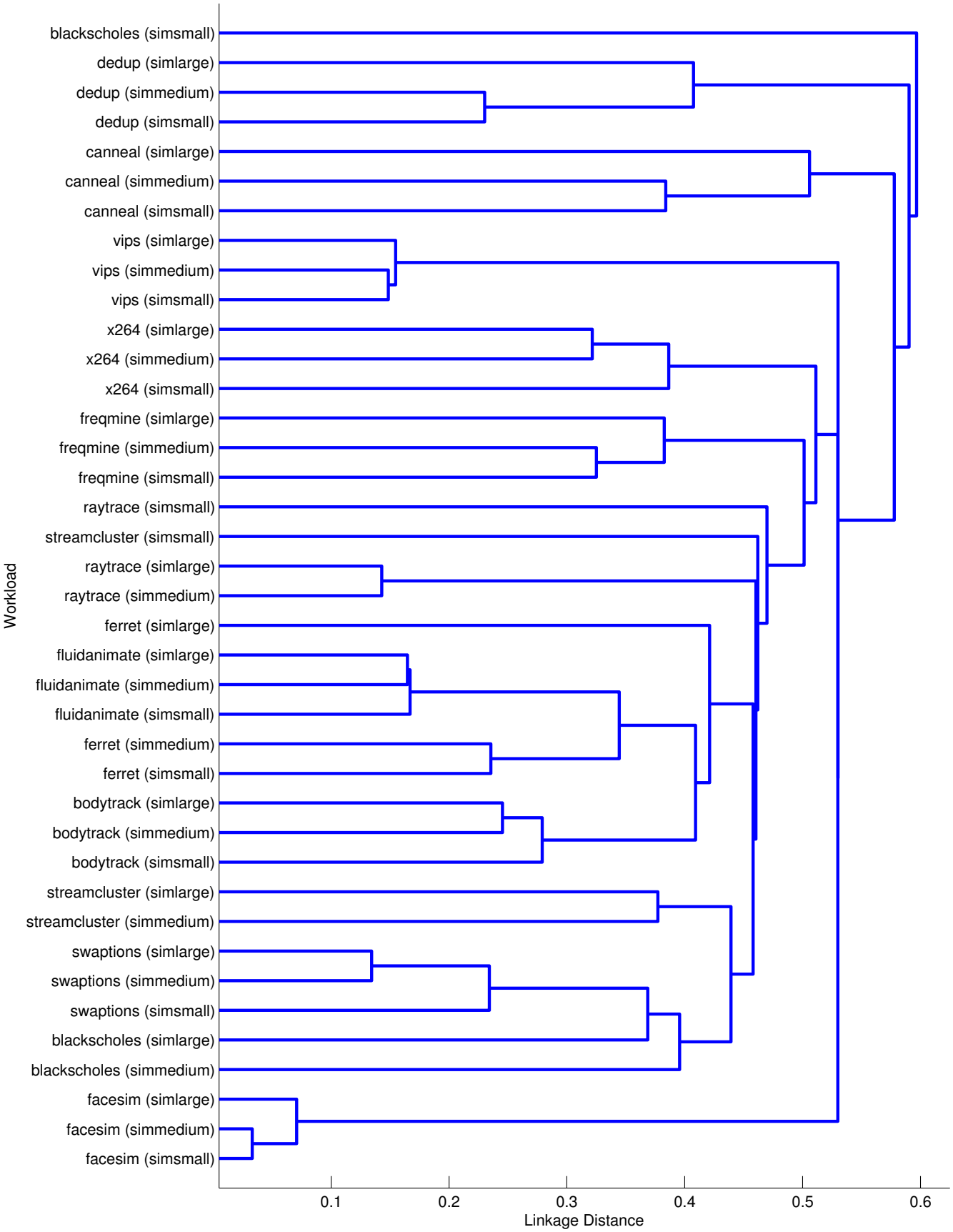

Fig. 2. Fidelity of the PARSEC input sets. The figure shows the similarity of all inputs. The approximation error is the lack of similarity between inputs for the same workload.

is more important than a more accurate reference point which would be feasible with experiments on real machines.

B. Validation Results

We studied the chosen characteristics of the inputs with PCA. The results of this comparison are summarized by the dendrogram in Figure 2.

The dendrogram shows that the inputs of most workloads form their own, complete clusters before they merge with inputs of other programs. That means that the inputs preserve the characteristics of the program with respect to the rest of the suite. These workloads are `dedup`, `canneal`, `vips`, `x264`, `freqmine`, `fluidanimate`, `bodytrack`, `swaptions` and `facesim`.

The inputs for the benchmarks `blackscholes`, `raytrace`, `streamcluster` and `ferret` merge with clusters formed by inputs of other programs before they can form their own, complete cluster. This indicates that the input scaling process skewed the inputs in a way that made the region in the characteristics space that corresponds to the workload overlap with the characteristics space of a different benchmark. This is somewhat less of an issue for `ferret`, which simply overlaps with `fluidanimate` before its inputs can fully merge. However, three inputs belonging to the `simsmall` input set merge significantly later than all other inputs. These are the `simsmall` inputs for `blackscholes`, `raytrace` and `streamcluster`. This indicates that these inputs not only start to be atypical for their workloads, they even become atypical for the entire suite. It is important to emphasize that this increase in diversity is not desirable because it is an artificial byproduct of input scaling which does not represent real-world program behavior. In the case of `streamcluster` the working sets change significantly as its inputs are scaled, as we explained in the last section.

It is also worth mentioning which inputs merge late. While forming their own clusters first, the inputs within the clusters for `dedup`, `canneal`, `x264` and `freqmine` have a distance to each other in the dendrogram which is larger than half of the maximum distance observed between any data points for the entire suite. In these cases there is still a significant amount of difference between inputs for the same workload, even though the inputs as a whole remain characteristic for their program.

The same data is also presented in Figure 3. It shows directly what the distances between the three simulation inputs for each workload are without considering other inputs that might be in the same region of the PCA space. The figure shows the approximation error a_{ij} of the simulation inputs, which is simply the distance from `simlarge` in our case. We have explained that we chose `simlarge` as our reference point because it typically is the best input that is feasible to use for microarchitectural simulations. It is worth mentioning that the data in Figure 3 follows the triangle inequality - the cumulative distance from `simsmall` to `simmedium` and then from `simmedium` to `simlarge` is never less than the distance from `simsmall` to `simlarge`.

The figure shows that the inputs for `bodytrack`, `dedup`, `facesim`, `fluidanimate`, `swaptions` and `vips` have an approximation error that is below average, which means they have a high degree of fidelity. This list includes nearly all benchmarks whose inputs could be scaled with linear input scaling. The inputs with the highest fidelity are the ones for `facesim`. This is not surprising considering that the simulation inputs for that workload have not been scaled at all and are, in fact, identical. The small differences between the inputs that can be observed are caused by background activity on the simulated machine. The inputs for `blackscholes`, `canneal` and `freqmine` have a very high approximation error. These are workloads where linear input scaling has a direct effect on their working sets or which are very difficult to scale.

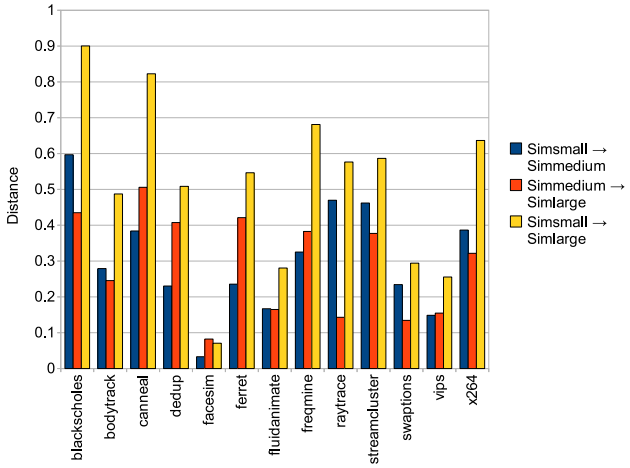

Fig. 3. Approximation error of inputs. The approximation error or dissimilarity between any two pairs of simulation inputs for the same workload is given by the distance between the corresponding points in PCA space.

V. Input Set Selection

PARSEC is one of few benchmark suites which offers multiple scaled-down input versions for each workload that were derived from a single real-world input. This requires users to select appropriate inputs for their simulation study. This section discusses how to make this decision in practice.

As mentioned in Section II-A, designing or selecting an input set is an optimization problem which has the structure of the classical multiple-choice knapsack problem. In practice, the simulation time budget T is often somewhat flexible. Thus, we can simplify the problem to the fractional knapsack problem, for which a simple greedy heuristic based on the benefit-cost ratio (BCR) leads to the optimal solution. With this approach we first determine the optimal order in which to select the inputs, then we implicitly size the simulation time budget so that there is no need to take fractional inputs. We furthermore express the input selection problem as a selection of upgrades over `simsmall` to prevent the pathological case where the greedy heuristic chooses no input at all for a given benchmark program. The cost in our case is the increase in instructions, which is a reasonable approximation of simulation time. The task of the greedy algorithms is thus to maximize error reduction relative to the increase of instructions for each input upgrade over `simsmall`.

More formally, the benefit-cost ratio of upgrading from input k to input l of benchmark i is $\mathrm{BCR}_i(k,l) = -\frac{a_{il}-a_{ik}}{t_{il}-t_{ik}}$. We use the negative value because the metric is derived from the reduction of the approximation error, which is a positive benefit. The values for the variables a_{ik} are simply the distances of the respective inputs from the corresponding reference inputs of the `simlarge` input set, which we give in Figure 3. The values for the cost t_{ik} can be measured directly by executing the workloads with the desired inputs and counting the instructions. We show all relevant BCRs in Figure 4. As can be expected, the data shows diminishing returns for upgrading to a more accurate input set: Making the step from `simsmall` to `simmedium` is always more beneficial than making the step from `simmedium` to `simlarge`.

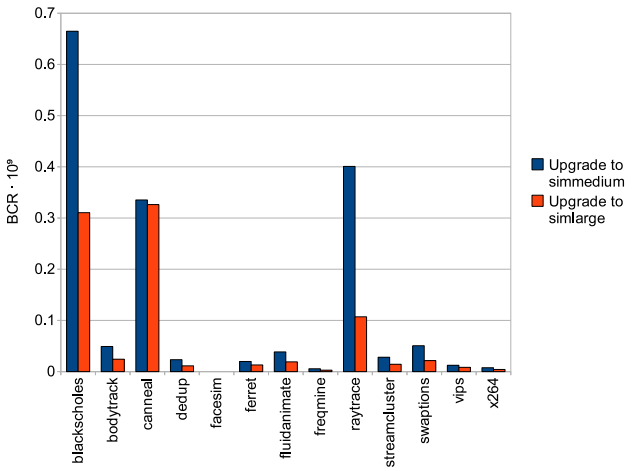

Fig. 4. Benefit-cost ratio (BCR) of using the next larger input. The chart shows the reduction in approximation error relative to the increase of cost in billion instructions. The data for `facesim` had to be omitted due to a division by zero.

As can be seen from the data, if simulation cost is considered there are three obvious candidates which are interesting for an upgrade: Using `simmedium` instead of `simsmall` is highly attractive for the workloads `blackscholes`, `canneal` and `raytrace`. The benchmarks `blackscholes` and `canneal` are furthermore interesting for another upgrade to the `simlarge` inputs, `raytrace` to a much lesser extent. The next tier of workloads with an attractive BCR is formed by `bodytrack`, `dedup`, `ferret`, `fluidanimate`, `streamcluster` and `swaptions`. These workloads also show an increased benefit-cost ratio for an upgrade from `simsmall` to `simmedium`. An interesting fact is that it is more attractive to upgrade `blackscholes`, `canneal` and `raytrace` to `simlarge` first before upgrading any other inputs to `simmedium`. This is primarily due to the fact that the inputs for these programs contain significantly fewer instructions than those of the other benchmarks. Benchmark users should furthermore verify that all selected inputs are used within their scope as defined in Section III-C. If this is not the case it might be necessary to create custom benchmark inputs, which we discuss in the next section.

VI. Customizing Input Sets

The input sets of the PARSEC release are scaled for a wide range of evaluation or simulation cases. However, for specific purposes one may want to consider designing their own input sets based on the methodology proposed in this paper. This section presents a guideline for customizing inputs for more parallelism, larger working sets or higher communication intensity.

A. More Parallelism

The amount of work units available in an input can typically be directly controlled with the linear component of the input. In almost all cases, the amount of parallelism can be increased significantly so that enough concurrency is provided by the input to allow the workload to stress CMPs with hundreds of cores. However, it is important to remember that the amount of work units contained in an input are at best only potential parallelism. It is likely that not all workloads will be able to scale well to processors with hundreds of cores. We describe some technical limitations in Section III-C.

B. Larger Working Sets

It is straightforward to create massive working sets with PARSEC. The standard input sets already provide fairly big working sets, but as mentioned in Section III, linear input scaling was used to aggressively reduce the input sizes of the simulation inputs. This can significantly reduce working set sizes if the outer-most loop has a direct impact on the amount of data that will be reused, which is especially true for benchmarks with unbounded working sets.

The general approach to obtain a larger working set with a PARSEC program is to first use complex input scaling to increase the amount of data that the program keeps in memory and then guaranteeing enough reuse of the data by linear input scaling.

Our experience shows that it is easy to underestimate the impact of increasing working set sizes on execution time. It is known that program execution time grows at least linearly with the program's memory requirements because each item in memory has to be touched at least once. However, most programs use algorithms with complexities higher than linear, which means increasing problem sizes may lead to a prohibitively long execution time. For example, increasing the working set size of an algorithm which runs in $O(N \log N)$ time and requires $O(N)$ memory by a factor of eight will result in a 24-fold increase of execution time. If the simulation of this workload took originally two weeks, it would now take nearly a full year for a single run - more than most researchers are willing to wait.

C. Higher Communication Intensity

The most common approach to increase communication intensity is to reduce the size of the work units for threads while keeping the total amount of work constant. For example, the communication intensity of `fluidanimate` can be increased by reducing the volume of a cell because communication between threads happens at the borders between cells. However, this is not always possible in a straightforward manner because the size of a work unit might be determined by the parallel algorithm or it might be hardwired in the program so that it cannot be chosen easily.

The communication intensity is limited in nearly all lock-based parallel programs. Communication among threads requires synchronization, which is already an expensive operation by itself that can quickly become a bottleneck for the achievable speedup. Programs with high communication intensity are typically limited by a combination of synchronization overhead, lock contention or load imbalance, which means that parallel programs have to be written with the goal to reduce communication to acceptable levels. The PARSEC benchmarks are no exception to this rule.

VII. Related Work

Closely related work falls into three categories [26]: *reduced inputs*, *truncated execution*, and *sampling*.

A popular method to reduce the number of simulated instructions of a benchmark is called *reduced inputs*. The `test` and `train` input sets of the SPEC CPU2006 benchmark suite are sometimes used as a reduced version of its authoritative `ref` input set. MinneSPEC [14] and SPEClite [22] are two alternative input sets for the SPEC CPU2000 suite that provide reduced inputs suitable for simulation. Our work goes beyond previous work by proposing a framework and employing a continuous scaling method to create multiple inputs suitable for performance studies with varying degrees of fidelity and simulation cost.

Truncated execution takes a single block of instructions for simulation, typically from the beginning (or close to the beginning) of the program. This method has been shown to be inaccurate [26].

Sampling is a statistical simulation method that provides an alternative to reduced inputs. It selects small subsets of an instruction stream for detailed simulation [4], [5], [20], [25]. These sampling methods choose brief instruction sequences either randomly or based on some form of behavior analysis. This can happen either offline or during simulation via fast forwarding. Statistically sampled simulation can be efficient and accurate [26], but it requires a sophisticated mechanism built into simulators.

The importance of limiting simulation cost while preserving accuracy has motivated studies that compare sampling with reduced inputs. Haskins et al. concluded that both approaches have their uses [11]. Eeckhout et al. showed that which method was superior depended on the benchmark [7]. Finally, Yi et al. concluded that sampling should generally be preferred over reduced inputs [26]. These comparisons however did not consider that the accuracy of either method is a function of the input size. Our work provides the framework to allow benchmark users to decide for themselves how much accuracy they are willing to give up for faster simulations.

Another approach is statistical simulation [8], [16], [17]. The fundamental concept of statistical simulation is to generate a new, synthetic instruction stream from a benchmark program with the same statistical properties and characteristics. These statistical properties have to be derived by first simulating a workload in sufficient detail. The statistical image obtained that way is then fed to a random instruction trace generator which drives the statistical simulator.

VIII. Conclusions

We have developed a framework to scale input sets for a benchmark suite. Our approach considers that benchmark inputs are approximations of real-world inputs that have varying degrees of fidelity and cost. By employing concepts of mathematical approximation theory and scientific modeling we have provided the methodology to allow benchmark creators and users to reason about the inherent accuracy and cost tradeoffs in a systematic way.

We have shown that the problem of choosing the best input size for a benchmark can be solved in an optimal way by expressing it as the multiple-choice knapsack optimization problem. The inputs can then be selected by standard algorithms so that benchmarking accuracy is maximized for a given time budget. For practical situations the problem can be further simplified so that it can be solved optimally with a simple greedy heuristic.

We have quantified the approximation errors of multiple scaled input sets of the PARSEC benchmark suite and have suggested a sequence of input upgrades for users to achieve higher simulation accuracies for their simulation time budgets. We have also analyzed the constraints of the PARSEC simulation inputs to provide users with the scope of architectures for which the inputs exhibit reasonably accurate behavior.

We have also given guidelines for users to create their own input sets for PARSEC benchmark programs with a scope more fitting for their specific simulation purpose. The proposed scaling model is used to categorize the various input parameters of the PARSEC benchmarks, which gives users a better understanding of the input creation process and its impact on program behavior.

References

[1] M. P. Bailey and W. G. Kemple. The Scientific Method of Choosing Model Fidelity. In *Proceedings of the 24th Conference on Winter Simulation*, pages 791–797, New York, NY, USA, 1992. ACM.

[2] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of the 2008 International Symposium on Workload Characterization*, September 2008.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[4] P. D. Bryan, M. C. Rosier, and T. M. Conte. Reverse State Reconstruction for Sampled Microarchitectural Simulation. *International Symmposium on Performance Analysis of Systems and Software*, pages 190–199, 2007.

[5] T. Conte, M. Ann, H. Kishore, and N. Menezes. Reducing State Loss For Effective Trace Sampling of Superscalar Processors. In *Proceedings of the 1996 International Conference on Computer Design* , pages 468–477, 1996.

[6] K. Dudzinski and S. Walukiewicz. Exact Methods for the Knapsack Problem and its Generalizations. *European Journal of Operations Research*, 28(1):3–21, 1987.

[7] L. Eeckhout, A. Georges, and K. D. Bosschere. Selecting a Reduced but Representative Workload. In *Middleware Benchmarking: Approaches, Results, Experiences. OOSPLA workshop*, 2003.

[8] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. D. Bosschere. Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox. *IEEE Micro*, 23:26–38, 2003.

[9] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the Impact of Input Data Sets on Program Behavior and its Applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2003.

[10] R. Giladi and N. Ahituv. SPEC as a Performance Evaluation Measure. *Computer*, 28(8):33–42, 1995.

[11] J. Haskins, K. Skadron, A. KleinOsowski, and D. J. Lilja. Techniques for Accurate, Accelerated Processor Simulation: Analysis of Reduced Inputs and Sampling. Technical report, Charlottesville, VA, USA, 2002.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[13] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring Benchmark Similarity Using Inherent Program Characteristics. *IEEE Transactions on Computers*, 28(8):33–42, 1995.

[14] A. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1, 2002.

[15] R. M. Nauss. The 0-1 Knapsack Problem with Multiple-Choice Constraints. *European Journal of Operations Research*, 2(2):125–131, 1978.

[16] S. Nussbaum and J. E. Smith. Modeling Superscalar Processors via Statistical Simulation. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.

[17] S. Nussbaum and J. E. Smith. Statistical Simulation of Symmetric Multiprocessor Systems. In *Proceedings of the 35th Annual Simulation Symposium*, page 89, Washington, DC, USA, 2002. IEEE Computer Society.

[18] A. Phansalkar, A. Joshi, and L. K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 412–423, New York, NY, USA, 2007. ACM.

[19] D. Pisinger. A Minimal Algorithm for the Multiple-Choice Knapsack Problem. *European Journal of Operational Research*, 83:394–410, 1994.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large-Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[21] P. Sinha and A. A. Zoltners. The Multiple-Choice Knapsack Problem. *Operations Research*, 27(3):503–515, 1979.

[22] R. Todi. SPEClite: Using Representative Samples to Reduce SPEC CPU2000 Workload. In *Proceedings of the 2001 IEEE International Workshop of Workload Characterization*, pages 15–23, Washington, DC, USA, 2001. IEEE Computer Society.

[23] Vandierendonck, H. and De Bosschere, K. Many Benchmarks Stress the Same Bottlenecks. In *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 57–64, 2 2004.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[25] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–97, 2003.

[26] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and Comparing Prevailing Simulation Techniques. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 266–277, Washington, DC, USA, 2005. IEEE Computer Society.