Modern computer systems have become so complex that understanding and predicting the performance of programs is a significant challenge. For instance, when designing microprocessor architectures, engineers must assess the trade-offs involved in allocating the on-die real estate (e.g., between the L1 cache, execution units, etc.) in order to achieve certain performance and power consumption targets.

Typically, an experimental hardware architecture is emulated in software, which is extremely computationally intensive. Assessing the hardware’s runtime behavior, often referred to as its runtime profile and measured by cycles per instruction executed (CPI) or cache miss rate, requires cycle-accurate simulation at the functional level of the hardware. Consequently, researchers have resorted to picking portions of the program execution that are considered typical, called simulation points, and extrapolating from the detailed simulation of these points to the entire runtime profile.

State-of-the-art algorithms attempt to strategically select a small set of simulation points that are characteristic of larger portions of program execution by exploiting the phase structure of a program. The idea underlying phase structure is that the execution dynamics of a large program can be understood in terms of a relatively small set of distinct patterns of program behavior. The execution trace is broken into short-duration intervals, and the goal is to assign a phase label to each interval such that the program behavior across all intervals with the same label is similar.

Phases are determined from statistics that can be easily and quickly collected as the program executes at the instruction level (i.e., hardware simulation is not necessary), such as counts of basic blocks executed. A basic block is a sequence of instructions that has a single entrance and exit point. An exit point, signaled by a branch instruction, causes program execution to jump to another section of program code. In a high level language, branch instructions include function calls, if/else constructs, looping constructs, etc.

Program execution is segmented into intervals containing the execution of \( n \) basic blocks. Each interval can be summarized by a basic block vector (BBV), an \( m \)-dimensional vector that specifies the frequency of execution of basic blocks, where \( m \) is the number of unique basic blocks. In contrast to the program’s runtime profile, BBVs can be collected at little cost using fast software instrumentation techniques.

Currently, the state-of-the-art algorithm for selecting simulation points, called SimPoint [3], uses \( k \)-means clustering of basic-block vectors to obtain \( k \) distinct phases. Subsequently, a multinomial mixture model has been used for phase discovery [2, 1]. These methods, however, ignore one key domain constraint: the BBVs are sequentially ordered.

We thus consider a class of dynamical probabilistic models that explicitly captures the sequential structure of the BBVs. Our expectation is that by modeling the sequential structure, we will obtain a more faithful representation of program execution phases, and consequently, will better estimate the program’s runtime profile. We explore hidden Markov models, in which the observation at each time step corresponds to a BBV, and the hidden state corresponds to the program’s latent phase. We explore two alternatives for the emission distribution: a multinomial and a Gaussian. We also explore a range of techniques for reducing the dimensionality of the basic-block vectors, picking simulation points, and estimating a particular profile metric given the simulation points.

We searched a large model space, but we found no variant that significantly outperforms \( k \)-means (used in SimPoint). Although our result is negative, it is consistent with several recent studies that also failed to beat \( k \)-means using probabilistic models. Our contribution is in specifying a general probabilistic framework that lays out a large space of possible models, and in a systematic exploration of this space. Without this type of exploration, other researchers are likely to waste effort attacking one or another point in the space.
In particular, we analyze the reasons for why \( k \)-means is as robust as it is, even when it ignores the sequential structure of BBVs, in contrast to the HMM, which has the ability to exploit this constraint. The explanation for this result does not lie in the fact that HMM state transitions are infrequent: transitions to a different state occur on about a third of time steps. The explanation also does not lie in the fact that state transitions have no structure: the entropy of the conditional state distribution is low. We therefore conjecture that the dynamical HMM shows little benefit because the observations (BBVs) are unambiguous: they provide a strong constraint on state. We provide evidence in support of this conjecture.

Despite the fact that we did not improve on SimPoint for estimating runtime profiles, our HMM offers a significant benefit: a graphical depiction of the runtime structure of a program—the finite-state machine representation of the HMM. This graphical description can be used to understand program behavior, and can be exploited, via phase prediction, for run-time optimization.

For example, Figure 1 shows finite-state machine graphs that summarize the inferred structure of the execution of a run of a compression/decompression test suite (gzip). Panels (a)-(e) correspond to increasing numbers of hidden states (phases) allowed in the model. The general structure of the 5 graphs is similar: there are three or four highly interconnected clusters with fairly sparse transitions. By linking these functional clusters with their corresponding source code, an understanding of dynamical program execution can be obtained, and used to target source code optimizations.

Acknowledgments

This research was supported by NSF BCS 0339103 and NSF CSE-SMA 0509521.

References

