Phase Detection and Analysis Among Multiple Program Inputs

Rafael Mendonça Soares[1], **Luis Fernando Antonioli**[1],
Emilio Francesquini[1 2], **Rodolfo Azevedo**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Av. Albert Einstein, 1251 – Cidade Universitária – Campinas – SP – Brazil

[2]Centro de Matemática, Computação e Cognição – Universidade Federal do ABC (UFABC)
Av. dos Estados, 5001 – Santo André – SP – Brazil

{rafael.soares,luis.antonioli}@students.ic.unicamp.br
e.francesquini@ufabc.edu.br, rodolfo@ic.unicamp.br

Abstract. *Phase analysis has been shown to be an efficient technique to decrease the time needed to execute detailed micro-architectural simulations. The SimPoint method selects small but representative portions of code from each execution phase to extrapolate performance and behavior results with high accuracy and fast execution time when compared to a complete execution of the code. However, the current SimPoint technique is limited to a single program and a single input. In this work we detail an analysis technique which is able to determine phase equivalence among multiple inputs for the same program and, consequently, avoid the redundant execution of the phases. We evaluate our proposal using SPECint 2006 with multiple inputs and show that our technique, while maintaining the precision of the original approach, reduces in 32% the number of SimPoints, on average, thus also improving performance proportionally.*

1. Introduction

Performance evaluation plays a fundamental role in the development of new computational systems, specially during design space exploration. A commonly used technique consists of realistic benchmark executions using detailed software models. However, not only do newer computational systems grow more complex each year, but also benchmarks grow in complexity and size. These factors have an unavoidable effect on designers productivity when we consider the higher time needed to obtain simulation results. Several solutions have been proposed to reduce this time [Eeckhout 2010].

A well-known approach is based on the simulation of small fractions of a given program and input. One of the main challenges involved in the use of this technique is to find which portions of code best represent the full execution of a program that can be used to extrapolate its full execution performance with good precision. SimPoint [Sherwood et al. 2002] is a technique that finds these portions of code based on program execution profile, which makes it independent of the micro-architecture. To do so, dynamic instructions of the program are divided into intervals – instruction sequences of the same length. These intervals are grouped into phases with similar behavior (*e.g.*, cache behavior, branches, IPC, energy consumption). For each phase, one representative interval is chosen (SimPoint), and used for the execution of a detailed simulation.

Interval clustering is made using a structure known as *Basic Block Vector* (BBV). BBV is a vector whose elements represent each basic block (BB) of the program. Each interval is characterized by a BBV, filled with the number of times each basic block was executed weighted by the basic block size. Indeed, SimPoint's authors argue that there is a strong correlation between the paths taken by the program and the expected architectural behavior [Lau et al. 2005]. Thus, interval clustering of similar BBVs can be used to cluster intervals with similar behaviors.

Indeed, the methodology presented by Sherwood *et al.* is capable of finding *Simulation Points* for a single program-input pair, and enables designers to obtain execution results quickly and within a good margin of accuracy when compared to a complete execution. However, designers often employ multiple inputs to reach a conclusion, which makes sense when we consider the number of variables and parameters involved in the micro-processor development project space exploration. In this case, there is still the need to reduce the number of simulated instructions required in each simulation round.

In this work we improve on the original SimPoint technique to analyze a program and multiple inputs at the same time, instead of a single program-input pair. Our goal is to find a SimPoint set that is representative for more than one input for the same program. The rationale is that, if the same simulation points can be used to characterize more than one input, we can decrease the total number of SimPoints needed to estimate the behavior of a given program.

We evaluated our proposal experimentally and found out that we could reduce the number of SimPoints in 32% on average while at the same time keeping the precision of the simulation on average at 0.5% of the original. We also discovered that the new clustering results in more representative phases, allowing regions considered as outliers in the single input technique, to find a better fitting SimPoint in our proposal.

The remainder of this paper is organized as follows. Section 2 presents related work and is followed by Section 3 which formalizes the phase classification problem for multiple program inputs. Then, in Section 4 we present our proposal for the problem formalization outlined in the previous section. Experimental evaluation results are presented in Section 5 and we finally conclude in Section 6.

2. Related Work

Some researches identify program phases by inspecting microarchitecture-dependent program behavior, such as CPI and cache miss rates [Balasubramonian et al. 2000]. Although these metrics may find cross-program phase similarity, they do not allow the phase classification to be used across different hardware design. An alternative approach is to model intervals using microarchitecture-independent metrics, such as working set signatures [Dhodapkar and Smith 2002], basic block vectors (BBVs) [Sherwood et al. 2003], and memory reuse distance [Shen et al. 2004].

The authors of PinPoints [Patil et al. 2004] were the first to propose the use of BBVs for finding cross-input similarity. They presented a superficial evaluation of the effectiveness of this technique for the SPEC2000 programs with multiple inputs. They compared the delta in CPI when using a maximum of 10 SimPoints for single program-input pair against 20 SimPoints for a set of program-input pairs of the same program.

[Eeckhout et al. 2005] presents the closest phase sharing analysis when compared to our own. The authors find cross-input and cross-program similarity at the level of intervals by characterizing intervals using a set of architecture independent characteristics such as instruction mix and register dependency distance. They presented a detailed comparison of how phases are shared among inputs and programs.

The main contributions of this work are:

- We show a detailed framework for using BBVs for capturing phase sharing among different inputs of a single program
- When considering the cross-input phase behavior similarity, we proposed to increase the number of SimPoints allowed per program proportionally to the number of inputs
- We show how phases are shared among inputs using BBVs as a phase metric for comparison of intervals

3. Problem Statement

Let P be any program and $P_D = (i_1, i_2, ..., i_n)$ the instructions that were executed by P for a given fixed input D, or simply its *instruction trace*. Also consider the collection of all the static basic blocks of P, let $(BB_1, BB_2, ..., BB_r)$ be that collection. Given m, the SimPoint methodology divides P_D in disjoint subsets of size m. Let $I = (I_1, I_2, .., I_l)$ where $l = \lceil \frac{n}{m} \rceil$, the sets resulting from this division. After the nomenclature established by Sherwood *et al.* , we call these sets *instruction intervals*. For each interval $I_k \in I$, a basic block vector BBV_k is generated, such that each position $BBV_k(j)$ represents the number of times the basic block BB_j was executed in the interval k multiplied by $|BB_j|$. Let $intervals(P, e, m)$ be a function that takes a program P, an input e, and an integer m and returns the set I we described above.

The main objective of the SimPoint technique is to find a partition of I such that each set of this partition contains intervals with similar BBVs and, as consequence, present a similar architectural behavior. Let $(A_1, A_2, ..., A_k)$ be a partition of I. Each set A_i contains intervals of a program phase. The first step consists in selecting a representative interval (SimPoint) from each phase for simulation. A partition of size k implies k SimPoints. Let $(sp_1, sp_2, .., sp_k)$ be the representatives of each one of the phases. Each result of the execution metric of a SimPoint of a phase A_i is weighted by w_i according to the phase coverage size, *i.e.*:

$$w_i = \frac{|A_i|}{|I|} \tag{1}$$

If we take, for instance, CPI as a metric to be evaluated using the SimPoint methodology, the CPI of the complete execution can be estimated by the following equation:

$$CPI(P, e) \approx w_1 \times CPI(sp_1) + w_2 \times CPI(sp_2) + ... + w_k \times CPI(sp_k) \tag{2}$$

Now, consider a set of inputs $E_P = (e_1, e_2, ..., e_n)$ of program P. Let I' be the set resulting of the union of all the input intervals in E_P, *i.e.*, $I' = \bigcup_{e \in E_P} intervals(P, e, m)$.

This work finds a partition of the set I', clustering all the intervals from all inputs. The objective is to find all distinct phases of P considering the inputs of E_p. We are looking for all equivalent phases of execution common to every input.

Additionally, this work also finds a weight matrix W in which the lines represent the inputs of E_P. and the columns represent the chosen SimPoints. Thus, $W(i)(j)$ represent the weight of the SimPoint j for the input i.

$$W = \begin{array}{c} \\ e_1 \\ e_2 \\ \vdots \\ e_n \end{array} \begin{array}{c} \begin{array}{cccc} sp_1 & sp_2 & \dots & sp_k \end{array} \\ \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,k} \\ w_{2,1} & w_{2,2} & \dots & w_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \dots & w_{n,k} \end{pmatrix} \end{array}$$

Once again, if we take CPI as the metric, the complete execution of P for the inputs in E_P can be estimated using k SimPoints as:

$$\begin{pmatrix} CPI(e_1) \\ CPI(e_2) \\ \vdots \\ CPI(e_n) \end{pmatrix} \approx \begin{array}{c} \\ e_1 \\ e_2 \\ \vdots \\ e_n \end{array} \begin{array}{c} \begin{array}{cccc} sp_1 & sp_2 & \cdots & sp_k \end{array} \\ \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,k} \\ w_{2,1} & w_{2,2} & \dots & w_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \dots & w_{n,k} \end{pmatrix} \end{array} \times \begin{pmatrix} CPI(sp_1) \\ CPI(sp_2) \\ \vdots \\ CPI(sp_k) \end{pmatrix} \quad (3)$$

A trivial partition of I' can be obtained using the SimPoint methodology individually for each input. In this case, values of the line $W(e)$, $e \in E_P$ are zero for the SimPoints of the inputs in $E_P \setminus e$. To illustrate this case, take for instance a program P with an input set $E_P = \{e_i, e_j, e_k\}$. Considering that $SP(e_i) = (sp_A, sp_B, sp_C)$, $SP(e_j) = (sp_D, sp_E)$ and $SP(e_k) = (sp_F, sp_G)$, the matrix W would be:

$$W = \begin{array}{c} \\ e_i \\ e_j \\ e_k \end{array} \begin{array}{c} \begin{array}{ccccccc} sp_A & sp_B & sp_C & sp_D & sp_E & sp_F & sp_G \end{array} \\ \begin{pmatrix} 21\% & 43\% & 36\% & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 38\% & 62\% & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 77\% & 23\% \end{pmatrix} \end{array}$$

The trivial matrix produces the same results of the technique presented by [Sherwood et al. 2002]. However, this work clusters all intervals of every input, enabling us to find different phases for different inputs. In other words, a subset of a partition of I' may contain distinct input intervals. The objective is to find the phases presented by an input set and choose a representative interval for each one of these phases. This means that a SimPoint can be used to estimate the execution of two or more entries.

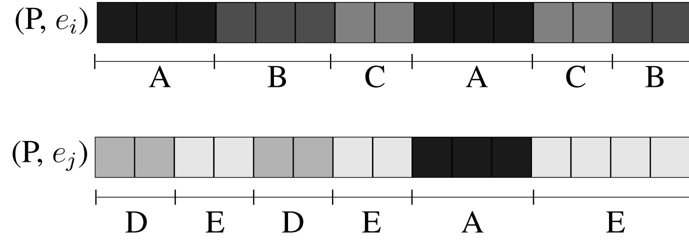


Figure 1. Phases of program P with inputs e_i e ej

Figure 1 illustrates our approach. Vertical lines show the division in intervals. The input i presents 15 intervals such that: 6 intervals were clustered in phase A, 5 intervals in phase B and 4 intervals phase C. The input j also resulted in 15 intervals clustered in phases D, E, and A. Therefore, 3 input intervals of the input j present similar behavior of the intervals of phase A of the input i, which results in the creation of a single phase A. For this example, the matrix W would have the form:

$$W = \begin{array}{c} \\ e_i \\ e_j \end{array} \begin{array}{c} \begin{array}{ccccc} sp_A & sp_B & sp_C & sp_D & sp_E \end{array} \\ \begin{pmatrix} 40\% & 33\% & 27\% & 0 & 0 \\ 21\% & 0 & 0 & 26\% & \%53 \end{pmatrix} \end{array}$$

For this particular example, the result of a SimPoint execution metric of phase A has distinct values for inputs i and j.

4. Multiple-Input Phase Classification

Our approach to find every distinct phase from a program P for an input set E_P employs the same comparison metric and clustering methodology described by [Sherwood et al. 2002]. For each interval $I_k \in I'$ a basic block vector BBV_k is generated in a way such that each position $BBV_k(j)$ represents the number of times the basic block BB_j was executed in the interval k. Then, the `k-means` clustering algorithm divides the intervals into phases based on the Euclidian distance among BBVs. Later, a single interval, the one closest to the cluster center (centroid), is chosen to represent each phase. Finally, a detailed simulation is executed using each SimPoint and properly weighted.

In the original methodology, the interval cluster size determines the weight of each phase. On the other hand, in our version the weight of each SimPoint for a given input e is defined by the number of input intervals of e contained by cluster c. Figure 2 shows a clustering example for three inputs which resulted in phases A, B, and C along with the chosen SimPoints (centroids) for each phase. In this example the weight of *SimPoint A* for *Input 1* is proportional to the number of intervals of *Input 1* present in cluster A, the weight of *SimPoint A* for *Input 2* is proportional to the number of intervals of *Input 2*, and so on.

Let $(A'_1, A'_2, ..., A'_k)$ be a partition of the set I'. Each A'_i represents a distinct phase presented by the program by the execution of one or more inputs of E_P. Formally, $w(i)(j)$ which represents the weight of the SimPoint j for the input i can be defined as:

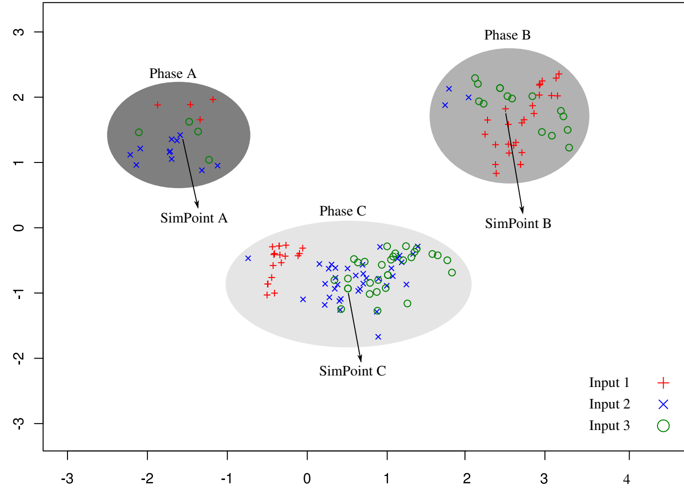


Figure 2. Phases A and B generated by the clustering of distinct input intervals of a same program.

$$w(i,j) = \frac{|intervals(P,i,m) \cap A_j|}{|intervals(P,i,m)|} \quad (4)$$

The SimPoint methodology requires information about the basic blocks executed in each program. Furthermore, workloads usually require a complex execution environment that cannot easily be reproduced on any simulator. PinPoint [Patil et al. 2004] is a tool that addresses such problems, it is built upon SimPoint methodology and Pin (a dynamic binary instrumentation framework) [Luk et al.]. It automatically (i) logs a program execution, (ii) profiles the application, (iii) detects representative regions, (iv) generates traces for the regions and (v) validates the results using hardware performance counters. In addition, these five tools can be executed independently.

For the purposes of this study, the PinPoint framework has been modified to find SimPoints among multiple program inputs. Steps (i), (ii) and (iv) were used without modification. First, steps (i) and (ii) are run for all inputs of a program. Step (ii) outputs a `.bb` file such that each line represents an execution interval and stores the number of times each basic block was executed during that interval. We then merge all the `.bb` file generated for the multiple inputs of a program. We also keep a record of the number of intervals of each input. We then apply (iii) on the merged `.bb` file.

Step (iii) creates the `.simpoints` and `.weights` files. Each simulation point in the `.simpoints` file contains the number of intervals from the first interval to reach the start of the simulation point. Based on this information and the number of intervals of each input, we are able to infer the input each interval belongs to. The `.weights` are the percentage of intervals of execution being represented by each simulation point. Since the SimPoint methodology was run on the merged BBVs those weights are useless. PinPoint also provides the information about the phase each interval has been assigned to. Based on this information we weight each SimPoint for every input according to Equation 4.

Next, we apply (iv) on every input to the generated SimPoints. Finally, in (v) we evaluate the accuracy of the SimPoints by running the whole program and the SimPoint

regions using Sniper (a x86 simulator to which PinPoint was integrated) [Carlson et al. 2011].

5. Experimental Evaluation

We analysed our proposal using SPECint 2006 reference programs that have multiple inputs `perlbench`, `bzip2`, `gcc`, `gobmk`, `hmmer`, `h264ref`, and `astar`. We used the PinPoint framework to collect all the basic block vector profiles and generate the representative regions. We evaluate our methodology using the micro-architectural simulator Sniper. After identifying program phases, we used Sniper to compute CPI for SimPoint regions and the whole program. The baseline micro-architectural model used was `nehalem-lite` which is included along with the simulator software.

We compared our technique to the original SimPoint technique, *i.e.*, we applied it to each input separately. When running PinPoint, the user has to specify three parameters: an interval size, a warm-up size and an upper limit on the number of cluster $maxK$, which is essentially the maximum number of distinct phases to be selected by SimPoint. Thus, if a program has N inputs, it will produce a maximum of $N \times maxK$ SimPoints.

To fairly compare the two SimPoint methods (single input and multiple input) we used the same SimPoint configurations of interval and warmup for both techniques. We set the interval size to 35 million instructions and warm up to 1 million instructions. We limited SimPoint's maximum number of clusters to 30 for the single input methodology. As outlined in Section 4, we found a clustering of the intervals from multiple inputs, based on the merged BBVs from all inputs, using the same clustering algorithm from the original methodology. Therefore, we also have to pick a `maxK` for this clustering. In our technique, for a program with N inputs, we defined the maximum number of clusters to $30 \times N$ and $20 \times N$.

5.1. Experimental Results

To evaluate our approach, we consider three main aspects: (i) the total number of SimPoints, (ii) the difference in precision between our approach and the original technique, and (iii) phase equivalence between the multiple inputs. The first aspect is important because it directly relates to the simulation time, while the second determines the feasibility of our approach. The third aspect, on the other hand, gives insights into the behavior of the programs when we vary the inputs and how those changes influence the generation of SimPoints.

5.2. Comparison of the Number of Simulation Points

The number of simulation points is a useful information to estimate the amount of simulation time required by both techniques. SimPoint's clustering algorithm generally picks fewer simulation points than $maxK$ (upper limit) because it usually finds a good phase characterization with fewer clusters. Figure 3 shows the number of simulation points generated for each of program in SPECint 2006 with more than one input.

Figure 3 shows that our technique is able to find a phase characterization with fewer clusters than SimPoint's original methodology. On average, our technique, compared to the original technique, used 36% fewer SimPoints for $maxK = 30$ and 47%

fewer SimPoints for $maxK = 20$ (32% fewer on average). This indicates that it can find a good phase characterization clustering of the intervals from multiple inputs of a program with fewer clusters. As result, this suggests that inputs do share phases, otherwise the number of simulation points would be close to the sum of all SimPoints applied separately.

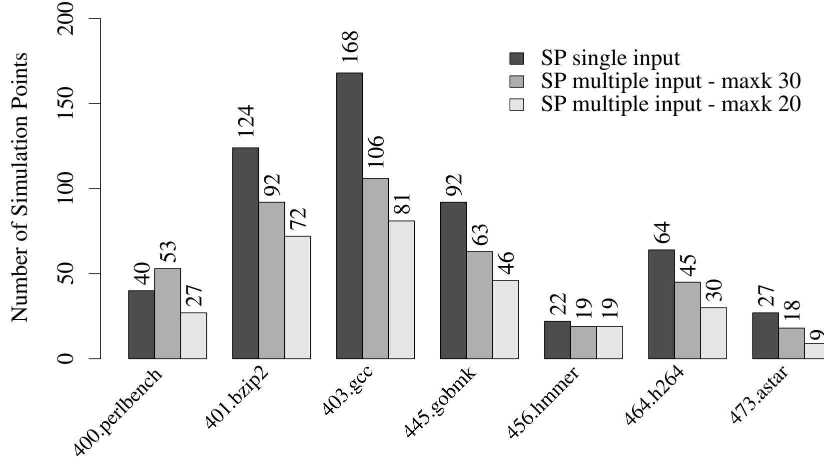


Figure 3. Number of Simulation Points

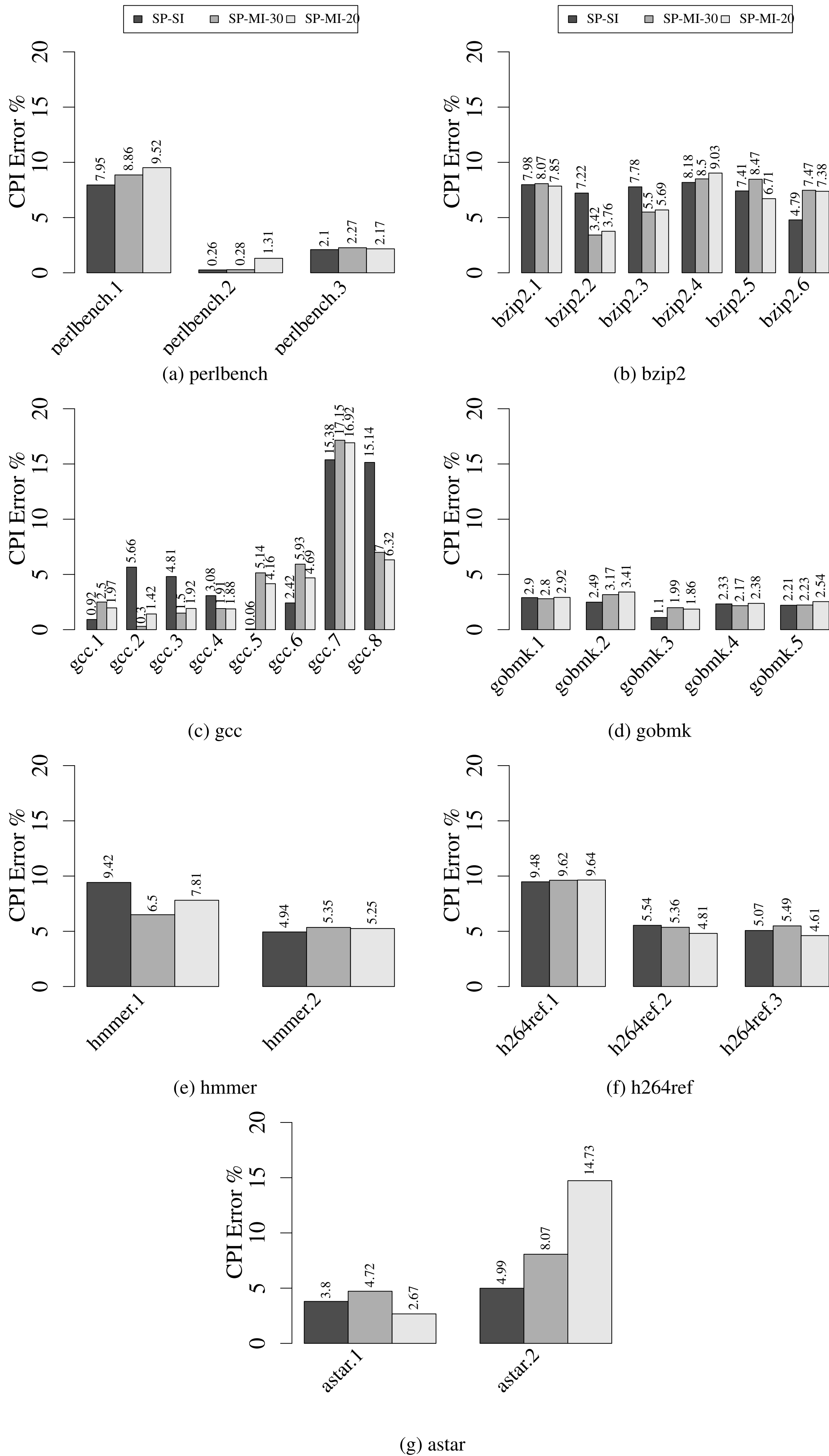
5.3. Comparison of Errors in CPI

Figure 4 shows the CPI error obtained by comparing a complete run to the results obtained from the SimPoint for single input and SimPoint for multiple inputs. The average error for a single input (original approach) is 5.30%; The average error in the analysis using multiple program inputs (our approach) is is 5.36% and 5.74% for $maxK = 30$ and $maxK = 20$ respectively. This implies that both techniques have similar errors. Therefore, this analysis suggests that we can reduce the number of simulation points used by a set of inputs of a program and get similar accuracy.

Indeed, even if the total number of SimPoints is lower, the number of SimPoints taken into consideration to estimate the results of each input is higher. This difference in the number of SimPoints does not change the total simulation time and offsets a potential increase in the error rate that could be caused by the use of more general SimPoints instead of more specific ones. Our experimental results show that the average CPI estimate error, when compared to the original approach, is 0.5% for $maxK = 20$ and negligible (0.06%) for $maxK = 30$.

5.4. Phase Sharing Analysis

In the previous sections we have shown the impact that shared phases have on the number of SimPoints and on the precision of the simulation results. Those results suggest that inputs share phases and that we can find a good phase characterization of the intervals from multiple inputs of a program. In this section we characterize how those phases are shared among inputs, *i.e.*, which phases are present in each input and their contribution to each input. To do so, we first exemplify how sharing takes place using the `astar` benchmark.



(a) perlbench

(b) bzip2

(c) gcc

(d) gobmk

(e) hmmer

(f) h264ref

(g) astar

Figure 4. CPI evaluation for single input and multi-input ($maxK = 30$ and $maxK = 20$)

Figure 5 shows how the weights of the SimPoints differ for distinct inputs, in percentages of the execution time. The top bar represents the phases of the first input, and their weights. Only six phases play relevant role for input 1 execution. The bottom bar shows the second input. Although the light green phase (rightmost) covers more than half of the program execution, this input takes the program through more distinct phases than input 1. Recalling Figure 3, by sharing several SimPoints, we could reduce their total number (from 27 to 18, with $maxK = 30$) but still keep a similar precision.

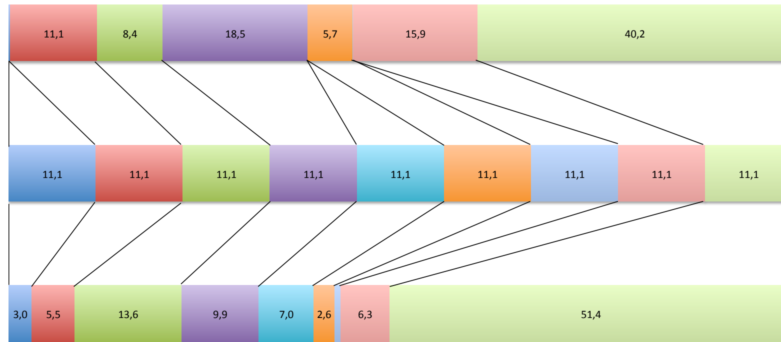


Figure 5. Distribution of SimPoints weigths for two inputs of the `astar` benchmark: input 1 on top and input 2 on bottom. Smaller weights omitted due to space restrictions.

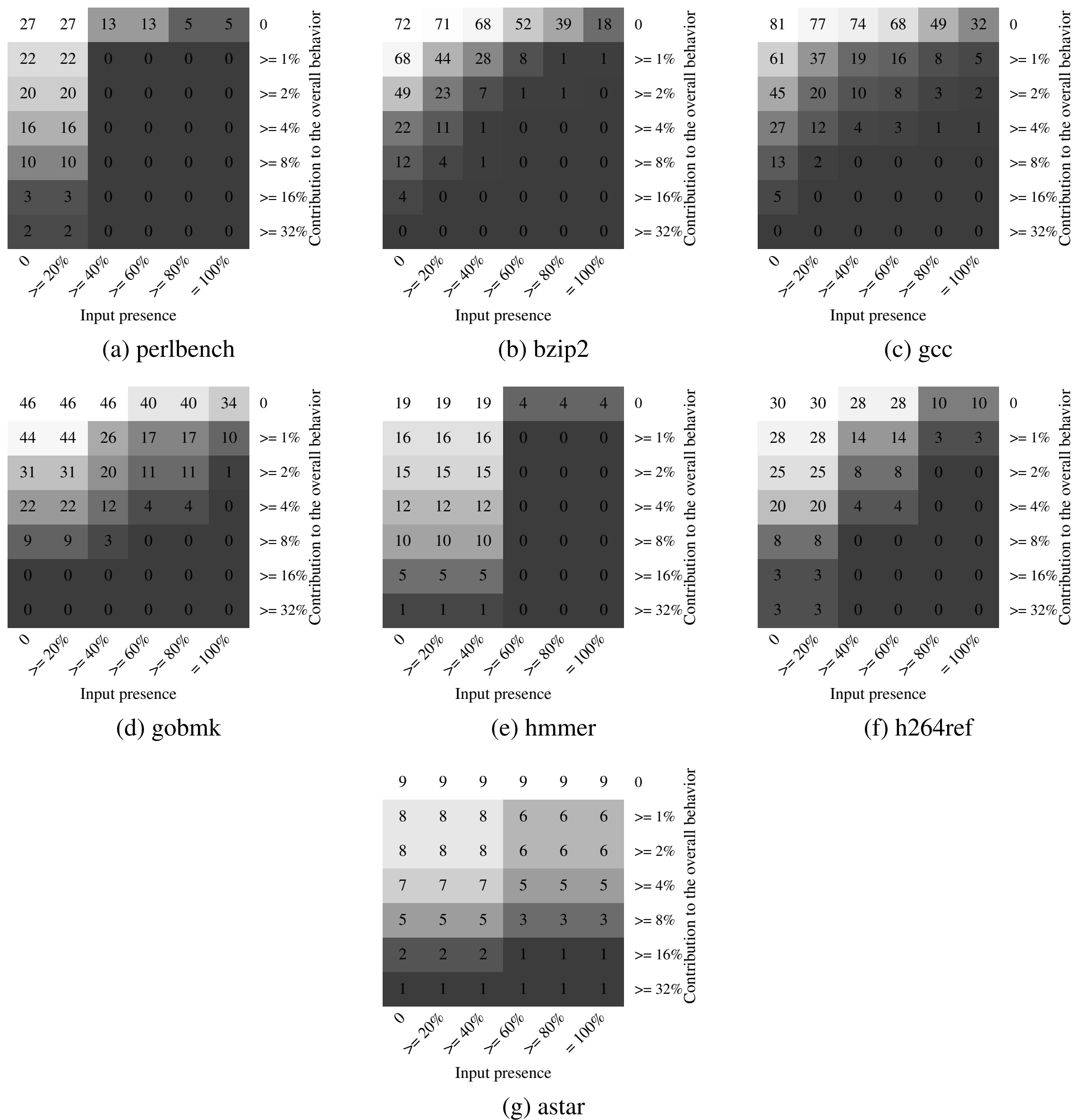
To quantify phase sharing, we characterized the number of phases and the percentage (in 20% increments) of the inputs that actually use them. Figure 6 shows our experimental results. For instance, `gcc` has 32 phases that are present in all inputs, 49 phases that are present in at least 80%, 68 phases present in at least 60%, and so on. The remainder of the results can be seen on the first line of each subfigure. Figure 6 also shows the contribution (weight) of those shared phases to the overall behavior of the inputs. These results are shown in lines (note the log axis). For instance, even though `gcc`'s inputs share 32 phases, only one of those phases covers every input with a contribution of at least 4% (4th line), and none has a contribution of at least 32%.

The heatmap shown in Figure 6 also allows us to quickly observe the difference in phase sharing behavior between the benchmarks. For instance, `astar` has, proportionally, a larger number of phases shared among the multiple program inputs which can be seen by the smaller number of dark cells on its heatmap. In particular, `astar` has one phase that is present in all inputs with a contribution of more than 32%, which is not the case for any other evaluated benchmark.

6. Conclusion

SimPoints have been extensively used to decrease simulation time and thus improve the productivity of processor designers. However, the original technique proposed by Sherwood *et al.* considers each program-input pair separately, forcing their users to employ a distinct set of SimPoints for each input of the same program. For this reason, the original technique ends up effectively disregarding similarities that are bound to exist among multiple executions with distinct inputs of the same program.

In this work we show that, when these similarities are taken into consideration, we can decrease the time needed to obtain simulation results even further. In particular, using SPECint 2006 we show that the number of SimPoints (which is directly proportional to simulation time) can be reduced on average 32% (in the specific case of `astar` we achieved a 66% reduction with $maxK = 20$). We also show that, even if the total number of SimPoints is lower, the number of SimPoints taken into consideration to estimate the results of each input can be higher, thus compensating for a possible increase in the error rate caused by the use of generic SimPoints instead of specific ones. In effect, we show that the average CPI estimate error when compared to the original approach is 0.5% for $maxK = 20$ and negligible (0.06%) for $maxK = 30$.



(a) perlbench (b) bzip2 (c) gcc

(d) gobmk (e) hmmer (f) h264ref

(g) astar

Figure 6. Heatmap showing how many phases are present and their coverage

7. Acknowledgments

We would like to thank CNPq, Capes (PROCAD 2966/2014) and FAPESP (2014/17925) for their support.

References

Balasubramonian, R., Albones, D., Buyuktosunoglu, A., and Dwarkadas, S. (2000). Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO-33 2000*, pages 245–257.

Carlson, T. E., Heirman, W., and Eeckhout, L. (2011). Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11.

Dhodapkar, A. S. and Smith, J. E. (2002). Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02.

Eeckhout, L. (2010). *Computer architecture performance evaluation methods*. Morgan & Claypool Publishers, San Rafael, CA, USA.

Eeckhout, L., Sampson, J., and Calder, B. (2005). Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 2–12.

Lau, J., Sampson, J., Perelman, E., Hamerly, G., and Calder, B. (2005). The strong correlation between code signatures and performance. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, ISPASS '05, pages 236–247. IEEE Computer Society.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*.

Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., and Karunanidhi, A. (2004). Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37.

Shen, X., Zhong, Y., and Ding, C. (2004). Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI.

Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. (2002). Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57.

Sherwood, T., Perelman, E., Hamerly, G., Sair, S., and Calder, B. (2003). Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93.