EVOLUTIONARY BENCHMARK SUBSETTING

TO IMPROVE SIMULATION EFFICIENCY AND RELIEVE BURDENED BENCHMARKING EFFORTS, THIS RESEARCH PROPOSES A SURVIVAL-OF-THE-FITTEST EVOLUTIONARY METHODOLOGY. THE GOAL IS TO SUBSET ANY GIVEN BENCHMARK SUITE BASED ON ITS INHERENT WORKLOAD CHARACTERISTICS, DESIRED WORKLOAD SPACE COVERAGE, AND TOTAL EXECUTION TIME. GIVEN A USER-SPECIFIED WORKLOAD SPACE COVERAGE THRESHOLD, THE PROPOSED TECHNIQUE CAN SYSTEMATICALLY YIELD THE “FITTEST” TIME-EFFICIENT BENCHMARK SUBSET.

Computer architects and system designers have long used benchmark programs that represent real programs to assess performance characteristics and estimate their processor designs’ power consumption. A representative benchmark composition and its consistent usage are extremely important for iteratively evaluating and improving processor performance and obtaining an optimal microarchitecture given the requisite design goal and constraints.

Simulation time has become more and more unwieldy, however, with the rapid development of computing techniques and the expansion of workloads integrated in benchmark suites. Researchers must now coevaluate many proposed architectural enhancements and design options on numerous microarchitectural configurations, resulting in immense design parameters that can take several days or even months of simulation to thoroughly examine on popular benchmark suites.

Thus, it’s impossible and impractical to run certain benchmark suites thoroughly during early design-space exploration. This situation prevents designers from carrying out exhaustive and complete system benchmarking as often as required.

To overcome this hurdle, we propose a biologically inspired, survival-of-the-fittest evolutionary technique to reduce benchmarking time and improve design-space exploration efficiency. Our goal is to introduce an objective benchmark subsetting methodology that can systematically make optimal trade-offs between evaluation efficacy and efficiency. Specifically, the proposed evolutionary technique can subset any given benchmark suite based on inherent workload characteristics, desired workload space coverage, and total execution time. To the best of our knowledge, this work is the first to comprehensively identify and consider these three benchmarking factors and encode them into the fitness function that directs the evolutionary process. (See the “Related Work in Benchmark Evaluation” sidebar for details on other approaches.)

We demonstrate the proposed technique’s usage, efficacy, and efficiency using...
a case study on the SPEC CPU2000 benchmark suite. Our experimental results show that our technique can automatically select an optimal subset of representative programs from SPEC CPU2000 that captures more than 70 percent of inherent workload characteristics of the entire benchmark suite; this takes less than half of the original total execution time (a more than 200 percent speedup). Compared to the prevailing ad hoc and subjective subsetting schemes, the closely matched instructions per cycle (IPC) and energy per instruction (EPI) results between our subset solution and the original benchmark suite demonstrate that our strategy generates a representative and accurate subsetting solution.

**Benchmarking evaluation strategies**

Because of the obstacles to thoroughly evaluating benchmarks, researchers have investigated ways to reduce benchmarking efforts while still providing an accurate and complete evaluation picture. Based on our observation, we classify such strategies (from industry and academia) into three basic categories: reducing the input data set, the number of program phases within an individual benchmark to be evaluated, or the number of benchmark programs to be evaluated by running only a subset of the whole suite on the target design.

The first strategy, which includes MinneSPEC, requires designers to find alternate input data sets somewhere else or modify an existing, large reference input data set. The second strategy, which includes SimPoint and SMARTS, requires that designers have access to a specific simulation infrastructure that provides phase identification and skipping capability, and they must port their benchmark suite to work with such a simulation infrastructure.

This article focuses on the third strategy, **benchmark subsetting**, because it is by far the most convenient; designers can use the provided reference input data sets and their existing tool chain platforms (simulators or real machines). However, it’s also perhaps the most easily abused option because it allows subjective and ad hoc design decisions. For example, designers might discard a program because it takes too long to run, they can’t port it to their tool chain, or it doesn’t provide a favorable evaluation for the particular design they are promoting.

We propose an objective and systematic strategy to subset a given benchmark suite based on the inherent microarchitecture-independent characteristics of the target benchmark programs and their overall execution efficiency. Although most researchers would agree that simulating incomplete benchmark programs can result in a biased evaluation, we found that doing the opposite can also be suboptimal. That is, simulating similar or overlapping programs will increase benchmarking time without providing much additional useful information. An ideal benchmark suite should contain programs that are evenly distributed within the target workload space without carrying any inherent overlapped estimator. In other words, a well-designed representative benchmark should cover as much disjointed workload characteristic space as possible.

Based on this observation, we believe that we can obtain a more time-efficient benchmark suite subset if we can identify the programs that have a limited contribution to the overall workload characteristic space. We remove them according to the desired workload space coverage and the expected benchmarking execution time to amplify the overall simulation efficiency.

Many researchers have explored possible optimization strategies that selectively simulate only a subset of benchmark suites to improve the overall simulation efficiency. Among all these efforts, principal component analysis (PCA) is so far the most widely used basis for benchmark subsetting problems. Such a strategy can help us investigate the similarity among all the programs in a benchmark suite and reduce the number of programs we must simulate based on their inherent program characteristics in the workload space. Yet, it doesn’t account for execution time, which is perhaps the most direct measure for subset-
Related Work in Benchmarking Evaluation

Researchers have explored three main simulation strategies and methodologies to improve simulation efficiency: reducing and modifying input data sets, simulating representative program execution phases, and subsetting benchmark suites. Figure A shows the optimization space and some representative ongoing projects.

Research shows that the most time-consuming increment of simulation time comes from the expansion of input data sets and the extension of application-specific workloads enclosed in benchmark suites. Therefore, it’s easy to imagine that the most efficient method of improving simulation efficiency and reducing simulation time is to compress or truncate input data sets while keeping as many benchmark characteristics as possible. For example, KleinOsowski and Lija created the MinneSPEC reduced input data set for the SPEC CPU2000 benchmarks, and Alameldeen and his colleagues developed reduced input sets for TPC-C based on the metric of transaction throughput.

Among the efforts to reduce the overall simulation time of benchmark suites, SimPoint, which only simulates certain representative program phases rather than the whole program, depicts a new research direction and has been widely used in academia. SimPoint automatically finds a small set of simulation points to represent a program’s complete execution for efficient and accurate simulation.

SMARTS, on the other hand, uses functional warming (continuous updates of the caches and branch predictor during simulation) to keep the key components warm. TurboSMARTS improves SMARTS’s simulation speed by using checkpoints. Recently, Bryan and his colleagues proposed a new warm-up method called Reverse State Reconstruction, which requires no profiling or analysis of skip-region instructions compared to other sampling simulation techniques.

To date, researchers have proposed several approaches to simulate only a subset of benchmark suites selectively. Eeckhout and his colleagues advocated using principal component analysis (PCA) as a statistically rigorous way to subset a benchmark suite, which eliminates the inherent correlation among programs by constructing new, uncorrelated statistical random variables called principal components, and then clustering the benchmarks based on their Euclidean distances from each other. Eeckhout also applied this PCA and clustering methodology to SimPoint’s model of breaking each program’s execution into a set of contiguous, nonoverlapping intervals of 100 million instructions of execution.

Jin and Cheng presented a case study of a new subsetting effort on the emerging ImplantBench benchmark suite using an alternative statistical factor analysis (FA) strategy. Furthermore, Hoste and his colleagues focused on identifying a set of microarchitecture-independent characteristics that provide an accurate picture of the inherent program behavior with PCA and clustering method.

Because the goal of our methodology is to find the most representative and time-efficient subset of programs, we can map this benchmark subsetting problem to a multi-objective global optimization problem that requires us to simultaneously optimize two or more conflicting objectives. We propose a new technique based on genetic algorithm
As an extension of the PCA strategy, Phansalkar and his colleagues paid more attention to the similarity and redundancy of SPEC CPU benchmark suites. They studied the evolution of four generations of CPU benchmark suites and measured the similarity between programs based on their inherent microarchitecture-independent characteristics. Unlike Eeckhout, Phansalkar used a k-means clustering algorithm and Bayesian information criterion (BIC) to cluster the benchmark and input set pairs. Yi and his colleagues proposed using the Plackeet and Burman (P&B) design to characterize performance bottlenecks induced by a benchmark running on a processor. Two benchmarks are similar if the Euclidean distance between their vectors of bottleneck ranking is small.

Indeed, these statistical methods have provided reasonable subsetting strategies to some extent, but none of them account for execution time. The novel benchmark subsetting strategy we propose explores the concept of biologically inspired survival-of-the-fittest evolutionary methodology, capable of subsetting a given benchmark suite by systematically making an balanced trade-off between the inherent workload characteristics, desired workload space coverage, and the execution efficiency. To the best of our knowledge, this is a novel proposal.

References


(GA), an effective and efficient evolutionary approach, to evolve the encoded subset candidates until one subset reaches the highest fitness score. GAs can often rapidly locate good solutions, even in difficult search spaces involving multiple optimization objectives. The downside of GAs is that, if not carefully crafted, the solution will converge toward local optima rather than the problem’s global optimum. Given this, we offer three specific contributions.

First, we propose an evolutionary subsetting methodology that comprehensively considers three significant and synergistic factors: the geometric stability of the workload space, inherent workload charac-
teristics coverage, and execution efficiency. Specifically, we encode them as three tightly integrated fitness facets into the evolutionary process so an optimal combination of benchmarks will emerge as the “fittest” subsetting solution.

Second, we conduct a thorough case study using the SPEC CPU2000 benchmark suite to demonstrate our technique’s usage, efficacy, and efficiency. Given a user-specified workload space coverage, the proposed evolutionary technique can systematically yield the fittest time-efficient benchmark subset compared to the original benchmark and other subsetting approaches. This case study provides a concrete example of how we can amplify benchmarking efficiency to facilitate complex next-generation computer systems design for important and diverse workloads.

Lastly, to prevent the evolutionary process from converging toward local optima, we’ve redesigned a typical GA with these components:

- We carefully formulate the fitness function as a single aggregate objective function that weighs all objectives without bias.
- We encode the workload space’s coverage as a function of the stability of the center of mass and augment it to the fitness function to guide the evolutionary process by smoothing the fitness landscape’s surface. This makes it easier to ascend toward a global optimum.
- We use a mask-based crossover genetic operator on top of the traditionally popular single- or two-point crossover operators to allow a more efficient fitness landscape exploration and make it easier to avoid or escape local optima.

Our strategy is orthogonal, so can be easily combined with techniques from the other directions—reducing input data sets and benchmarking only representative program phases (such as the widely used SimPoint)—to further reduce total benchmarking latency and effort to help meet today’s tight time-to-market constraint. We intend for this technique to be portable and adaptive; it should work with any benchmark suite on any processor the user wants to evaluate. It should also offer an objective benchmark subsetting suggestion that can provide a representative evaluation based on affordable simulation resources and the targeted workload characteristics coverage.

Workload analysis

When facing the task of selecting the workload subset that’s most representative of the entire benchmark suite, it’s imperative to first decide what characteristics we want our subset to represent. There are many ways to characterize a workload’s behavior and nature, just as there are many different ways to describe a person or an object. Thus, every evaluation metric we use to characterize a workload should be prudently selected based on its contribution and significance in an assessment of the entire benchmark suite. Another nonnegligible criterion is whether such characteristics can facilitate or hurt our ability to perform data analysis.

Benchmark programs could be characterized using microarchitecture-independent or microarchitecture-dependent characteristics. The former is important for comparisons between programs based on their inherent properties regardless of a particular machine configuration’s specific features. On the other hand, the latter plays a vital role in evaluating a particular machine configuration or architecture. Microarchitecture-independent characteristics are more significant when conducting a general analysis and investigating a certain benchmark suite’s inherent behaviors, which are expected to be isolated from the features of particular microarchitectural components.

In this study, we choose the microarchitecture-independent characteristics that Phansalkar and his colleagues identify, as Table 1 shows, which are a subset of all possible microarchitecture-independent characteristics that researchers are usually concerned about, but they can easily be extended to contain other metrics for evaluating any particular or emerging computing system.
Benchmark selection criteria

Generally, the SPEC benchmark suites are the most popular and widely used commercial evaluation tool. To date, the SPEC CPU series benchmark suite has become an almost incontestable source for benchmarking general-purpose computing systems. Thus, we choose the most popular and widely used SPEC benchmark suite to demonstrate the efficacy and effectiveness of our methodology framework.

The main purpose of this study, however, is to deliver a systematic benchmark sub-setting strategy that is portable and adaptive. To facilitate an objective comparison with other state-of-the-art subsetting techniques, we report our results based on the SPEC CPU2000 benchmark suite as a case study because there are more mature and complete results published from other research groups.

Table 2 shows the specific benchmark programs and their dynamic instruction counts.

Workload space geometric analysis

In this study, we use 29 characteristics to represent each of the 22 programs in the SPEC benchmark suite. In practice, the user can choose to include even more

---

**Table 1. Workload characterization.**

<table>
<thead>
<tr>
<th>Microarchitecture-independent metric categories</th>
<th>Number of metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction mix</td>
<td>3</td>
</tr>
<tr>
<td>Dynamic basic block size</td>
<td>1</td>
</tr>
<tr>
<td>Branch directions</td>
<td>1</td>
</tr>
<tr>
<td>Taken branches</td>
<td>1</td>
</tr>
<tr>
<td>Forward-taken branches</td>
<td>2</td>
</tr>
<tr>
<td>Dependency distance</td>
<td>7</td>
</tr>
<tr>
<td>Data temporal locality</td>
<td>4</td>
</tr>
<tr>
<td>Data spatial locality</td>
<td>3</td>
</tr>
<tr>
<td>Instruction temporal locality</td>
<td>4</td>
</tr>
<tr>
<td>Instruction spatial locality</td>
<td>3</td>
</tr>
<tr>
<td>Total metrics</td>
<td>29</td>
</tr>
</tbody>
</table>

**Table 2. SPEC CPU2000 benchmarks.**

<table>
<thead>
<tr>
<th>Program</th>
<th>Input</th>
<th>INT/FP*</th>
<th>Dynamic instruction count (in billions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>input.graphic</td>
<td>INT</td>
<td>103.7</td>
</tr>
<tr>
<td>175.vpr</td>
<td>route</td>
<td>INT</td>
<td>84.1</td>
</tr>
<tr>
<td>176.gcc</td>
<td>166.i</td>
<td>INT</td>
<td>46.9</td>
</tr>
<tr>
<td>181.mcf</td>
<td>mcf inp.in</td>
<td>INT</td>
<td>61.8</td>
</tr>
<tr>
<td>186.crafty</td>
<td>crafty.in</td>
<td>INT</td>
<td>191.8</td>
</tr>
<tr>
<td>197.parser</td>
<td>ref</td>
<td>INT</td>
<td>546.7</td>
</tr>
<tr>
<td>252.eon</td>
<td>cook</td>
<td>INT</td>
<td>80.6</td>
</tr>
<tr>
<td>255.vortex</td>
<td>lendian.l.raw</td>
<td>INT</td>
<td>118.9</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>input.graphic</td>
<td>INT</td>
<td>128.7</td>
</tr>
<tr>
<td>300.twolf</td>
<td>ref</td>
<td>INT</td>
<td>346.4</td>
</tr>
<tr>
<td>168.wupwise</td>
<td>wupwise.in</td>
<td>FP</td>
<td>349.6</td>
</tr>
<tr>
<td>171.swim</td>
<td>swim.in</td>
<td>FP</td>
<td>225.8</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>mgrid.in</td>
<td>FP</td>
<td>419.1</td>
</tr>
<tr>
<td>173.applu</td>
<td>applu.in</td>
<td>FP</td>
<td>223.8</td>
</tr>
<tr>
<td>177.mesa</td>
<td>mesa.in</td>
<td>FP</td>
<td>141.9</td>
</tr>
<tr>
<td>178.galgel</td>
<td>galgel.in</td>
<td>FP</td>
<td>409.3</td>
</tr>
<tr>
<td>179.art</td>
<td>c746hel.in</td>
<td>FP</td>
<td>45.0</td>
</tr>
<tr>
<td>183.equake</td>
<td>inp.in</td>
<td>FP</td>
<td>131.5</td>
</tr>
<tr>
<td>188.ammp</td>
<td>ammp.in</td>
<td>FP</td>
<td>326.5</td>
</tr>
<tr>
<td>189.lucas</td>
<td>lucas2.in</td>
<td>FP</td>
<td>142.4</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>fma3d.in</td>
<td>FP</td>
<td>268.3</td>
</tr>
<tr>
<td>301.apsi</td>
<td>aps.in</td>
<td>FP</td>
<td>347.9</td>
</tr>
</tbody>
</table>

* Integer benchmarks (INT); Floating-point benchmarks (FP).
characteristics, depending on the design goal and target applications. To systematically and consistently evaluate benchmark programs represented by such high-dimensional characteristics, which is at an abstraction level too high to undergo a formal optimization and search procedure, we propose constructing a geometric model of a workload characteristic space by using concepts from physics and mathematics. By transforming the construction of this workload space and mapping benchmark programs to it, we can more easily formulate the task of benchmark subsetting into a systematic optimization problem.

Center of mass

In physics, the center of mass (or the center of gravity) of a system of particles is a specific point at which, for many purposes, the system’s mass behaves as if it was concentrated. The center of mass is a function of the positions and masses of the particles comprised by the system. In the case of a rigid body, this position is fixed in relation to the object but not necessarily in contact with it. In the case of a loose distribution of masses in free space, such as shots from a shotgun, that position is a point in space among them that might not correspond to the position of any individual mass. For a solid body, it’s often possible to replace the body’s entire mass with a mass point equal to that of the body’s mass.

The center of mass for a system of independently moving particles is useful in analyzing the interactions and overall behaviors among the system particles. A body’s center of mass doesn’t always coincide with its intuitive geometric center. The center of mass $R$ of a system of particles is defined as the average of their positions $r$ weighted by their masses $m_i$:

$$ R = \left( \frac{\sum m_i r_i}{\sum m_i} \right) \quad (1) $$

That is, for a system of $n$ particles with a total mass of $M$ in 3D space, the center of mass is:

$$ x_{cm} = \frac{m_1x_1 + m_2x_2 + \ldots + m_nx_n}{m_1 + m_2 + \ldots + m_n} $$

$$ y_{cm} = \frac{m_1y_1 + m_2y_2 + \ldots + m_ny_n}{m_1 + m_2 + \ldots + m_n} $$

$$ z_{cm} = \frac{m_1z_1 + m_2z_2 + \ldots + m_nz_n}{m_1 + m_2 + \ldots + m_n} $$

For the SPEC CPU2000 benchmark suite, we consider 29 microarchitecture-independent characteristics in this study (see Table 1). That is, we associate each of the 22 benchmark programs with 29 variables, which could be regarded as a system of particles $X$ distributed in a 29-dimensional space. However, such a particle system isn’t homogeneous due to its inherent diversified behaviors for benchmarking efficiency.

One of the properties that researchers are always most concerned with is the execution time of individual benchmark programs. Correspondingly, reducing the overall execution time on benchmarking processors for target workloads has become one of computer architects’ top priorities. Thus, we can view the execution time of individual benchmark programs as the unbiased weight for each particle (benchmark program). Because the idiosyncrasies of a particular microarchitecture implementation or a specific machine configuration could vary and bias the actual execution time—that is, machine-cycle or wall-clock time—we use a benchmark program’s dynamic instruction count to approximate its machine-independent execution efficiency.

Thus, we obtain normalized proportions of dynamic instruction counts of individual programs over the whole benchmark suite as the masses of particles that comprise the whole benchmark suite system. Such a particle system’s center of mass $X_{cm}$ is the average of their positions $x_i$ weighted by their masses $m_i$: 
In mathematics, the convex hull (or convex envelope) for a set of points $X$ is the minimal convex set containing $X$. We can prove the existence of the convex hull of a set $X$ by the fact that $X$ is contained in at least one convex set (the whole space $V$, for example) and any intersection of convex sets containing $X$ is also a convex set containing $X$. It is then clear that the convex hull is the intersection of all convex sets containing $X$. More directly, we can constructively describe the convex hull of $X$ as the set of convex combinations of points from $X$—that is, the set of points of the form $\sum_{i=1}^{k} a_i x_i$, where $k$ is an arbitrary natural number, the number $a_i$ is nonnegative and sums to 1, and the points $x_i$ are in $X$. The convex hull $H_{\text{convex}}(X)$ of set $X$ is

$$H_{\text{convex}}(X) = \left\{ \sum_{i=1}^{k} a_i x_i | x_i \in X, a_i \in \mathbb{R}, a_i \geq 0, \sum_{i=1}^{k} a_i = 1, i = 1, 2, \ldots \right\}$$

(4)

In fact, if $X$ is a subset of an $N$-dimensional vector space, sums of up to $N + 1$ points are sufficient in this definition. This is equivalent to saying that the convex hull of $X$ is the union of all simplexes with at most $N + 1$ vertices from $X$, which is known as Carathéodory’s theorem. The convex hull of finite sets of points and other geometrical objects in a 2D plane or 3D space are special cases of practical importance. Figure 1 shows an example of a 3D convex hull.

![Figure 1. Example of a 3D convex hull.](image)

When one of the 22 points is removed, we denote the convex hull of those remaining points as $H_{\text{convex}}(X_{22} - x_i)$, $i = 1, 2, \ldots, 22$. To perform benchmark subsetting while keeping as much diversity and representative characteristics of benchmark programs as possible in the $H_{\text{convex}}$ workload space, we remove the program that contributes the least to the overall convex hull space—that is, $\min\{H_{\text{convex}}(X_{22}) - H_{\text{convex}}(X_{22} - x_i)\}$. Then, the new convex hull of the remaining 21 points is

$$H_{\text{convex}}(X_{21}) = H_{\text{convex}}(X_{22}) - \min\{H_{\text{convex}}(X_{22}) - H_{\text{convex}}(X_{22} - x_i)\};$$

$$H_{\text{convex}}(X_{21}) = \left\{ \sum_{i=1}^{k} a_i x_i | x_i \in X_{21}, a_i \in \mathbb{R}, a_i \geq 0, \sum_{i=1}^{k} a_i = 1, i = 1, 2, \ldots \right\}$$

(5)

Following a similar procedure, we can slice the entire convex hull of the SPEC CPU2000 workload space one by one as follows:

$$H_{\text{convex}}(X_{j-1}) = H_{\text{convex}}(X_j) - \min\{H_{\text{convex}}(X_j) - H_{\text{convex}}(X_j - x_i)\};$$

$$H_{\text{convex}}(X_{j-1}) = \left\{ \sum_{i=1}^{k} a_i x_i | x_i \in X_{j-1}, a_i \in \mathbb{R}, a_i \geq 0, \sum_{i=1}^{k} a_i = 1, i = 1, 2, \ldots \right\}$$

(6)

However, accurately assessing convex hulls in a multidimensional space is a complex, sometimes infeasible process. An alternative
is to evaluate convex hulls via their diameters, which are the pair of points a maximum distance apart. The diameter will always be the distance between two points on the convex hull. The $O(n \log n)$ algorithm for computing diameter proceeds by first constructing the convex hull, and then for each hull vertex finding which other hull vertex is farther away from it. Although we’ve been able to evaluate the convex hulls more accurately using such a diameter scheme, it’s still more computationally demanding than the fast, efficient way we need. Therefore, we need to explore faster approximate evaluation criteria.

Based on our discussion so far, we propose a radius-based approximate expression for evaluating convex hulls. (A radius is the distance of each point from the geometric center.) In this study, all benchmark programs could be mapped into an $N$-dimensional workload space, where $N$ is the number of characteristics the researchers identified to evaluate the benchmarking performance. Given a priori judgment, a program’s larger radius from the geometric center indicates its behavior is more distinct from others, and thus, a more significant representative in the benchmark suite. Although this radius-based approximation can’t perfectly depict the overall geometric idiosyncrasies of the multidimensional convex hull, it is a computationally efficient and effective heuristic for evaluating the inherent behaviors of program points constituting the workload convex hull.

Thus, we can express the overall workload convex hull as

$$H_{\text{convex}}(X_{\text{SPEC}}) = \sum_{i=1}^{22} (x_i - \bar{X})$$  \hspace{1cm} (7)

where $\bar{X}$ is the geometric center of all benchmark programs in this workload space. Given that some programs are abandoned from the new subset, the corresponding convex hull of the new subset is

$$H_{\text{convex}}(X_{\text{subset}}) = \left\{ \sum_{i=1}^{k} (x_i - \bar{X}) | x_i \in X_{\text{subset}} \right\}$$  \hspace{1cm} (8)

**Evolutionary design**

According to Charles Darwin’s theory of evolution, all species of life have evolved over time from one or a few common ancestors through natural selection, which is the process by which favorable heritable traits become more common in successive generations of a population of reproducing organisms, and unfavorable heritable traits become less common. Although natural selection acts on individuals, its average effect on all individuals with a particular genotype corresponds to the fitness of that genotype. On average, very low fitness genotypes cause their bearers to have few or no offspring.

**Evolutionary genetic algorithm**

The evolutionary GA is a generic population-based metaheuristic optimization algorithm that emulates the paradigm of biological evolution obeying Darwin’s theory. GAs use some mechanisms inspired by biological evolution: reproduction, mutation, recombination, and selection. In the most common GA, a population is created with a group of random individuals, which are then evaluated. The user defines the evaluation function and gives the individuals a score based on how well they perform at a given task. Two individuals are then selected based on their fitness; the higher the fitness, the higher the chance of being selected. These individuals then “reproduce” to create one or more offspring, after which the offspring are mutated randomly. This continues until we find a suitable solution or a certain number of generations have passed.

**Chromosome encoding**

One important problem in applying a GA is finding a suitable encoding scheme to map the problem domain to a chromosome. Thus, the first task is to find a suitable chromosome representation for our benchmark subsetting problem. We use a vector of symbols of length 22 (one bit per program in the SPEC CPU2000 suite) where the symbol (or *allele*, with the possible values $\{0, 1\}$) at the $i$th position (which is called a *gene*) controls whether we retain or abandon the $i$th benchmark
program. So, for the following chromosome,

\[
(110100001...110000)
\]

Number of programs

from left to right, the first and the second program are chosen in the potential subset
ing scheme, the third program is discarded, the fourth program is chosen, and so on.

Such chromosome encoding offers a direct indication for the possible benchmark suite subset
ing solutions, which will become the individuals in the whole evolution
ary chain. After we’ve determined a suitable encoding scheme, we then select
our genetic operators.

Genetic operators

The chromosomal crossover is the process by which two paired-up chromosomes exchange some portion of their DNA. In this study, we apply both single- and two-point crossover (see Figures 2a and 2b). We also propose to use a mask-based crossover (see Figure 2c), where a randomly generated mask sequence controls the exchange of genes.

Once a pair of chromosomes has been selected, crossover can take place to produce offspring. A crossover probability of 1.0 indicates that selected chromosomes will always be crossed over during reproduction—that is, there are no survivors. However, empirical studies have shown that we can achieve better results with a crossover probability of between 0.65 and 0.85, which implies that the probability of a selected chromosome surviving to the next generation unchanged (except for any changes arising from mutation) ranges from 0.35 to 0.15. Thus, we make crossover probability an adjustable parameter and assume a value of 0.75 in our experiments.

If we only use the crossover operator to produce offspring, one problem might arise: if all the chromosomes in the initial population have the same value at a particular position, then all future offspring will definitely have this value at this position. To overcome this undesirable situation, our design also uses a mutation operator. It attempts to introduce some random alterna
tion of the genes by flipping their values—for example, 0 becomes 1 and vice versa. This typically occurs infrequently, so mutation is on the order of one bit change for approximately every thousand generations. As with the crossover probability, we also present a mutation probability, which is an adjustable parameter that assumes a value of 0.01. We implement this mutation operation by checking each bit in each chromosome for possible mutation by generating a random number between zero and one. If this number is less than or equal to the given mutation probability (for example, the 0.01 we set), then the bit value is changed. Otherwise, we leave the bit alone.

Fitness function

A fitness function quantifies the optimality of a solution (that is, a chromosome) in a

---

*Figure 2. Crossover operators: one-point crossover (a), two-point crossover (b), and mask-point crossover (c).*
genetic algorithm to rank a particular chromosome against all other chromosomes. Optimal chromosomes, or at least chromosomes that are more optimal, are allowed to breed and mix their data sets to produce a new generation that will be hopefully even better.

For the purpose of identifying an optimal subsetting strategy, we encode three optimization facets—the inherent workload characteristics, the physical geometric stability of the covered workload space, and the execution efficiency—into the fitness function to direct the evolutionary process. This trifacet fitness function provides an objective and balanced metric to evaluate the quality and suitability of chromosomes (in this case, subsetting solutions) over generations during the whole evolutionary procedure.

We can best explain our rationale and motivation for proposing this trifacet fitness function with three a priori lemmas.

**Lemma 1.** In a mature benchmark program suite, an optimal benchmark subset, which is most closely equivalent or comparable to the whole benchmark suite, should have the most physical geometric stability with respect to the original workload space—that is, the least aberrance of the center of mass.

\[
\min \{|X_{cm}(all) - X_{cm}(subset)|\}
\]

**Lemma 2.** An optimal benchmark subset, which is most closely equivalent or comparable to the whole benchmark suite, should keep the most inherent behaviors and characteristics of the original workload space. That is, it should maximize the portion of the convex hull constructed by program points in the new subset over the one enveloped by the whole benchmark suite.

\[
\max \left\{ \frac{H_{convex}(X_{subset})}{H_{convex}(X_{all})} \right\}
\]

**Lemma 3.** An optimal benchmark subset, which is most closely equivalent or comparable to the whole benchmark suite, should be the most time-efficient subset. That is, it should minimize the portion of the execution time consumed by the programs in the new subset over the time spent by the whole benchmark suite.

\[
\min \left\{ \frac{\text{Execution time}_{subset}}{\text{Execution time}_{all}} \right\}
\]

Fusing the synergistic objectives of these three lemmas together gives us one global optimization objective: minimizing the fitness-function value. We can formally formulate this fitness measure:

\[
\text{Fitness} = |X_{cm}(all) - X_{cm}(subset)| \times \left\{ \frac{\text{Execution time}_{subset}}{\text{Execution time}_{all}} \right\} \times \frac{H_{convex}(X_{all})}{H_{convex}(X_{subset})}
\]

Because we consider many heterogeneous characteristics in this study that will generate data with up to orders of magnitude difference, we first preprocess and then statistically normalize all workload characterization data. That is, we find the normal distribution with the mean of each program characteristic zero, and its variance one. The individual with chromosome encodings that are all zeros (that is, none of program is chosen) is excluded for fitness evaluation because it will bring a meaningless workload space coverage, causing the \( H_{convex}(X_{subset}) \) to equal zero. Similarly, the individual with chromosome encodings that are all ones (all of programs are chosen) is also excluded for fitness evaluation because it definitely has the best workload space coverage, but it isn’t what we want to achieve in this study.

**Evolutionary workflow**

Based on the specific components we’ve already discussed, the entire evolutionary benchmark subsetting workflow consists of many successive iterations of an evolutionary process. At each generation, individuals with a high fitness are selected, and the chromosomes of the selected individuals are then recombined and subjected to small
mutations to produce the next generation. Formally, we can formulate this evolutionary subsetting workflow into the following algorithm, where the population at the $t$th generation is designated by $S_k(t)$—where $S_k$ is the chromosome of individuals, $k = 1, \ldots, n$, and $n$ is a population size:

1. Create a random initial population $S_k(0)$ containing $k$ individuals with chromosome encodings initialized to random values.
2. Evaluate the fitness function $f(S_k)$ of each individual $S_k$ in the population $S_k(t)$ based on the workload space coverage threshold specified by the user.
3. Select the individuals $S_k$ according to their fitness-function values ($f(S_k)$) and apply genetic operators (crossovers and mutations) to selected chromosomes to generate the offspring population $S_k(t + 1)$. (In this study, we regard the smallest value of fitness function as the conceptually highest fitness.)
4. Repeat Steps 2 and 3 for $t = 0, 1, 2, \ldots$, until satisfying some convergence criterion (the maximum fitness in the population ceases to increase, or $t$ reaches a certain value).
5. Finally, translate the optimal chromosome encoding schemes that have the maximum fitness among all generations into the benchmark suite subsetting solutions.

Based on the subsetting result from Step 5, we obtain a clear global optimal solution best balanced between the inherent workload characteristics, workload space coverage, and execution efficiency, and we can use it to faithfully represent the original whole benchmark suite.

**Subsetting SPEC CPU2000**

Following the workflow and experimental procedure we just described, we performed a case study and evolutionary analysis using the SPEC CPU2000 benchmark suite. In our experiments, we used an initial parent population size of 30 and a constant mutation rate of 0.01. To investigate the applicability of GA-based evolutionary subsetting, we used incremental workload space coverage ratios from 10 to 80 percent as thresholds in our algorithm—that is, any optimal subsetting solution our algorithm yielded had to reach or exceed the desired workload space coverage. We excluded a 90 percent workload space coverage ratio because of its extremely high similarity with the entire benchmark suite in evaluating behaviors and performance.

Figure 3 depicts the complete evolutionary processes for each of eight workload space coverage thresholds. The mean fitness function (MFF) values converge to a stable status, which is also their optimal status, within 1,000 generations. Among all eight evolutionary chains, most of them could evolve into an optimal generation from the randomly chosen population of parents within 100 generations.

For those evolutionary chains of higher workload space coverage threshold (that is, when the threshold is 70 or 80 percent), the evolutionary process becomes a little bumpy and prolonged. To accelerate the evolution process, we assumed the fitness-function value was positive infinity when the workload space coverage of a certain generation didn’t reach the threshold. We also observed that finding an optimal population satisfying the coverage threshold becomes more difficult as the threshold gradually increases. Figure 3 clearly presents this trend. For the coverage threshold of 80 percent, the qualified population can’t be found, and its corresponding fitness function value can’t converge, until hundreds of generations of evolution elapse. Moreover, the lower the coverage threshold is, the lower the fitness-function value we could obtain. That is, looser threshold requirements can provide more flexibility to choose programs constituting an optimal benchmark subset.

Figure 4 presents the specific evolutionary results for eight evolutionary chains with different workload space coverage threshold requirements. The “—*—” line gives the actual workload space coverage, which is always greater than or equal to the thresholds specified. The “—o—” line gives the ratio of the number of programs in the...
subset to the total number of programs in the benchmark suite. The “–x–” line illustrates the ratio of the total time it takes to complete executing the overall benchmark subset to the total time needed for executing the original whole benchmark suite. The figure shows that our subsetting schemes could provide better benchmark program usage efficiency compared to the workload space coverage they represent. More notably, our subsetting schemes can present considerable workload space coverage using considerably less execution time. For example, in the case of a 40 percent coverage threshold, the proposed evolutionary subsetting technique only requires 22.9 percent of the original total execution time to achieve a remarkable 40.9 percent of the total workload space coverage. This demonstrates that its efficacy in amplifying the total benchmarking efficiency is indeed promising.

Table 3 shows all eight subsetting schemes as well as their actual workload space coverage and execution time ratios.

**Performance evaluation**

To assess our proposed technique’s subsetting performance, we compared the subset’s actual instructions per cycle (IPC) and energy per instruction (EPI) to that of the original benchmark suite. Without losing generality, we assumed two distinct microarchitecture configurations. The baseline microarchitecture configuration is an eight-way superscalar model that represents a processor in the current technology generation. We also included a 16-way superscalar configuration to reflect an aggressive future design point. This configuration has a wider data path, larger out-of-order window, and larger caches to test the effects of an enlarged state set. We obtained the individual IPC and EPI values of each program in the SPEC CPU2000 benchmark suite for both microarchitecture configurations from a previous work, which also contains the specific details of these two machine configurations.

Figure 5 shows the average IPC calculated for each particular subset with workload space coverage ratios from 10 to 80 percent and the IPC for the entire SPEC CPU2000 suite. For the eight-way machine, the average IPCs of benchmark subsets with workload space coverage ratios greater than or equal to 40 percent are very closely
matched with the IPC of the entire benchmark suite: the largest variation is 10.7 percent, while all the others were less than 5 percent. When the workload space coverage ratio declined to less than 40 percent, the average IPC variation becomes more noticeable, with the biggest variation reaching 57 percent. The 16-way machine configuration produced similar trends.

Figure 6 presents EPI results for the benchmark subsets. As with the IPC comparisons, we observed that the average EPIs for the entire benchmark suite and those subsets with workload space coverage

Table 3. SPEC CPU2000 subsetting results.

<table>
<thead>
<tr>
<th>Threshold (%)</th>
<th>SPEC CPU2000 benchmark programs</th>
<th>Number of programs</th>
<th>Workload space coverage ratio (%)</th>
<th>Elapsed time ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>gcc, eon</td>
<td>2</td>
<td>10.35</td>
<td>2.69</td>
</tr>
<tr>
<td>20</td>
<td>mcf, vortex, swim, art</td>
<td>4</td>
<td>20.50</td>
<td>9.52</td>
</tr>
<tr>
<td>30</td>
<td>gcc, mcf, vortex, vpr, swim, art</td>
<td>6</td>
<td>30.24</td>
<td>12.29</td>
</tr>
<tr>
<td>40</td>
<td>vpr, gcc, crafty, eon, bzip2, mesa, art, lucas, applu</td>
<td>9</td>
<td>40.93</td>
<td>22.89</td>
</tr>
<tr>
<td>50</td>
<td>vpr, mcf, crafty, eon, vortex, bzip2, swim, mesa, art, equake, applu</td>
<td>11</td>
<td>51.30</td>
<td>30.48</td>
</tr>
<tr>
<td>60</td>
<td>gzip, gcc, mcf, eon, vortex, twolf, swim, art, ammp, mesa, fma3d, applu</td>
<td>12</td>
<td>60.53</td>
<td>41.98</td>
</tr>
<tr>
<td>70</td>
<td>gzip, vpr, gcc, mcf, vortex, bzip2, twolf, swim, art, equake, ammp, mesa, fma3d, applu</td>
<td>14</td>
<td>70.05</td>
<td>47.54</td>
</tr>
<tr>
<td>80</td>
<td>gzip, vpr, gcc, crafty, eon, vortex, bzip2, twolf, wupwise, mgrid, mesa, galgel, art, ammp, lucas, applu</td>
<td>17</td>
<td>82.10</td>
<td>67.93</td>
</tr>
</tbody>
</table>
ratio greater than 40 percent were comparable: the largest variation was 8.7 percent, while all the others were less than 5 percent. As the workload space coverage ratio went below 40 percent, the worst variation of average EPI was approximately 71.3 percent.

We compared our solutions with results obtained from prior subsetting techniques to demonstrate the improvement of the proposed evolutionary subsetting scheme in simulation efficiency. Because of the importance of this problem, many researchers have explored possible optimization strategies that simulate only a subset of benchmark suites selectively. Among all these efforts, that PCA-based subsetting strategy is a widely used methodology for benchmark subsetting problems. For example, Joshi and colleagues applied such a PCA-based method to the SPEC CPU2000 benchmark suite and proposed an eight-cluster subsetting solution based on 29 microarchitecture-independent characteristics. We chose their work as a representative example to compare our method to a PCA-based approach. We calculated the corresponding workload space coverage and total elapsed time of the Phansalkar eight-program subsetting solution, and compared it to one subset scheme based on the proposed subsetting strategy. The goal was to show that our methodology yields a more time-efficient and representative subset using the same workload characteristics on the same benchmark suite.

Table 4 illustrates the subsetting solutions, their workload space coverage, and the total runtime. We can see that to implement the comparable workload space coverage, our approach selects a similar number of programs compared to PCA, but our subsetting solution’s total execution time is far less. This comparison validates the efficiency of our proposed scheme.

Although such PCA-based strategies can identify the similarity among all programs in the benchmark suite and reduce the number of programs that must be simulated based on their inherent program characteristics in the workload space, they neglect a far more relevant factor: execution time. Our proposed evolutionary approach, on the other hand, can explicitly co-optimize the total execution time given the desired workload space coverage, and vice versa.

Looking forward, we plan to enhance our evolutionary simulation strategy by extending the workload design space, including exploring more program characteristics, such as thread-level parallelism, loop profiles, and caching behaviors. We will also investigate program behaviors on more
architectures, such as multicore processors, dynamic voltage- and frequency-scaling machines (DVFS), and dynamic cache-resizing microarchitectures.

References


Table 4. Comparison of SPEC CPU2000 subsetting results.

<table>
<thead>
<tr>
<th>Methods</th>
<th>SPEC CPU2000 benchmark programs</th>
<th>Number of programs</th>
<th>Workload space coverage ratio (%)</th>
<th>Elapsed time ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCA-based subsetting</td>
<td>gcc, crafty, bzip2, applu, mesa, mcf, twolf, fma3d</td>
<td>8</td>
<td>40.28</td>
<td>29.73</td>
</tr>
<tr>
<td>Evolutionary subsetting</td>
<td>gcc, crafty, bzip2, applu, mesa, art, lucas, vpr, eon</td>
<td>9</td>
<td>40.93</td>
<td>22.89</td>
</tr>
</tbody>
</table>

Zhanpeng Jin is a PhD student in the Department of Electrical and Computer Engineering at the University of Pittsburgh. His research interests include computer architecture, biomedical and bioimplantable computing systems, ultralow-power energy-efficient processors, VLSI and ASIC system design, and mathematical modeling. Jin received his MS in computer science from the Northwestern Polytechnical University in China. He is a student member of the IEEE and the IEEE Computer Society. Allen C. Cheng is an assistant professor in the Department of Electrical and Computer Engineering, the Department of Computer Science, and the Department of Neurological Surgery at the University of Pittsburgh. He also serves as the director of the Advanced Computing Technology (ACT) Laboratory at University of Pittsburgh. His research interests are at the interdisciplinary confluence of computer engineering, computer science, electrical engineering, neural engineering, biomedical engineering, and medicine. Cheng received his PhD in computer science and engineering from the University of Michigan at Ann Arbor. He is a member of the IEEE, the ACM, and AAAS.

Direct questions and comments about this article to Zhanpeng Jin at the Dept. of Electrical and Computer Eng., Univ. of Pittsburgh, Pittsburgh, PA 16261; zhj6@pitt.edu.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/csdl.