Universidad de Alcalá

# Reduced input data sets selection for SPEC CPUint2006

Technical Report TR-HPC-02-2009

Virginia Escuder, Rafael Rico

*Department of Computer Engineering, Universidad de Alcalá, Spain*

April 2009

## Index

# Abstract

SPEC CPU benchmark suite has become the most frequently used suite for computer architecture research. The workload is designed to stress the hardware of the machines for the next generation. Consequently, the executed instruction count has been considerably increased in the SPEC CPU2006, compared to the previous suites. But this fact results in prohibitive experimentation time or resources requirements for research when using simulation techniques or working in embedded systems development.

In the Literature has been described a wide range of approaches for reducing the workloads in experimental environments. In this work an extensive revision of them is offered.

Moreover, a detailed analysis of the influence of input data sets in the workload of the CPUint2006 suite is presented. As a result, a suggestion of alternative workload with 2 orders of magnitude dynamic executed instructions count lower than test workload is proposed and characterized.

The aim of our work is to help researchers finding a representative set of workloads for the SPEC CPUint2006 programs to use in their experiments whenever they have to discard using the reference workload due to time or resource constraints.

# 1. SPEC benchmarks

SPEC is the acronym for Standard Performance Evaluation Corporation, a non-profit organization whose purpose is to define and maintain a set of standard benchmarks for computer systems and make them available to the users of such systems as a common reference point in the evaluation of computer performance. It is participated by computer manufacturers, system integrators, consultants, publishers, universities and research organizations [50].

Since its foundation in 1988, the SPEC consortium has developed and distributed technically reliable benchmarks based on real applications. The selection of inputs and workload is performed by the consensus amongst consortium members willing for a transparent, comparable, reproducible and non-proprietary solution, as the organization aim is "*an ounce of honest data is worth a pound of marketing hype*".

SPEC currently offers testbench for the evaluation of different aspects of computation such as performance of CPU, graphics, distributed Java computing, web servers, and network file systems.

## 1.1. The SPEC CPU suite

The present technical report belongs to the field of performance quantification for intensive-computing. The testbench set from the SPEC organization that best fits this field is the SPEC CPU suite. The SPEC CPU suite aim is to be representative of programming style and application fields of real worldwide computer-intensive workload.

The first delivered set in 1989 had 10 programs and it was known as SPECmark. The most recent generation of the set is from 2006 (SPEC CPU2006) and it is made of 29 programs classified into two groups: 12 programs for integer computation (SPEC CPUint2006) and 17 programs for floating point computation (SPEC CPUfp2006) [20]. A more complete historical perspective of computer-intensive tests can be found in Henning's work [23].

SPEC CPU programs are well known real world applications written in high level, portable, language (C or C++) with slight code modifications in order to minimize input/output and thus let the processor, memory and compiler be the factors under evaluation. In fact, it is a requirement that input/output workload is less than 5% and the article from Ye, Ray and Kaeli show that the I/O activity of the SPEC CPU2006 is far less than this limit [58]. Another requirement is that memory consumption and execution time should be significant for each generation of computers with growing power and capacity. The organization keeps tight restrictions of evaluation rules affecting code, compilation flags and other aspect of execution environment of the test programs.

The workload sets the amount of processing performed by each benchmark run. The tools distributed by SPEC allow the specification of three different sizes of input data producing different workloads: test, train and reference ("ref" for short). The reference size stands for the reference workload, that is, the input data and command-line options when applicable, used for actual measurements. The test input sets are only used to check that programs compile and execute correctly before launching a real run or to tune optimization options. Similarly, the train inputs are used for profile-based compiler optimizations, so the reference set is the only reportable set.

The SPEC suite has a widespread usage by computer vendors, it is widely accepted by consumers and it is very commonly found too in the academic and research worlds although there is an outstanding debate about how to use it, its convenience and drawbacks, and whether is it or not necessary to design an alternative for research usage, etc.

The goal of this present work is reduced to the integer benchmarks (SPEC CPUint2006). A thoroughly characterization of SPEC CPUint2006 can be found in the technical report "*SPEC CPUint2006 characterization*" [13].

## 1.2. Controversia in the academic world

There is a live debate in the academic world lasting several years about whether the SPEC tests are adequate or not for research. As we said before, the workload has been design to contrast the hardware of the machines for the next following years. Consequently, the executed instructions count has considerably increased as well as the size of memory map used. Then, one of the problems stated is that they are far too large (in size and execution time) for experimentation and analysis. This has lead to some misuse by the research community in the attempt to cut down the size of experiments, according to some authors who

make an alert about the potential fragility of the results presented when conclusions are gathered in subsets selected without adequate assessment. In the ISCA 2003 (*International Symposium of Computer Architecture 2003*), Citron, Patterson and Sohi with Hennessy as moderator develop this subject commenting on a study made in the context of papers presented at a conference [5]. There, 90% of the papers providing measures used the SPEC, but a subset of the benchmarks was used instead of the full set in about 70% of the cases. But then only 30% of these explained why, and it was mentioned that, in some cases the subset consists simply of the benchmarks that did compile.

In summary, there is no agreement about the convenience of using the full set or just a subset of the SPEC CPU programs selected according to a researcher needs. Furthermore, some people think that binary code should be used instead of compiling source code or, like Sohi suggests, maybe there should be a test set designed for research purposes. Other alternative benchmarks programs have also been proposed recently [48].

### 1.2.1. Excessive workload

A major issue to take into account is the size of the testbench. As the suite SPEC CPU2006 was launched to keep up with the current technological and application changes from nowadays society, the inputs fed to the benchmarks had to grow to produce longer execution times and higher memory intensity. Consequently, instruction traces are much larger in the SPEC CPU2006 than they were in SPEC CPU2000.

The latest distribution of the *suite* executes more than 3 millions lines of source code which according to Phansalkar *et al*. [42] may correspond to a dynamic instruction count in the order of few trillions of instructions per program while in the SPEC CPU2000 they were in the order of few hundreds of billions.

This considerably large code size proposed by the SPEC organization as tests follows the principle "*more is better*" explained by the facts exposed in Henning work [23]:

- allow the testbench keep-up with the technological advances in computers[1] intensifying both execution time and memory usage,
- usage of real programs representing most commonly used application areas [20]; this is to avoid that a reduced selection may not be representative enough,
- try to include several programming languages and techniques following technology trends such as object orientation [56][2],
- pretend to force compilers developers to include optimizations for a wide range of programs,
- given that the SPEC CPU are designed to quantify CPU intensive behavior, input/output is reduced but most previous[3] and related code is preserved in order to keep code as much close to reality as possible,
- the variety of applications makes it possible that two duplicated programs (from the facts under test point of view) in today machines may be different in tomorrow's machines,
- a wide program base permits finding errors in *hardware-software* platforms or in development tools[4].

The considerably large workload inherently has some problems like heavy operability and a more complex maintenance. In addition, there are other disadvantages from the scientific point of view:

- evaluation becomes a tedious process,
- it is more probable to experiment difficulties at compilation time,
- simulation time grow far too much if simulators should be precise,
- evaluating embedded systems whose memory is necessarily limited in size becomes a difficult task because of the large memory map used by the tests,
- there might be redundancy or duplicity of workloads.

[1] "The quality and the runtime of benchmark data is one of the major problems with any benchmark. The runtime must be sufficiently large today to take account of future machine speed improvement. On the processors used during suite development, the runtime turns out to be about 20-30 minutes for reference data, 25% of the reference time for train data, and less than 2 minutes for test data" (taken from Wong's article [56]). "When processing is performed in an execution-driven simulator the time to complete the reference workload for the SPEC CPU 95 is several weeks and for the SPEC CPU2000 it is almost a year" (taken from Haskins and Skadron's [19] article).

[2] Using several programming languages and styles produce specific effects at code-level. So, for example, according to [21], floating point programs written in C++ produce a higher number of branches than programs written in C or FORTRAN.

[3] In the SPEC consortium there are reasonable doubts about whether input/output sequences should be reduced or not, that is, screen output may be avoided but, should the sequence to output an error message be totally excluded?

[4] Sometimes the SPEC programs don't work... because of an error never found before in the program.

In order to overcome these inconveniences, some techniques have been proposed to cut down work load size which will be analyzed thoroughly later on section 2.

## 1.2.2. Redundancy

Some authors state that the large size of the suite produces redundancy because input loads as well as applications may be quantifying similar facts. Redundancy increments evaluation costs, increases research and design times adding no extra information.

McGhan ensures in his paper published in *Microprocessor Report* [37] that the SPEC CPU2006 applications are redundant whereas, in contrast, some important application fields are not represented, like Electronic Design Automation (EDA). Other later publications applying statistical methods [42, 29] in fact conclude that there is redundancy and propose a truly representative subset [41]. These articles also state that the objection from McGhan about the lack of representation of the EDA sector in the suite is incorrect.

However, redundancy is not exclusive from the SPEC CPU latest distribution set. It was also present in previous suites, as stated by Saavedra and Smith [44], and by Giladi and Ahituv [14] where the SPEC89 release is analyzed. Later studies propose what are the fundamental characteristics a testbench must fulfill and emphasize the redundancy in the SPEC CPU95 *suite* [8, 18, 52]. Gustafson and Snell design a testbench (HINT) which can predict the behavior of the SPEC CPU95 [17] using only 300 lines of code. Vandierendonck and De Bosschere conclude that the SPEC CPU2000 programs are redundant [54]. Luo *et al.* analyze different techniques for quantifying similarity of work loads and measure redundancy in a selection of the SPEC CPU2000 [35].

Finally, the conclusions from some works [42] mainly state that application areas representativeness may not be an issue and that some programs from one area may behave more similar to another from a different area depending on the input data set used.

### 1.2.2.1. Quantification of similarity

Quantifying similarity basically consists first on a phase for selecting the characteristics describing the test program and then on a phase for grouping (clustering) using some method to estimate the difference (distance measure, grouping algorithms, etc.).

When selecting the set of characteristics to evaluate, these are normally classified as dependent or independent of the machine or microarchitecture. Independent characteristics are interesting to consider because they are characteristics inherent to a program and, therefore, they will permit to force always the same bottlenecks and hardware properties in different machines so providing for analyzing different behavior.

Among program inherent characteristics it is common to have:

- instruction mix
  - distribution of process, memory access and branch instructions;
- control flow behavior
  - basic block size
  - branch effective address
  - percentage of taken branches
  - percentage of forward taken branches
- instruction-level parallelism in source code
  - measured as the distance (in number of instructions) between data production (register write) and consumption (register read);
- data locality; and
- instructions locality.

Some common microarchitecture dependent characteristics are cache miss-rate, branch prediction accuracy, sequential flow breaks, instruction mix and instruction level parallelism actually achieves in execution.

Gathering information for these characteristics can be done in different ways: accessing hardware counters in real processors, instrumenting code or using simulators.

It is well known that many characteristics depend on each other. In order to simplify the problem and avoid correlations than can influence results some researchers use techniques to extract a representative

sample of characteristics, such as PCA (*Principal Component Analysis*). This technique is explained in [26, 9, 36] and in [60] there is a formalization of its application to the methodology of simulation.

Once the desired characteristics have been selected and reduced the next step is to measure the difference (or similarity) of each test program. This produces a grouping or *clustering* of similar test programs[5].

The measure of the difference can be done in different ways. Some works use distance measures (Euclidean, Manhattan, cosine, etc.), other use other specific definitions from a concrete problem (for example, in [16] they use coverage). In many work instances they use univariant statistical methods like chi-squared goodness-of-fit test [28] or multivariant statistical methods like Cluster Analysis [27, 45], among which we have the K-means clustering method, the hierarchical clustering method[6] or the K-medoids method. The chi-square is used, for instance by the authors of *MinneSPEC*, KleinOsowski y Lilja [31]; the K-means clustering method is applyed by Todi (for SPEClite) [53]; K-means and hierarchical clustering is used by Phansalkar *et al*. (for SPEC CPU2006 subsetting) [41, 42, 29], and K-medoids clustering by Luo *et al*. [35].

There is no actual agreement upon the best method to measure distance or clustering. Luo *et al.* [35] justifies why not to use K-means. Todi argues that resulting clusters are strongly dependent on the method of CA as well as on the selected characteristics [53]. And, it is rather meaningful that the similarity quantification used in MinneSPEC [31] (based on the chi-square distance) generally agrees with Eeckhout *et al.* (based on multivariant PCA and CA[7]) [12] when selecting the function-level execution profiles characteristic but, in contrast it differs when using another set of characteristics.

Then, we can conclude that the concept of similarity is not an universal one and that the criteria to be used for selecting inputs that may produce similar results to the reference set depends on the actual goal of the experiment.

# 2. Workload reduction

While large execution times are desirable for better results of performance measurements, it can be unaffordable for computer architects to perform precise evaluations. It is necessary to find lighter workloads that permits the experiments to be perform in reasonable times while still keeping the representativity of the benchmark.

Execution-driven or trace-driven simulators are examples of experimental systems for research that require benchmarks that permit obtaining results in a contained time without compromising the precision of research. On the other hand embedded systems in which there are limitations of memory size, can't execute programs with high memory consumption as the programs from the suite SPEC do under typical workload conditions, therefore these systems also require a reduction of memory requirements of test workloads.

In the relevant literature there is a wide range of alternatives oriented towards reduction of workloads. In the following sections we address the ones that appear in two works: the review from Haskins *et al.* [19] and Ringenberg's work [43].

The two most commonly used methods to reduce workload are the reduction of the suite and the sampling. Reducing the suite can be done either by reducing the input or by selecting a subset of programs. Sampling is performed in many different ways.

In Haskins *et al.* [19] we find a comparison of both techniques, concluding that both can produce significant errors but they also decrement considerably the evaluation time. Input reduction is a very valid technique although it requires becoming very knowledgeable of the code of the programs to be precise. Sampling has the problem of having to start from an unrealistic situation form the hardware standpoint and that needs to be taken onto account.

In addition to these two techniques other less extensively used are analytical modeling and statistical simulation.

[5] In the work of Joshi *et al*. [29], in section "*Related Work*", we dispose of a review about different ways of quantifying the difference that can be found in literature.

[6] "The K-means clustering algorithm divides a set of N programs into K groups, where K is a value specified by the user. Therefore, in order to evaluate different grouping possibilities one needs to cluster programs for different values of K and then select the best fit. Hierarchical clustering is useful in simultaneously looking at multiple clustering possibilities and the user can select the desired number of clusters using a dendrogram. Hierarchical clustering is a bottom up approach and starts with a matrix of distance between N cases or benchmarks. The distance is the Euclidean distance between the program characteristics" (taken from [42]).

[7] Some authors work with the so called phase techniques and they define phase as "a portion of dynamic execution of a program for which most of the performance metrics show very little variance". When that technique is applied to sampling they call it "phase based representative sampling" [35].

## *2.1.* Reducing the *suite*

The suite may be reduced by modification of the input data used or by selecting a subset of programs from the suite. The former technique allows reducing the number of processed instructions and consequently, execution time while it is also possible to reduce the memory map size used. The later technique reduces the executed code too discarding some of programs of the set. Both techniques can also be combined.

A paradigmatic example of the first technique is the so called *suite MinneSPEC* which is a reduction of input data for the SPEC CPU2000 testbech set [31]. As an example of the second technique, Phansalkar *et al*. [41] propose subsetting the programs of the SPEC CPU2006 suite.

It is a common practice to submit the reduced set to some kind of comparison with the original one in order to quantify similarities and discrepancies caused by the modified input and/or the programs used.

### 2.1.1. Reducing input data sets

Reducing input data is a traditional procedure to decrease the problem size to be processed by the test program and so reduce the evaluation time. Additionally it also allows reducing the memory map size used by the program.

The idea is to use the same programs of the original suite and decrease the workload submitted by acting upon the arguments of the call or input files used.

The great advantage of reducing the input is that programs execute completely including its initialization phase, computing phase and finish and cleaning phase [53, 19][8]. That is, the program executing with a reduced input is valid by itself and there is a high probability that the program will show the same execution profile as it does with the original workload, investing similar proportional times in each phase.

However, a "blind" input reduction is not adequate as it may change the behavior of the program if it happens to exercise a different hardware than the original case.

Deviations in computational behavior upon input changes have been studied using the different input data sets provided by SPEC for their benchmarks. The goal of the train workload is to be representative of the reference workload requiring less execution time in order to obtain profiling information and thus tune compiler options accordingly. However, analysis performed by Gove and Spracklen show that there are non-neglectable differences between the train and reference workloads for concrete programs of the suite although the program flow is similar for both workloads [16].

Additionally, analysis made by Phansalkar *et al*. show that, occasionally, changing the arguments of a program produces changes in its computational profile rather resembling another program of the suite [42][9].

In summary and in agreement with Haskins *et al.* [19], we must state that generating a reduced input that can reproduce the behavior of the original workload is not easy and requires a confident knowledge of the program code. This requirement produces open discrepancies in the scientific community. So, for instance, while it is absolutely essential for Sohi in order to perform a correct evaluation [5], for Todi it is an evident disadvantage [53]. On the other hand, the practice of using reduced input is inherently trustable in contrast to sampling and its potential error of initialization.

#### *2.1.1.1. MinneSPEC*

In 2002 a reduced workload was developed for the SPEC CPU2000 benchmarks: the *MinneSPEC* [31]. This workload design seeks equivalence of results compared to the reference workload still executing the same benchmark programs from start to finish, which is a simple and practical approach that can be applied.

The authors proposed a workload set reduced by the application of the following methods:

- command-line modification (avoiding or changing arguments)
- acting upon the input file

[8] The work of Haskins and Skadron states that it is a good practice to preserve all execution phases of a program [19].

[9] "It is not surprising (and even, to some degree, intended) that different input files often cause the program to exercise different paths, and that the subroutine distribution may vary considerably between invocations." This is stated by Weicker and Henning in an article where they analyze subroutine call profiles under the reference workload of the SPEC CPU2006 [55].

    - truncating the input file
    - sampling the input file
    - changing the contents of the input file
- using the train or test workload instead of the reference workload
- a combination of the above

*MinneSPEC* provides 3 workload levels: large, medium or small and they are called *lgred*, *mgred* and *sgred* respectively. These reduced datasets are meant to be options to decide upon the length of the tests vs. the fidelity with the original sets for a given experiment a user may want to carry out. The goal was to obtain a range of dynamic instructions in the order of 100 million, 500 million and 1,000 million.

The authors firmly state that it is very difficult to obtain proportional profiles of the reference load for some programs and even impossible for others. Nevertheless, they ensure that workload they propose is adequate for computer architecture analysis based upon the fact that the programs of the testbench remain the same, which is not a very strong argumentation.

#### *2.1.1.2.* Validation of the *MinneSPEC*

The first validation we deal with is from the authors of the *suite MinneSPEC* who claim that their workloads are representative of reference workloads. The authors quantified the differences perceived in terms of:

- function-level execution profiles,
- instruction mix,
- memory behavior.

The results obtained quantifying the similarity with the chi-square goodness-of-fit test, as exposed in [31], show that function level execution matches the behavior of the reference set reasonably well but not so for the instruction mix and memory footprint. Nevertheless, these published results had to be amended after a software bug was discovered in the *SimpleScalar* simulator and new, corrected results are shown in [1].

Eeckhout, Vandierendonck y De Bosschere in [12] show an alternative validation for the *MinneSPEC* proposal applying phase methods. This comparison use additional workload characteristics to the ones mentioned before (function execution, instruction count and memory footprint) such as instruction cache miss rate and branch prediction. The conclusion is that the large workload (*lgred*) behaves similar to the reference workload but not so the medium (*mgred*) and small (*sgred*) workloads.

The validation of *MinneSPEC* [31] based on characteristics from *function-level execution profiles* using *chi-square* as distance quantification generally fits with the evaluation from Eeckhout *et al.* [12] based on phase techniques. However, they show there can be differences when accounting for the other factors and find differences in program behavior even among pairs of inputs used for the same program.

### 2.1.2. Subsetting

Citron in [4] shows that the SPEC suite is used only partially in the work presented on the most relevant Computer Architecture events. So, the immediate question to pose is: is this partial usage a representative one? Phansalkar *et al*. [42] say that it depends on the actual subset being used.

The SPEC suite programs are redundant because of similarity in programs (see Section 1.2.2 Redundancy). An adequate selection can be convenient in order to decrease evaluation time[10].

Citron in [3] made an alert on the potential fragility of the results when conclusions are gathered in sub-sets selected without adequate assessment. In [4] Citron and Sohi together with Hennessy and Patterson develop this subject commenting on a study made in the context of papers presented at a conference. There, 90% of the papers providing measures used the SPEC, but a subset of the benchmarks was used instead of the full set in about 70% of the cases. But then only 30% of these explained why, and it was mentioned that, in some cases the subset consists simply of the benchmarks that did compile.

Phansalkar, Joshi and John propose measuring workload similarity using phase techniques in order to avoid redundancy and reduce the number of programs in the tets set thus saving experimentation (simulation) time. Their proposal based on the SPEC CPU2006 is in [41].

[10] The problem is that the subsets analysed by Citron are due to compilation difficulties, problems with system calls, problems with libraries, and so on, instead of a careful selection. Consequently, results can be considered incomplete [4].

## 2.2. Sampling

Sampling reduces workload to evaluate (by simulation or analysis) using only segments of execution. The simplest way to select those segments is by random sampling or uniform sampling. Using a more sophisticated method, we may impose some restrictions such as forcing the sample to be from the application essential part or, in general that it should be a representative sample. A popular technique to measure the representativity of the sample is to use quantification of similarity methods (see Section 1.2.2.1 Quantification of Similarity).

Sampling may also be used together with reduced input data sets.

There are some tools developed by the scientific community to select and analyze samples automatically. As examples we have *SMARTS* [57] and *SimPoint* [47]. Yi and Lilja make some recommendations about how to use the method of sampling [61].

According to Luo *et al*. [35], it is extremely important to quantify similarity in order to be confident on the quality of the samples. Changing the characteristics set to evaluate or using different grouping algorithms takes to different results. This is one of the main problems with sampling.

Another problem with sampling techniques is that the machine state when the simulation (evaluation) process is started is not the same than it was at the point of execution of the program where the sample was actually taken. Therefore, this can produce unrealistic results. Designing a solution to this problem is not easy: we shortly comment on it in the next section.

### 2.2.1. *Checkpointing* and *warmup* techniques

The big problem with using sampling to reduce workload is the impact produced by the so called initial state. When a sample is being processed the state of the machine must be as close as possible to the real case in architectural terms –registers and memory– and at the micro-architectural level –branch prediction, cache state, etc. –. This is very important for results to be realistic.

The way to solve this problem, in general terms, is to save the initial state during a real execution right before sampling and then apply it later upon evaluation of the sample. The name *checkpointing* is used for the process of state saving and *warmup* is the technique name used for restoring the state when the sample is performed later [19]. Ringenberg's thesis proposes and explains several methods and tools to do this [43].

### *2.2.2. SPEClite*

An example of sampling using phase techniques (see Section 1.2.2.1 Quantification of similarity) is *SPEClite* [53]. The method used to find the right sample is to monitor program execution (using the hardware counters in the Itanium in this case) and measure a set of variables. These variables are reduced using PCA (*Principal Component Analysis*) and then the distance among them is evaluated clustering the samples in order to select the most representative ones.

The author of this work specifies the following advantages of his proposal:

- it is a black box approach that only requires the binary code thus needing no precise knowledge at all of the source code[11],
- the use of statistical analysis permits identification of representative samples,
- the technique can be applied to any hardware and the sampling can be adapted to that hardware,
- the process can be automated,
- the methods permits extrapolation and result prediction based on sample representativity.

He also states some disadvantages:

- a sample in a given hardware today does not have to be good for another hardware in the future,
- the selection technique is strongly dependent on the clustering algorithm used.

*SPEClite* was expected to deliver a complete subset representing the whole suite SPEC CPU2000 but the project was completed.

[11] This is not so good according to Sohi [5] as we mentioned before.

## 2.3. Analytical modeling

Analytical modeling is not a very much extended technique although it has been used sometimes [3, 38, 49]. It consists on the definition of mathematical models which tend to work with statistical data which allows reducing experimentation time considerably.

## 2.4. Statistical simulation

Statistical simulation consists on the generation of synthesized workloads by gathering statistical information. The idea is to record information out of a set of variables chosen as the characteristics describing the behavior of a testbench in a real machine (execution profile) and then obtain a workload that fulfill those requirements by the process of synthesis. The synthesized workload may a binary trace (trace synthesis) or a source code program that should be compiled later (program synthesis).

Statistical simulation is used in this works [11, 25, 39, 40].

### 2.4.1. Trace synthesis

In the case of trace synthesis a synthetic trace is generating out of an execution profile which is then introduced in a trace-driven simulator in order to carry out the experimentation.

Oskin *et al*. [40] proposes HLS, a trace synthesis tool. This tool executes a program into a simulator and then obtains execution statistics. With these statistic figures it builds an instrumentalized trace which contains statistical information. Finally, a statistical simulator interprets this trace. Results obtained with these methods differ from the SPECint95 on 5 – 7%.

Nussbaum and Smith do something similar obtaining an execution profile which is introduced in the simulator [39].

Other authors propose obtaining a synthetic trace out of real trace [10].

### 2.4.2. Program synthesis

In the case of program synthesis, a C language program is build out of an execution profile. Once this source code is compiled, the binary obtained is in conformant to the required profile. Finally, this executable is evaluated into an code execution-driven simulator.

Bell and John in [2] are an example of this type of technique.

# 3. Workload reduction approach

Among all the methods addressed by the relevant literature we decided to choose input data sets reduction (see section 2.1.1, Reducing input data sets).

The main goal is to reduce the number of executed instructions down to a number of, approximately 100 millions ($10^8$) executed instructions. It would also be a secondary benefit to reduce the amount of memory consumption.

The idea is to avoid program code changes and work with input data only. In summary we must analyze the instruction counts (and the memory usage profile) upon program arguments modifications. The goal is to discover the possible relations between these arguments and the dynamic instruction count and, consequently, the processing time[12].

Once the new input data sets are defined, we analyze the distribution of instruction types, CPI, procedure call and system calls as well as the memory profile produced and compare it with the corresponding figures for the reference workload.

In the following sections we perform an in-depth analysis of each benchmark of the SPEC CPU2006 integer suite (SPEC CPUint2006) from different perspectives, including the influence of each of the input data set used into the actual workload with the purpose of helping researchers finding a representative set

[12] Processing time is a function of the instruction count and the CPI, if the CPI profile is not changed, the processing time would be uniformly proportional to the instruction count. But, usually, the CPI is not maintained easily.

of workloads for the SPEC CPUint2006 programs to use in their experiments whenever they have to discard using the reference workload due to time or resource constraints.

## 3.1. 400.perlbench

### 3.1.1. Description

The program *400.perlbench* is a cut-down version of Perl v5.8.7, the popular scripting language. SPEC's version of Perl has had most of OS-specific features removed. In addition to the core Perl interpreter, several third-party modules are used [20].

### 3.1.2. Input data driven behavior

The input data sets for *400.perlbench* consist of a variety of scripts. As can be seen in a previous technical report, each invocation produces quite different workloads when they are described by microarchitecture-independent characteristics [13]. Program *400.perlbench* exhibits quite different subroutine call distributions for each invocation, which means that the program follows different execution flow paths depending on the inputs.

Program *400.perlbench* under the test workload is called several times with different inputs and each invocation produces a very low dynamic count each. Thus, any one input data set from the test workload can be chosen to decrease the executed instruction count since the subroutine call distribution suggests that every invocation exercises different execution flow paths and then it is reasonable to say that selecting one invocation or another would be as much adequate as incomplete at the same time.

## 3.2. 401.bzip2

### 3.2.1. Description

The *401.bzip2* benchmark is a modified version of the popular *bzip* compressor accessing disk only to read the input; the only output of the program is to the standard output as short messages to indicate execution progress. Compression and de-compression occurs entirely in memory [20].

Two arguments are passed to the program: the file name to read containing the basic data to compress and de-compress, and the buffer size. The program starts reading the input file into memory, then this data is duplicated several times up to buffer size in MB thus in creasing the size of the sample. Compression and decompression is then performed for three compression factors: 5, 7 and 9 (also called blocking factors). Decompressed data is compared to the original image and a short message states that it compared successfully.

### 3.2.2. Input data driven behavior

Table 1 shows the invocation parameters making up the reference workload used for each input set, and some dimension related values taken for each invocation.

The first column shows the elapsed times for each invocation. These times are just a reference for comparative purposes and they were taken for *Pentium* processor. Next column data is a percentage of the time spent in that invocation with respect to the total time of the workload set. The following two columns show the input buffer size and its contribution to the actual total amount of input data processed by the program in the reference set. The last three columns show the percentage of data compression obtained when the *401.bzip2* benchmark is used to compress the input buffer data with compression factors 5, 7 and 9 respectively.

Table 1. Input data for *401.bzip2* using the reference workload and some figures used for comparison.

| | **elapsed time** | | **input buffer** | | **compression factor** | | |
|---|---|---|---|---|---|---|---|
| | **secs** | **% total** | **bytes** | **% total** | **5** | **7** | **9** |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| `chicken.jpg 30` | 131.40 | 7.81% | 31,457,280 | 2.73% | 0.5 | 3.3 | 17.5 |
| `liberty.jpg 30` | 259.90 | 15.45% | 31,457,280 | 2.73% | 30.3 | 43.7 | 56.3 |
| `input.program 280` | 251.50 | 14.95% | 293,601,280 | 25.45% | 65.8 | 65.3 | 64.9 |
| `input.combined 200` | 412.84 | 24.54% | 209,715,200 | 18.18% | 78.6 | 79.1 | 79.4 |
| `input.source 280` | 309.47 | 18.40% | 293,601,280 | 25.45% | 81.5 | 81.7 | 82.0 |
| `text.html 280` | 317.03 | 18.85% | 293,601,280 | 25.45% | 92.4 | 93.9 | 95.1 |

There are three types of inputs: images, text and binary code. Images represent almost 23.3% of the total reference workload in terms of time and almost 5.5% in terms of size[13]. The images used as inputs show the lowest compression obtained amongst the input. The image `chicken.jpg` shows a compression percentage extremely low, in particular for a compression factor of 5. It substantially improves the amount of compression reached for a 9 factor compared to 7 and of course 5. In the other image (`liberty.jpg`) this is not so dramatic at all. It tends to behave more like the rest of input types where the compression reached for 5 and 7 is closer to the one obtained with the 9 compression factor.

Text is about 62% of the total reference workload in terms of time and slightly over 69% in terms of size[14]. Text type inputs show very high compression levels (almost 80% to 95%) and they don't show excessive improvement in the amount of compression reached at factors higher than 5.

Binary code represents in practice 15% of the total reference workload in terms of time and slightly over 25% in terms of size[15]. The executable file too exhibits practically no impact of the compression factors into the compression achieved which is, as can be expected, more conservative than that of text files, achieving about 65% compression.

We can reduce the workload making the program to use only one compression factor. Compressing at factor 7 looks a good choice. Figures in table 1 show that it is a good representative of the compression quantity that is achievable in each case. It would generate a third of the instructions compared to 5+7+9 factors. The problem is that we need to change the program (alter the benchmark) and this can disappoint people.

Without changing the code of the benchmark, we can generate a single input file showing a mix of the different input types we have classified keeping its relative weights. In case of text type of input, the compression achieved is about the same for the 3 compression factors. It means its behavior is very regular; and its total weight is about 70% in terms of size. The binary file is also regular and must be present in about 25%. The 5% remaining should be a picture. This mix should be scaled in such a way that produces the desired instruction dynamic count.

## 3.3. 403.gcc

### 3.3.1. Description

The program *403.gcc* is based on *gcc* Version 3.2. It generates code for an AMD Opteron processor. The benchmark runs as a compiler with many of its optimization flags enabled. It has had its inlining heuristics altered slightly. This was done so that *403.gcc* would spend more time analyzing its source code inputs, and use more memory. Without this effect, 403.gcc would have done less analysis, and needed more input workloads to achieve the run times required for CPU2006 [20].

### 3.3.2. Input data driven behavior

The input data sets of each invocation correspond to files that are preprocessed C code (.i files). As it can be observed in a previous technical report [13], the behaviour of the program is quite no-elastic, that is, every invocation always produces a large amount of dynamic executed instructions.

Taking into account that a “hello world” program already takes 20 million instructions, we deleted some C functions from the smallest program used in the invocations of the test workloads until a satisfactory dynamic count was obtained; in our case that was 257 millions for a 464 lines of code with 16 functions.

[13] For the test workload, the processing of images takes about 67.8% of total time and represents 28.6% of the total input size while for the train workload they take 11.4% of the time and 5.3% of the input size.

[14] For the test workload, no text files are used while for the train workload, text files represent 78% of the time and a bit more than 84% of size.

[15] Binary code takes 32.2% of total execution time and a 71.4% of the size for the test workload, and about 10% for both time and size in the case of the train workload.

As far as the representativeness is concerned, *403.gcc* exhibits quite different subroutine call profiles for each invocation (as it also happens with *400.perlbench*), which means that the programs follows different execution flow paths depending on the inputs, thus concluding that any one invocation can be considered adequate and incomplete at the same time.

## 3.4. 429.mcf

### 3.4.1. Description

The *429.mcf* benchmark is a C program for the scheduling of a single-depot vehicle fleet in public mass transportation. A timely plan expressed as time-table trips with fixed departure/arrival locations and times is used to assign a vehicle to each trip. Trips are linked by the so-called dead-head trips and to enter and leave the base there are pull-out and pull-in trips. These three trips types have associated cost coefficients which have to be minimized by the algorithm used as well as the number of necessary vehicles to schedule all time tabled trips. The algorithm used is a network simplex algorithm. This program is basically the same as in the CPUint2000 set, but the input files are designed to obtain longer execution times, thus increasing the heap data size, and with it the overall memory footprint [20].

### 3.4.2. Input data driven behavior

The program *429.mcf* takes time-table entries from a file which contains the number of Timetabled Trips (*TT*) and Dead-Head trips (*DHT*); for each *TT* trip it states the starting and ending time and then it lists all links for pairs of *TT* which are *DHT* and its cost.

The strategy to define a reduced input is to monitor the executed instruction count *vs.* a reduced timetable specification. For this, we build a new input file taking the original input file, say the one for the test workload, and change the number of trips (*TT* in first line) to *N* and select only the first *N* trips together with the *DHT* which link those *N* trips. After selecting the right *DHT* entries, we count them and complete the first line of the new file, assembling the new input file.

Table 2. Executed instructions in *429.mcf* according to the number of the *TT* and *DHT* arguments.

| ***TT*** | ***DHT*** | **millions of executed instructions** |
|---|---|---|
| 100 | 51 | 3.8 |
| 799 | 6,141 | 115 |
| 999 | 9,296 | 164 |
| 5,985 | 84,449 | (test workload) 4.8 |
| 13,225 | 164,741 | (train workload) 24,000 |
| 25,137 | 185,356 | (reference workload) 357,000 |

We obtained some figures increasing *TT* from 100 on to find the most convenient size of the input graph, as shown in Table 2. The last three rows correspond to the three SPEC workloads. A range of 799 to 999 nodes can produce an experimentally adequate number of executed instructions.

## 3.5. 445.gobmk

### 3.5.1. Description

The program plays Go and executes a set of commands to analyze Go positions. This benchmark is typical in SPEC CPU suites as it has been included in other releases [20].

### 3.5.2. Input data driven behavior

Most input is in "SmartGo Format" (.sgf), a widely used de facto standard representation of Go games. A typical test involves reading in a game to a certain point, then executing a command to analyze the position.

From the SPEC CPUint2006 characterization [13], we observe that this benchmark is a non-elastic binary, that is, it produces a regular dynamic executed instruction count for different input data sets. Moreover, we know that for each invocation of *445.gobmk* the program exercises similar execution flow paths. Then, researchers may select any invocation that best fits their needs from the test or train workload sets.

## 3.6. 456.hmmer

### 3.6.1. Description

DNA pattern sequence searching is based on statistical models such as Hidden Markov Models profiles (HMMs) for multiple sequence alignments. The benchmark includes a search function for sentitive searching in a statistical descriptions database and a calibration function to calibrate HMM search statistics. Train and test workloads as well as one of the two invocations of the reference workloads use the calibration function with different calibration files each. The second invocation of the reference workload uses the search function to find patterns in a database file (sprot41.dat) [20].

### 3.6.2. Input data driven behavior

The arguments used for the test invocation which uses the calibration function, its meaning and the impact of each in the dynamic instruction count are the following:

- *fixed*: random sequences length; if 0, a normalized gaussian distribution is used to control the length of the generated sequences
- *mean*: mean length of synthetic sequences (350 by default); this directly impacts the number of executed instructions
- *num*: number of synthetic sequences (should be greater than 1000 and 5000 by default); also affects dynamic counts decisively
- *sd*: standard deviation for the length of the synthetic sequences (350 by default); it doesn't affect much dynamic counts.
- *seed*: random seed used; by default time() function is used. It has no effect on the number of executed instructions

Therefore, we should use different values for arguments *mean* and *num*. In Fig. 1 we present the evolution of dynamic count *vs.* variations of *num* for different *mean* values. We can observe how scaling *num* from 45000, which the test workload value for this argument, down to 5000 (default value), the number of executed instructions decreases by an order of magnitude for different values of argument *mean*. If we keep decreasing *num* down to the limit 1000 beyond which it is not allowed to pass, we can still decrease dynamic counts by an additional order of magnitude compared to the test workload test.

Under the train and reference workloads, this benchmark executes a single subroutine for more than 95% of the execution time. For the test workload the percentage of time in that subroutine is slightly smaller than 70%. For a workload using *num* = 1.500 it is relaxed to 62%. This can be a convenient situation for testing purposes since it presents a less peaky profile.

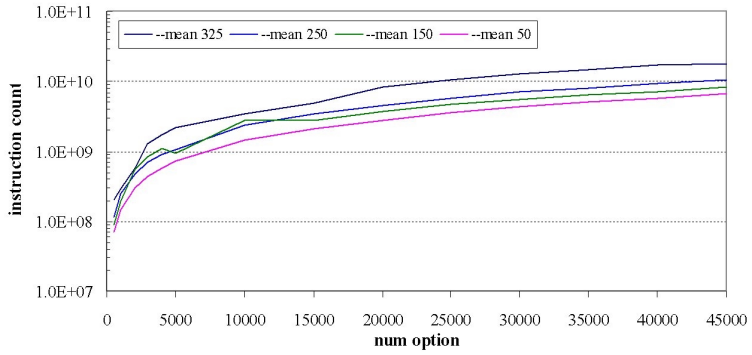


Fig. 1. Dynamic counts of *456.hmmer vs.* option *num* for different values of option *mean*. Y axis is in logarithmic scale.

## 3.7. 458.sjeng

### 3.7.1. Description

This an artificial intelligence program for chess. The input is a text file containing a variable number of lines containing two informations: the position in the board of the chess pieces using FEN (Forsyth-Edwards Notation) and the depth for the search of possible moves (position analysis depth). Each file line corresponds to a new program execution [20].

### 3.7.2. Input data driven behavior

Depth affects execution time and, consequently instruction count too. Memory usage seems independent of both parameters since it is maintained in about 180.000KB all the time. After a quick look at the program we observe that memory requests are independent of depth and position.

In order to better understand the relation between depth and instruction count, we performed some tests, selecting 4 scenarios (states, situations) on the board specifying the corresponding FEN coordinates and these were input to the program using position analysis depths 1 to 10. The following are the scenario used in each round:

| Table 3. FEN coordinates used for testing *458.sjeng*. | |
|---|---|
| testPOS1 | `r3kb1r/3n1pp1/p6p/2pPp2q/Pp2N3/3B2PP/1PQ2P2/R3K2R w KQkq - bm d6; id "LCT2-POS-01"` |
| testPOS2 | `1k1r3r/pp2qpp1/3b1n1p/3pNQ2/2pP1P2/2N1P3/PP4PP/1K1RR3 b - - bm Bb4; id "LCT2-POS-02"` |
| refPOS6 | `8/3b4/5k2/2pPnp2/1pP4N/pP1B2P1/P3K3/8 b - - bm f4; id "LCT2-FIN-06"` |
| refPOS7 | `r4r1k/pbnq1ppp/np3b2/3p1N2/5B2/2N3PB/PP3P1P/R2QR1K1 w - - bm Ne4; id "KASP-1"` |

They were all taken from the SPEC workloads: the first 2 from the test workload and the other 2 from the reference workload. Tabla 4 shows the results obtained.

Table 4. Executed instruction count for the *458.sjeng* program for the different depth values used in 4 different scenarios.

| Depth | testPOS1 | testPOS2 | refPOS6 | refPOS7 |
|---|---|---|---|---|
| 1 | 11.649.617 | 11.623.104 | 11.485.428 | 12.495.003 |
| 2 | 13.539.288 | 14.832.823 | 11.927.435 | 16.196.008 |
| 3 | 18.223.516 | 17.058.114 | 12.484.079 | 27.419.780 |
| 4 | 26.415.024 | 26.069.145 | 16.595.774 | 63.482.813 |
| 5 | 45.415.616 | 60.022.667 | 22.310.129 | 135.031.060 |
| 6 | 92.666.606 | 116.056.546 | 33.570.361 | 755.331.078 |
| 7 | 229.183.640 | 291.685.649 | 45.561.373 | 1.790.128.125 |
| 8 | 1.921.354.106 | 550.733.349 | 92.143.753 | 16.155.050.225 |
| 9 | 4.377.332.556 | 1.277.611.078 | 164.398.472 | 30.254.455.017 |
| 10 | 8.696.563.868 | 2.600.004.037 | 592.725.929 | 380.946.797.626 |

Graph in Fig. 2 shows program behavior. Executed instruction counts are in logarithmic scale for each series.

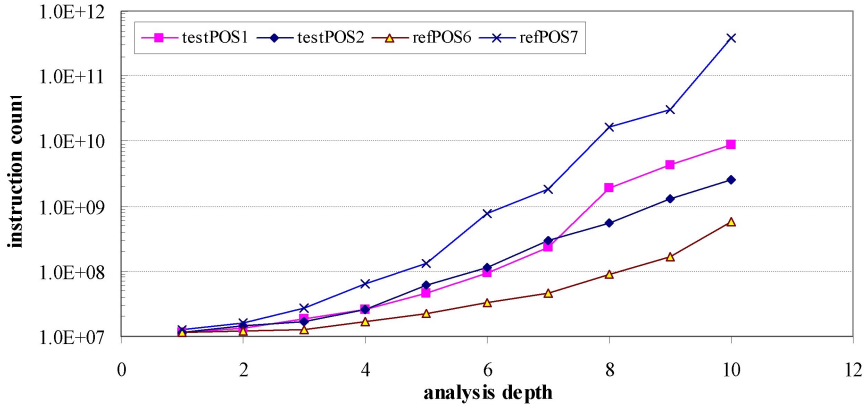

Fig. 2. Executed instruction count for program ***458.sjeng*** for the different depth values used in 4 different scenarios. Y axis is in logarithmic scale.

The main in influence on instruction count comes from the analysis depth and the complexity of the board state is next. We register executed instruction counts ranging from $10^7$ to the $5 \cdot 10^{11}$.

## 3.8. 462.libquantum

### 3.8.1. Description

The program is based on a library for the simulation of quantum computers. Specifically, it includes an implementation of the Shor's factorization algorithm. The input is an argument representing the number to factorize. A second optional argument called *base* may be provided specifying a ramdom seed for the modular exponentiation of the Shor's algorithm [20].

### 3.8.2. Input data driven behavior

The magnitude of the factorizing number affects execution time as well as used memory size. The second optional parameter, *base*, has no relevant impact on the execution time.

We defined a test set using different combinations of factorization number and base. Fig. 3 represents the mean instruction count for the different combinations. Standard deviations are neglectable. Instruction count increases with the factorization number (although there can be some local factorization consuming less time). The base optional parameter has not noticeable impact into the final count.

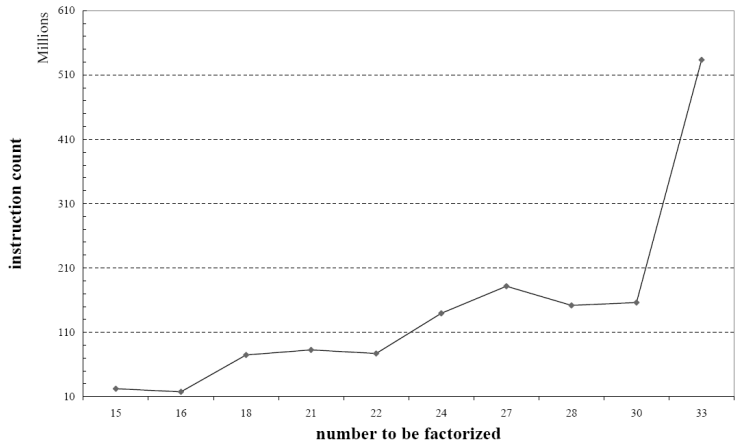


Fig. 3. Instruction count mean for each factorization number with different bases (program *462.libquantum*).

For the range of arguments used in the tests, we obtain from $2 \cdot 10^7$ to $5 \cdot 10^8$ executed instructions.

## 3.9. 464.h264ref

### 3.9.1. Description

This program implements the h.264/avc video compression latest generation algorithm. It may operate in baseline mode (good compression, fast coding) or in main mode (very good compression) [20].

The invocation argument is a configuration file which sets an uncompressed video sequence in YUV[16] format and a large number of operation parameters. When the program works under the baseline

[16] The SPEC workloads only work alternatively over two video sequences in YUV format without compression: `foreman_qcif.yuv` (120 *frames* of 176x144 pixels); and `sss.yuv` (171 *frames* of 512x320 pixels).

mode, the only parameter is being modified among the different configuration fields is the number of frames (`FramesToBeEncoded`) to be processed. Under the main mode, many more parameters are modified.

## 3.9.2. Input data driven behavior

Workload changes according to the values given to the parameters found in the configuration file. From all these available parameters, we try to focus on the ones showing evident meaning to avoid having to deal with a deeper knowledge of signal treatment techniques.

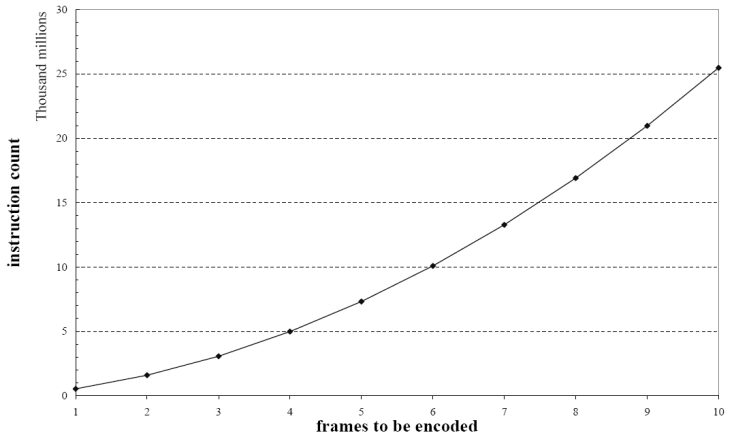


Fig. 4. Instructions executed for program *464.h264ref vs.* number of *frames* to encode.

The chart in Fig. 4 shows how executed instruction count grows with the number of frames to encode (parameter `FramesToBeEncoded`), thus concluding that this parameter has a decisive impact on it. However, this is not enough because even for the lowest computational load (550 millions instructions), our limit is exceeded (100 millions).

We also tried the parameter for number of reference frames (`NumberReferenceFrames`) but it was not relevant.

The size of the image or *frame* is the multiplication of width (`SourceWidth`) and height (`SourceHeight`). The following chart shows a direct (almost linear) dependency between instruction count and image size. Therefore, this parameter does allow a more efficient way to decrease the number of executed instructions.

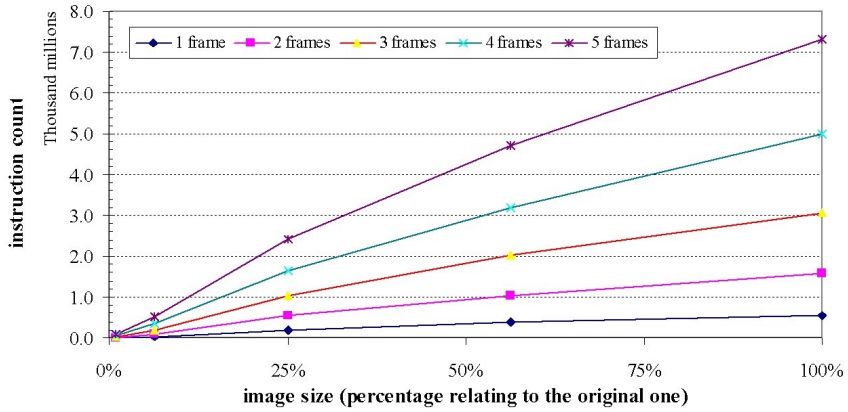


Fig. 5. Dynamic instruction counts of program *464.h264ref vs.* image size (percentage of original area) for different values of the parameter frames to encode.

For the arguments rage used in the tests we can obtain instruction counts ranging from $8 \cdot 10^6$ to $7 \cdot 10^9$ executed instructions.

This program is quite susceptible to changes in instruction set architecture between x86-32 bits and x86-64 bits because function calls are much faster in 64-bit mode since the calling convention allows up

to 8 arguments to be passed through registers (because of the availability of additional registers) in contrast with 32-bit mode where arguments are passed through the stack. As a result, there are more instructions and more memory accesses in the 32-bit version of *464.h264ref*, causing about a 2x slowdown performance as reported in [59].

## 3.10. 471.omnetpp

### 3.10.1. Description

The benchmark performs discrete event simulation of a large Ethernet network. The simulation is based on the OMNeT++ discrete event simulation system, a generic and open simulation framework.

This program simulates an Ethernet network described in a file (input file *omnetpp.ini*) written in the NED language [20]. This file contains several parameters affecting basically, the following:

- output stream control: to screen and files
- topology of the network to simulate
- network traffic model and total time to simulate.

Concerning I/O, screen output is controlled by parameters `express-mode` and `status-frequency`. If the first one is set to `true` it forces an operation mode with few messages whereas the second one allows setting the number of events after which a status message should be written to the screen. Parameters `*.enabled` and `*.writeScalars` control whether or not statistics files are to be generated.

The topology of the network is described through many parameters. Among them all it can be worth mentioning the length of requests and answers (`*.cli.reqLength` and `*.cli.respLength`) as well as the time between request requests (`*.cli.waitTime`). For all the workloads the value of `*.cli.reqLength` is a distribution along the range 50-1400. Parameter `*.cli.respLength` is a normalized distribution truncated to non-negative numbers with average 5000 and standard deviation 5000 for the train and reference workloads. For tests it is 3000 for both values. The parameter `*.cli.waitTime` is an exponential distribution whose average is the reciprocal of requests per second. That is, the inverse of `*.cli.waitTime` represents the average of requests per second. These parameters are modelled through random distribution functions defined in the NED language.

Finally, simulation time is given by `sim-time-limit`. This parameter can be specified in different time units: us (microseconds), ms (milliseconds), s (seconds), etc.

It is also relevant to mention that the program goes through an initial phase where it reads the configuration file setting up the network according to the read-in parameters and then enters a second, final phase where simulation takes place.

### 3.10.2. Input data driven behavior

The fundamental differences between the files for the test, train and reference workloads are:

- the topological complexity of the network described in each case;
- simulation time;
- average requests per second; and
- request and response lengths

The main factor affecting instruction count is the complexity of the network topology. There is a single network in the case of test and reference while there are two of similar complexities in the case of the train workload. Out of the four networks proposed by the SPEC consortium, the one for reference is the most complex while **test** is the one with the lowest complexity (produces lower execution counts).

Simulation time (`sim-time-limit`) influences dramatically the amount of executed instructions too.

For testing purposes, we took separately each of the networks proposed (one for test, 2 for train and one for reference) and processed them with different simulation times.

Results are shown in the following chart. First we observe that both topologies proposed for train are practically the same.

Executed instruction count grows with workload (test -> train -> reference). We see that only a few simulation times for the test network topology allow remaining fewer than 100 million instructions (our objective limit).

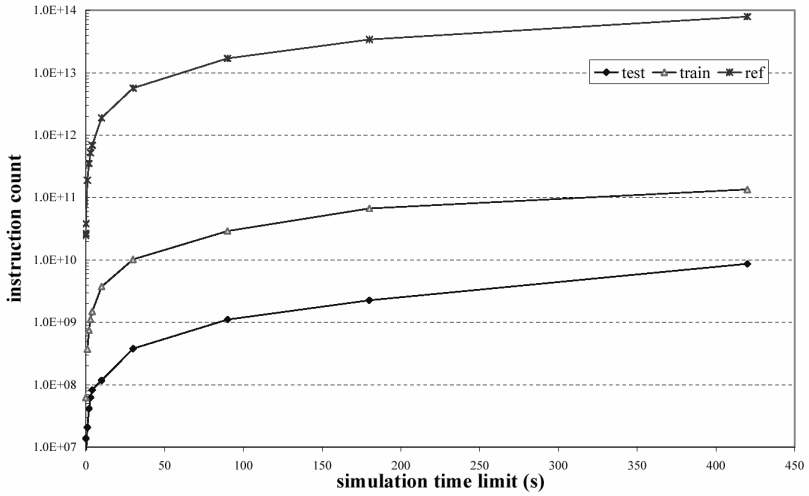


Fig. 6. Instruction count *vs.* limit of simulation time for program *471.omnetpp*. There are a series for each network topology of the 3 workloads. Train workload has 2 networks which produce extremely similar results, so these cannot be distinguished in the graph and then we plot only one of them. Y axis has logarithmic scale.

Request length (`*.cli.reqLength`) slightly conditions instruction count. Response length (`*.cli.respLength`) influences the count in a way inversely proportional to the average. The time between requests (`*.cli.waitTime`) influences is inversely proportional to the count.

However, all the modifications of parameters `*.cli.reqLength`, `*.cli.respLength` and `*.cli.waitTime` which allow to modify the operation mode in the network without altering its topology does not substantially reduce the executed instruction count.

Consequently, the only option is to use the test network (the simplest topology) modifying simulation time limit. This way, we may obtain counts ranging from $6 \cdot 10^6$ to $8 \cdot 10^9$ executed instructions.

## 3.11. 473.astar

### 3.11.1. Description

This program implements path-finding algorithms using Artificial Intelligent techniques that are commonly applied in games. An input file defines the binary map to use and other parameters used in these algorithms such as the number of paths to simulate, region size, density and others [20].

### 3.11.2. Input data driven behavior

The input data set is a text file specifying values for several parameters and the name of a file containing a binary map. This binary map file is really a file of bytes which can take the values 0, 1 or 2. Additionally, it includes a heading section made of 8 bytes which specify the map dimensions according to axis X and Y (four byes per dimension). The program reads in these dimensions and builds a binary map repeating the data found in the file as much as needed (a fixed value written in code). The test workload binary map file size is 64KB plus the 8 byes for the heading setting the map size. The bytes used for dimensions are organized orderly for growing offset. So, the lowest weight corresponds to the first byte, the following weight to the second, and so on. Original map dimensions for binary map *lake.bin* from the test workload are 00 01 00 00 and 00 01 00 00 which correspond to 00000100h x 00000100h, that is, 256 x 256. Smaller dimensions means to build a binary map usually more uniform and then easier to explore.

Parameters present in the input text file refer to two types of configurations: profiling of the area and amount of work to perform (first with the input binary map file and then with a map created randomly).

About profiling of the area, we have a parameter that measures the degree of obstacles in the map (`random map density`). The workload is inversely proportional to it since the resolution is worst then there are fewer obstacles and fewer regions.

For reducing the amount of work to perform, we should reduce the size of the random map changing the argument `random map size x` and `y`.

## 3.12. 483.xalancbmk

### 3.12.1. Description

This program is a modified version of Xalan-C++, an XSLT processor written in a portable subset of C++. This program transforms XML documents into HTML documents [20]. The input is an XML file and an XSL style sheet.

The test and reference workloads style sheets are identical. The train workload style sheet is simpler. XML files are larger for a larger workload. So, the test workload is different from the reference workload only in the size of the XML document used.

### 3.12.2. Input data driven behavior

The only option to reduce workload in a substantial way is to modify the XML document file making it simpler.

The following chart shows the number of executed instructions for different XML documents which were obtained by modifying the original document used for the test workload. They are simpler in the right side of X axis in the graph where we can observe how the instruction count tends to decrease.

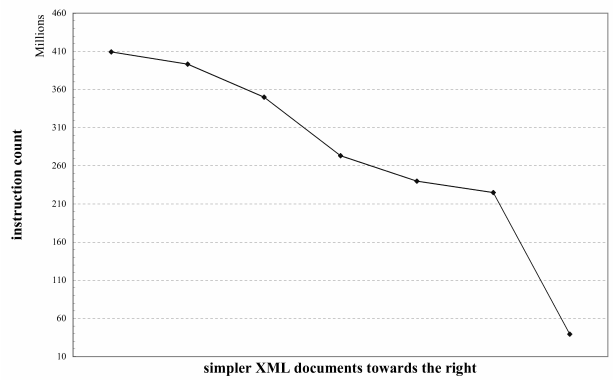


Fig. 7. Instruction count for the program *483.xalancbmk* for different input XML documents. They are simpler for growing values of the X axis.

Additionally, we can also omit argument `-v` which forces a validation producing a little more code execution.

# References


[1] Arctic Labs Web Page, University of Minnesota. http://www.arctic.umn.edu/minnespec/index.html/

[2] R. H. Bell and L. K. John. Improved automatic testcase synthesis for performance model validation, in Proceedings of the 19th Annual International Conference on Supercomputing, pp. 111-120, June 2005.

[3] R. Carl and J. E. Smith. Modeling superscalar processors via statistical simulation, in *Proceedings of the Workshop on Performance Analysis and its Impact on Design*, June 1998.

[4] D. Citron. MisSPECulation: partial and misleading use of spec CPU2000 in computer architecture conferences, in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 52-59, June 9-11, 2003.

[5] D. Citron, J. Hennessy, D. Patterson, and G. Sohi. The use and abuse of SPEC: An ISCA panel, *IEEE Micro*, Vol. 23, No. 4, pp. 73-77, July/August 2003.

[6] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors, in *Proceedings of the 1996 International Conference on Computer Design (ICCD-96)*, pp. 468-477, October 1996.

[7] P. K. Dubey and R. Nair. Profile-driven sampled trace generation, *Technical Report RC 20041*, IBM Research Division T. J. Watson Research Center, April 1995.

[8] J. Dujmovic. Universal Benchmark Suites – A Quantitative Approach to Benchmark Design, in *Performance Evaluation and Benchmarking with Realistic Applications*, MIT Press, Editor: R. Eigenmann, p. 304. January 2001.

[9] G. Dunteman. *Principal Component Analysis*, Sage Publications, 1989.

[10] L. Eeckhout, K. De Bosschere, and H. Neefs. Performance analysis through synthetic trace generation, in *Proceedings of the International Symposium on Performance Analysis of systems and Software*, pp. 1-6, April 2000.

[11] L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces, in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pp. 25-34, September 2001.

[12] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Designing computer architecture research workloads, *IEEE Computer*, Vol. 36, No. 2, pp. 65-71, February 2003.

[13] V. Escuder and R. Rico. 2009. SPEC CPUint2006 characterization. *Technical Report TR-HPC-01-2009*. Universidad de Alcalá. URL= http://atc2.aut.uah.es/~hpc.

[14] R. Giladi and N. Ahituv. SPEC as a Performance Evaluation Measure, *IEEE Computer*, Vol. 28, No. 8, pp. 33-42, August 1995.

[15] D. Gove. CPU2006 Working Set Size. *Computer Architecture News*, Vol. 35, No. 1, pp. 90-96, March 2007.

[16] D. Gove and L. Spracklen. Evaluating the correspondence between training and reference workloads in SPEC CPU2006, *Computer Architecture News*, Vol. 35, No. 1, pp. 122-129, March 2007.

[17] J. L. Gustafson and Q. O. Snell. HINT: A new way to measure computer performance, in *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, Vol. 2: Software Technology, IEEE Computer Society Press, Editors: H. El-Rewini and B. D. Shriver, pp. 392-401, January 1995.

[18] J. L. Gustafson and R. Todi. Conventional Benchmarks as a Sample of the Performance Spectrum, in *Performance Evaluation and Benchmarking with Realistic Applications*, MIT Press, Editor: R. Eigenmann, p. 304. January 2001.

[19] J. Haskins, K. Skadron, A. KleinOsowski, and D. J. Lilja. Techniques for Accurate, Accelerated Processor Simulation: Analysis of Reduced Inputs and Sampling, *Technical Report CS-2002-01*, University of Virginia, 2002.

[20] J. L. Henning. SPEC CPU2006 benchmark descriptions, *Computer Architecture News*, Vol. 34, No. 4, pp. 1-17, September 2006.

[21] J. L. Henning. Performance counters and development of SPEC CPU2006, *Computer Architecture News*, Vol. 35, No. 1, pp. 118-121, March 2007.

[22] J. L. Henning. SPEC CPU2006 Memory Footprint, *Computer Architecture News*, Vol. 35, No. 1, pp. 84-89, March 2007.

[23] J. L. Henning. SPEC CPU Suite Growth: An Historical Perspective, *Computer Architecture News*, Vol. 35, No. 1, pp. 65-68, March 2007.

[24] V. S. Iyengar, L. H. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache, in *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA-2)*, pp. 62-73, February 1996.

[25] V. S. Iyengar and L. H. Trevillyan. Evaluation and generation of reduced traces for benchmarks, *Research Report RC 20610*, IBM, October 1995.
[26] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley-Interscience, 1991.
[27] A. Jain and R. Dubes, *Algorithms for Clustering Data*, Prentice Hall, 1988.
[28] N.L. Jonhson, S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*, Second Ed., John Willey and Sons, 1994.
[29] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring Benchmark Similarity Using Inherent Program Characteristic". *IEEE Transactions on Computers*. Vol. 55, No. 6, June 2006.
[30] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches, *IEEE Transactions on Computers*, Vol. 43, pp. 664-675, June 1994.
[31] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research, *Computer Architecture Letters*, Vol. 1, No. 1, pp. 7-10, January 2002.
[32] W. Korn and M. S. Chang. SPEC CPU2006 sensitivity to memory page sizes, *Computer Architecture News*, Vol. 35, No. 1, pp. 97-101, March 2007.
[33] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream, in *Proceedings of the IEEE 3rd Annual Workshop on Workload Characterization (WWC-2000*, September 2000.
[34] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples, *Technical Report SMLI TR-93-22*, Sun Microsystems Laboratories Inc., December 1993.
[35] Y. Luo, A. Joshi, A. Phansalkar, L. John, and J. Ghosh. Analyzing and improving clustering based sampling for microprocessor simulation, in *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing*, pp. 193-200, October 2005.
[36] B. F. J. Manly. *Multivariate Statistical Methods: A Primer*, 2nd ed., Chapman & Hall, 1994.
[37] H. McGhan. SPEC CPU2006 Benchmark Suite, *Microprocessor Report*, October 2006.
[38] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance, in *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pp. 298-309, February 1997.
[39] 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*, pp. 15-24, September 2001.
[40] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design, in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 71-82, June 2000.
[41] A. Phansalkar, A. Joshi and L. K. John. Subsetting the SPEC CPU2006 benchmark suite, *Computer Architecture News*, Vol. 35, No. 1, pp. 69-76, March 2007.
[42] A. Phansalkar, A. Joshi, and L. K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In *Proceedings of the International Symposium on Computer Architecture (ISCA'07)*, 2007.
[43] J. S. Ringenberg. *The fast, efficient, and representative benchmarking of future microarchitectures*, Ph.D Dissertation, University of Michigan, 2008.
[44] R. H. Saavedra and A. J. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction, *Technical Report USC-CS-92-524*, Computer Science Division, University of California, Berkeley, September 1992.
[45] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications, in *Proceeding of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 3-14, 2000.
[46] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications, in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT- 2001)*, pp. 3-14, September 2001.
[47] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior, in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pp. 45-57, October 2002.
[48] F. N. Sibai. Evaluating the performance of single and multiple core processors with PCMARK®5 and benchmark analysis, *SIGMETRICS Performance Evaluation Review*, Vol. 35, pp. 62-71, 2008.
[49] D. Sorin, V. Pai, S. Adve, M. Vernon, and D. Wood. Analytic evaluation of shared-memory systems with ILP processors, in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 380-391, June 1998.

[50] SPEC. http://www.spec.org/
[51] C. D. Spradling. SPEC CPU2006 benchmark tools, Computer Architecture News, Vol. 35, No. 1, pp. 130-134, March 2007.
[52] R. Todi, *Application Signature: A New Way To Predict Application Performance*, Ph. D. thesis, Iowa State University, December 2001.
[53] R. Todi. SPEClite: using representative samples to reduce SPEC CPU2000 workload, in *Proceeding of the 2001 IEEE International Workshop on Workload Characterization (WWC-4)*, December 2001.
[54] H. Vandierendonck and K. De Bosschere. Many Benchmarks Stress the Same Bottlenecks, in *Proceedings of the Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-7)*, pp. 75-71, 2004.
[55] R. P. Weicker and J. L. Henning. Subroutine profiling results for the CPU2006 benchmarks, *Computer Architecture News*, Vol. 35, No. 1, pp. 102-111, March 2007.
[56] M. Wong. C++ Benchmarks in SPEC CPU2006, *Computer Architecture News*, Vol. 35, No. 1, pp. 77-83, March 2007.
[57] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling, in *Proceedings of the International Symposium on Computer Architecture*, pp. 84-95, June 2003.
[58] D. Ye, J. Ray and D. Kaeli. Characterization of file I/O activity for SPEC CPU2006, *Computer Architecture News*, Vol. 35, No. 1, pp. 112-117, March 2007.
[59] D. Ye, J. Ray, C. Harle, and D. Kaeli. Performance Characterization of SPEC CPU2006 Integer Benchmarks on x86-64 Architecture, in *Proceeding of the IEEE International Symposium on Workload Characterization*, pp. 120-127, October 2006.
[60] J. Yi, D. Lilja, and D. Hawkins. A Statistically Rigorous Approach for Improving Simulation Methodology, in *Proceedings of International Conference on High Performance Computer Architecture*, pp. 281-291, February 2003.
[61] J.Yi, D. Lilja., Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations, In *IEEE Transactions on Computers*, Vol. 55, No. 3, pp. 268-280, March 2006.