A case for code-representative microbenchmarks

Calvin Bulla, Miquel Moreto Barcelona Supercomputing Center
{calvin.bulla,miquel.moreto}@bsc.es

***Abstract*—Microbenchmarks are fundamental in the design of a microarchitecture. They allow rapid evaluation of the system, while incurring little exploration overhead. One key design aspect is the thermal design point (TDP), the maximum sustained power that a system will experience in typical conditions. Designers tend to use hand-coded microbenchmarks to provide an estimation for TDP. In this work we make the case for a systematic methodology to automatically generate code-representative microbenchmarks that can be used to drive the TDP estimation.**

I. Introduction

The thermal design point (TDP) is a key aspect in the design of a microarchitecture. It indicates the maximum sustained power dissipation that a system will experience during typical runtime conditions and constrains the design of the cooling system. It is critical to have a good TDP estimation. If it is too low, power-hungry workloads will get throttled, which negatively impacts the overall system performance. If it is too high, we incur an increase in terms of packaging and cooling cost. As a consequence, a TDP estimation that is representative of real workloads is a main goal during the design process.

In a typical design environment it is infeasible to execute a wide range of applications to come up with a realistic TDP. Simulating cycle-intensive applications is a serious time investment, which prevents rapid design space exploration in the early stages of development. As a consequence, the industry relies on microbenchmarks to single out and evaluate the crucial parts of real applications. These microbenchmarks are often hand-crafted, L1-contained kernels with a low branch misprediction to maximize power dissipation and provide a conservative TDP estimation. Creating these microbenchmarks requires a significant engineering effort, and knowledge of the underlying architecture as well as the common workloads.

Previous work has successfully highlighted the benefit of microbenchmark generators for a wide range of purposes: cloning program behavior [5], [6], power profiling [4], [7], or to generate synthetic tests [1]. In this work we make the case for a framework to automatically extract code-representative microbenchmarks from real applications, which can then be used to effectively drive the TDP.

In this context, we have set the following goals:

- The generated microbenchmarks should provide a safe upper-bound for power. Therefore, we will force them to be L1-contained, with predictable branches.
- The generated instruction sequences should be code-representative, i.e. occur like this in the original benchmark.
- The methodology should be easily extendable, and applicable to a variety of architectures, without being tied to a specific simulation infrastructure.

II. Framework

Fig. 1 gives an overview of the workflow for our framework. It can be divided into four stages: 1) Execution Tracing, 2) CFG Annotation, 3) Snippet Selection, and 4) Microbenchmark Synthesis.

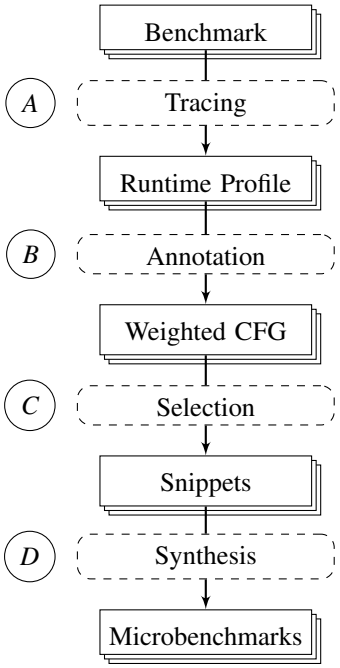


Fig. 1: Overview of our proposed workflow.

A. Execution Tracing

We first trace the execution of the selected benchmark. To this end, we have developed a sampling based profiler using the `perf_event` interface. At regular intervals, we gather the current program counter (PC), as well as a set of configurable performance monitoring counters (PMCs), such as the branch and cache miss/hit ratio.

B. CFG Annotation

In this step we use the obtained runtime profile to annotate the control flow graph (CFG) of the application. A CFG is a graph representation of the program's executional flow. Each node represents an uninterupted sequence of instructions, or basic block (BB), each vertex the corresponding branch or fall-through at the end of a BB.

With the sampled program counter as key, we associate each sample to a specific BB. We then weigh each BB, based on the number of samples it generates. The measured PMCs can be used as an additional heuristic to increase the weight of impactful BBs.

C. Snippet Selection

Using the CFG annotation we select a subset of the heaviest BBs. In our initial approach, we ignore all BB, which sample count is a below an α threshold ($0.1 - 1.0\%$ in our experiments).

We can now generate code snippets for each selected BB (seed). A code snippet is a short sequence of BBs between a loop (or function) entry and exit point. We form these snippets by finding all acyclic paths between the entry and exit BB that go through the seed. For each BB we only select the snippet with the heighest accumulated weight. The snippets are validated against the instruction trace of the original application.

D. Microbenchmark Synthesis

Finally, we pass the snippets to a MicroProbe-based [2] backend that generates standalone executables. To this end we take the snippet kernel and embed it into a new loop. All branches are modified so that they directly jump to the next BB in the snippet chain. Where necessary, memory references are changed so that they access a previously allocated, L1-contained, memory region. The generated assembly code now can be compiled for the target architecture.

III. Evaluation

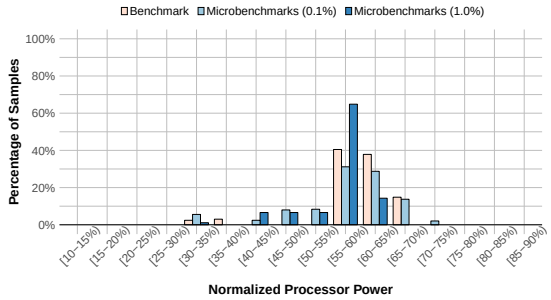
We have tested our first approach on an IBM BladeCenter PS701 system, featuring an eight-core IBM POWER7 processor running at 3.8 GHz. The benchmarks have been selected from the SPEC CPU 2006 benchmark suite. We execute each benchmark in isolation, and use our toolchain to generate the corresponding microbenchmarks with a threshold $\alpha = 0.1$ and 1.0. We then execute the created microbenchmarks, and compare their power dissipation with the original application. We use AMESTER to access the on system power meters each millisecond. The shown values are normalized with respect to the maximum observed system power.

Fig. 2 shows the power histogram for two of the selected benchmarks and the corresponding microbenchmarks with different thresholds. We plot the percentage of samples/snippets (y-axis) observed for each power range (x-axis). We can observe that the original benchmarks tend to have two to three dominant power ranges, corresponding to the main kernels of the application. In the case of perlbench (a), our approach is able to replicate both spikes. As we decrease the threshold, our framework will consider more snippets and better approximate the original application. For zeusmp (b), a lower threshold produces an even more notable change in proportion of sampled power regions. We plan to investigate the most appropriate threshold in the future. It is important to note, that in all cases, we are able to provide a safe upper bound for power.

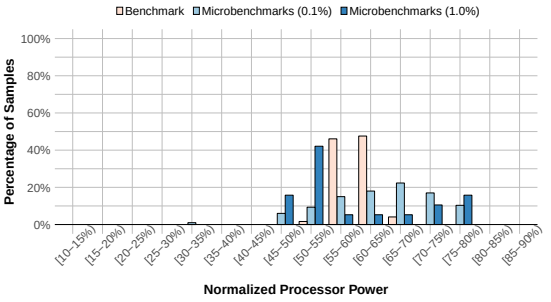
IV. Conclusion

In this work we have highlighted the need for microbenchmarks in estimating the TDP of a microarchitecture design. We have also presented our initial approach to automatically extract code-representative microbenchmarks from real applications using runtime profiling and CFG analysis. Finally, we present preliminary results on a POWER7 system.

For future work we plan to further test our methodology and apply it to parallel workloads and in the context of runtime-aware architectures [3], [8]. We also consider expanding the snippet selection phase with more sophistcated heuristics. Lastly, we plan to port our framework to different architectures and release it as an open-source toolchain.



(a) 400.perlbench



(b) 434.zeusmp

Fig. 2: Power histograms for benchmarks and their corresponding microbenchmarks

References

[1] 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*, ICS '05, pages 111–120, 2005.

[2] R. Bertran, A. Buyuktosunoglu, M. S. Gupta, M. Gonzalez, and P. Bose. Systematic energy characterization of CMP/SMT processor systems via automated micro-benchmarks. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, 2012.

[3] M. Casas, M. Moretó, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. Unsal, A. Cristal, et al. Runtime-aware architectures. In *European Conference on Parallel Processing*, pages 16–27. Springer, 2015.

[4] C.-T. Hsieh and M. Pedram. Microprocessor power estimation using profile-driven program synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(11):1080 –1089, Nov 1998.

[5] A. Joshi, L. Eeckhout, R. Bell, and L. John. Performance cloning: A technique for disseminating proprietary applications as benchmarks. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 105 –115, Oct. 2006.

[6] A. Joshi, L. Eeckhout, R. H. Bell, and L. K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Transaction of Architecture and Code Optimization*, 5:10:1–10:33, September 2008.

[7] L. Mukhanov, D. S. Nikolopoulos, and B. R. de Supinski. Alea: fine-grain energy profiling with basic block sampling. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 87–98. IEEE, 2015.

[8] M. Valero, M. Moreto, M. Casas, E. Ayguade, and J. Labarta. Runtime-aware architectures: A first approach. *Supercomputing frontiers and innovations*, 1(1):29–44, 2014.

Calvin Bulla received his BSc degree in Computer Science from Universidad Las Palmas de Gran Canaria in 2015. Now, he is enrolled in the Master's degree of Innovation and Research in Informatics at Universitat Politècnica de Catalunya (UPC) with a mention in High Performance Computing. He joined the RoMoL team at the Barcelona Supercomputing Center in September 2015.

His research focuses on runtime-aware architectures, performance analysis, and system profiling. Currently he is developing a framework to trace applications and extract standalone microbenchmarks. These can then be used to guide the design characteristics of future microarchitectures, such as the thermal design point.