Abstract

Modern computer systems have become fairly complex in nature; they require long development cycles and often require thousands of engineers to build and verify. In such a setting, design from scratch (ab initio design) has been replaced by a modular design process, where prebuilt components possibly sourced from third parties are used as much as possible. The role of developers is limited to system integration and coding parts of the system that are particularly novel. Given the extensive verification effort that is involved these days along with concerns about reliability and security, vendors are hesitant to make revolutionary changes in hardware/software offerings. The trend is to make evolutionary changes – small improvements with disproportionate benefits. The classical method to do such research is to thoroughly analyze a system, find and analyze bottlenecks, and then try to propose a better solution. However, the benefits of such approaches are quickly diminishing and given the scale of today's systems starting from smart watches to servers, bottom up approaches are proving to be very challenging.

Hence, we propose to look at comparative approaches where we compare the execution of two systems keeping a large part of the execution environment the same. The comparative analysis can help us pinpoint which design choices exactly lead to better performance in a set of workloads. We choose two concrete problems in this space and show that such comparative approaches yield solutions that significantly outperform the state of the art. The problems are ISA design and OS design.

We start with a quintessential problem: design an ISA for a workload with realistic constraints in mind such as minimal changes to an existing ISA. We propose a simple graphical approach that helps us nicely visualize the execution of a program. It helps us identify the issues with an ISA and also what custom instructions can be added to disproportionately increase the performance of the program. We followed up with creating a vector representation of such diagrams that includes FFTs of the equivalent 2D image and other features. We show that we can train a learner on these representations and we can then use this to perform performance estimation (for three different ISAs: x86s, ARM, and RISC-V). Such diagrams are very useful in *static single-ISA and cross-ISA performance estimation. We significantly outperform competing work in this space. We increase the performance by 10-28% of basic RISC ISAs such as RISC-V by adding 2-10 new instructions and at the same time reduce the number of infructuous and trivial suggestions made by custom ISA generators by 5-40 times. Our average error for the performance estimation is less than 2%. The prediction time of our algorithm is* $5\times$ *faster than the state of the art.*

Some of this evaluation is done on an architectural simulator because it is necessary to keep the microarchitecture the same. That is when we realized that simulation technology has several shortcomings when it comes to doing such work because it does not simulate OSes correctly. We thus extended our architectural simulator Tejas to support correct and fast OS simulation. We needed to add special support for recognizing and simulating events of interest such as DVFS and DMA requests, Timer and I/O interrupts along with a fast-forward mode of simulation. The speedup in the case of our algorithm is $17\times$ *higher than SimPoint for the Linux OS. Similarly, we are* $16\times$ *and* $51\times$ *faster as compared to SimPoint for the FreeBSD and the OpenBSD OSs respectively.*

We then solved the second problem, which was automatically comparing the execution of large software on both real and simulated hardware and understanding the reasons for performance differences (if any). The motivating example here is that for the same workload, we found out that the performance difference across Linux and FreeBSD (compiled with the same libraries and compiler) can be as high as 60%. We wrote an automated tool SoftMon *that analyzes the execution of such workloads by performing extensive graph analyses, comment mining using NLP tools, and using custom heuristics. In less than 200 seconds, our tool can pinpoint a superset of 5-10 functions that are responsible for a difference in performance when a full-scale workload is run on regular operating systems: Linux, FreeBSD, and OpenBSD. We repeated the same exercise on commercial software and got many insights regarding the correct design choices for workloads. SoftMon saved 500 man hours of analysis, and found 25 reasons for 6 categories of large open-source programs.*

Comparative analyses techniques have great promise: they are vital tools for analyzing, understanding, and optimizing the performance of workloads.