Performance Prediction of Memory Access Intensive Apps with Delay Insertion: A Vision

Soramichi Akiyama, Takahiro Hirofuchi, Hirotaka Ogawa
Artificial Intelligence Research Center,
National Institute of Advanced Industrial Science and Technology (AIST), Japan
Email: {s.akiyama, t.hirofuchi, h-ogawa}@aist.go.jp

***Abstract*—Predicting performance of a given program on a given machine is highly important because the environment where the program is developed and the one where it is actually executed are often different. However, this prediction is also difficult because the performance of the same program on different machines is not the same, due to the different balances in performance of the various computer components (e.g. CPU, memory, etc.). Although many studies tackle this problem by modelling the target program and/or the target machine, model-based techniques can only provide what they model and cannot leverage existing performance analysis tools. In this paper, we tackle this problem by actually executing the target program in an emulated environment, where the performance balance of the CPU and the memory subsystem is virtually tweaked using a dynamic binary instrumentation technique. We show that this approach can emulate the total execution time of a memory-access-intensive application on different machines, and provide a vision of the future, showing how our approach can outperform existing model-based approaches.**

I. Introduction

A. Performance Prediction

Predicting the performance of a given program in a target environment is important to achieve the expected performance of the program, because the environment where an application is developed and the one in which it is executed are often different. For example, Artificial Intelligence (AI) applications are often developed on a local machine convinient for interactively choosing a large number of parameters (e.g. the number of intermediate layers and the connectivity between them in neural networks), and then executed on a powerful server to deal with big data, such as 1.28 million images [1] or 252 million elements of an input matrix [2]. However, predicting the performance is difficult at the same time because *the performance of the same program on different machines differ greatly*, because different machines can show a different performance balance among their components. For example, Hennessy and Patterson [3] shows in Figure 5.2 of page 289 that the performance balance of processors and memory subsystems is not constant, because processors have been speeding up relative to memory subsystems ever since 1980s, due to evolutionary differences in speed.

B. An Example of Performance Difference

Differences in the performance balance between processors and memory subsystems largely affects the performance characteristics of memory-access-intensive applications. Figure 1 shows the performance of a Matrix Factorization application on three different machines. The program uses the alternate least square (ALS) [4] algorithm and is written in Python. Matrix Factorization decomposes a large sparse matrix R into a product of two smaller dense matrices A and B ($R \approx A \times B$), and the size of R used in this experiment is 8K $\times$ 8K. The x axes show the number of processes used for calculation and the y axes show the elapsed time required to finish one iteration of ALS, which typically needs tens of iterations to converge. The solid lines show the actual execution time obtained on each machine, and the dashed lines show the expected performance (linear speedup):

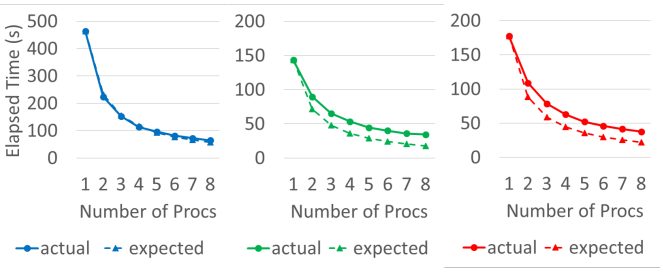

Fig. 1. Actual and Expected Execution Time of Matrix Factorization on Different Machines (two lines are overlapped in the left-most figure)

$$y_{\text{expected}}(x) = \frac{y_{\text{actual}}(1)}{x}. \quad (1)$$

The values in the three graphs are measured on machine 1 and 2 in Table I and a large shared-memory machine with 16 Xeon E7-8867 v3 processors and 12TB of memory, respectively from left to right. These results clearly show that the performance characteristics of the same program differ depending on the underlying machines. The blue lines (left) are almost overlapped with each other, and that means the program scales up to 8 processes completely on this machine; while the green (middle) and red (right) lines show that the same program does not scale well on these machines. We confirmed similar trends with the CG and BT benchmarks of the NAS Parallel Benchmarks [5], and a C program for large matrix-vector multiplication (known to be memory-access-intensive) that we use in Section III for preliminary experiments.

9781-5090-1445-3/16$31.00 © 2016 IEEE (CloudCom'16)

C. Our Goals and Contribution

Our goal is to improve performance prediction in different environments by running the actual code of a given program in an emulated environment, rather than using abstracted performance models that have been widely researched already. The aim of this approach is to draw out richer and more integrated information for performance prediction than model-based techniques provide.

The common underlying idea of model-based techniques is to retrieve one or several performance metrics efficiently by abstracting the target program and/or the target machine. However, a model-based technique only predicts the metrics that the model focuses on, and it is difficult to use the predicted metrics in any way that the model does not support. For example, DraMon [6] predicts the memory bandwidth usage of multi-threaded programs without actually running them by abstracting the memory access pattern from the actual code. Although this method is efficient to know the memory bandwidth usage, what if the user wants to know the chronological change pattern of the memory bandwidth usage? In this case the user has to modify DraMon to collect the chronological pattern, because simply combining another model that extracts the chronological change pattern of a given program is difficult due to the difference of the abstraction level of the two models.

To mitigate this shortcoming of model-based techniques, our goal is to allow users to do whatever analysis they want in an emulated environment where the performance balance of the target machine is emulated. In the memory bandwidth example above, the user can run a normal profiler to retrieve the bandwidth usage and its chronological pattern at the same time. As the first step toward this goal, the contributions of this paper are as follows:

1) We describe our system that emulates the performance balance of the CPU and the memory subsystem of the target machine using the dynamic binary instrumentation technique with QEMU and show that it can predict the total execution time of a memory-access-intensive program.
2) Although emulating total execution time does not outperform what model-based techniques provide, we discuss two concrete examples on how our approach can be the basis of richer performance analysis in the future than model-based techniques.

II. Proposed System

A. Overview

The main idea of our work is to emulate the performance balance of the target machine and execute the target memory-access-intensive application in the emulated environment. We achieve this by tweaking the performance balance of the CPU and the memory subsystem virtually. Technically, we insert a small amount of delay after the memory accesses of the target program at runtime.

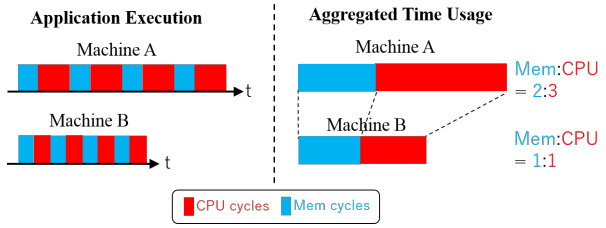

Fig. 2. Different Performance Balance between Machines

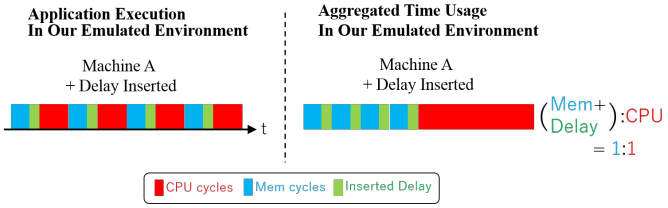

Fig. 3. The Main Idea: Delay Insertion

Figure 2 illustrates how the performance of a memory-access-intensive application differs on two machines with different performance balances. The left-hand side shows a breakdown of the execution time into CPU-cycles, during which the CPU is actually doing some calculation, and Mem-cycles, during which the CPU is stalled to wait for data accesses to be served by the memory subsystem. The right-hand side shows the aggregated time each cycle uses, and the ratio of time CPU- and Mem- cycles use against the total execution time are different between machine A and machine B, because of the difference of the performance balance of their CPUs and memory subsystems.

Figure 3 shows how delay insertion can emulate the performance of the given application for differente performance balance of the CPU and the memory subsystem. By inserting sufficient amounf of delay after every memory accesses, the aggregated ratio of time that CPU- and Mem- cycles use become the same as the ones on the target machine.

B. Proof-of-Concept Implementation

There are several possible ways to implement a delay insertion after memory accesses:

1) **Hardware-based mechanisms** install a special hardware module into the emulating machine. This approach can yield the lowest performance overhead for the target program, but installing extra hardware reduces the practicality.
2) **Compiler-based mechanisms** insert delays at the compile time and also yield small amount of performance overhead, but a new compiler has to be built for every target language. Since pursuing performance is also common for non-HPC programmers today, a uniform mechanism for many target languages is preferable.
3) **Dynamic binary instrumentation** inserts delays at the runtime by dynamically modifying the executed binary. This method is most suitable for our purpose because

it is independent of the programming language and the performance overhead caused by the instrumentation is acceptable, since we focus on the performance **balance**, not actual values.

We use the user-mode of the QEMU hardware emulator as the basic dynamic binary instrumentation mechanism we rely on. In the user-mode, QEMU can execute a binary of any supported architecture on a machine of any supported architecture (for example, it can run MIPS binaries on x86 CPUs). The user-mode first disassembles the binary of a given program and translates it into QEMU-specific intermediate instructions. After performing some optimization on the intermediate instructions, QEMU translates the intermediate instructions back into the target architecture instructions. The output code is stored in the JIT cache and reused when the same region of the program is executed again (disassembling and translating back and forth are avoided when the JIT cache hits).

Our system inserts a small number of NOP instructions after every load- and store- related instructions when QEMU translates intermediate instructions back to target architecture instructions. Inserting NOPs changes the size of the output executables, but we do not need to care about the jump addresses embedded in the binary because they are automatically replaced by QEMU itself. By inserting NOPs at this stage, we do not need to consider many kinds of CISC instructions because they are merged into a small number of intermediate instructions (for example, x86 has 10s of mov-related instructions but QEMU has only four depending on if it's a load or a store and the target data size). This not only allows easy implementation but also allows adapting the system to multiple instruction set architectures, some of which are emerging thanks to their energy-efficiency compared to x86.

An advantage of using QEMU compared to using other dynamic binary instrumentation tools such as Intel PIN [7] or Valgrind [8] is that QEMU can run even a whole operating system under dynamic binary instrumentation in its system-mode. Although in this paper we only use the user-mode, this gives our mechanism a chance to be applicable for predicting the performance of a whole operating system in the future. Note that in the current version (2.6) of QEMU it only uses single thread of the host operating system when running a guest operating system with dynamic binary instrumentation mode, but there are active discussions on allowing it to use multiple host threads (e.g. [9]).

III. Preliminary Experiment

We show our system can tweak and emulate the performance balance of underlying machines virtually. Table I shows the hardware specification of the machines. For software, Debian GNU/Linux 8.5 and QEMU 2.6 are used. Note that the CPU and the DRAM are from different generations, thus the performance balance is also different between the two machines.

TABLE I
Hardware Specifications

	Machine 1	Machine 2
CPU Model	Xeon E5-2603 v**0**	Xeon E5-2699 v**3**
CPU Cores	8 (4 cores × 2 sockets)	18 (1 socket)
CPU Micro Arch.	Sandy Bridge	Haswell
Last Level Cache	10 MB each	45 MB
Memory Model	DDR3 1600 MHz	DDR4 2133 MHz
Memory Ch.	4 channels	4 channels

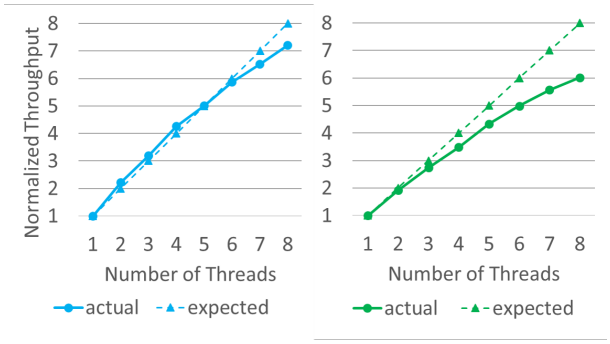

Fig. 4. Normalized Throughput of Matrix-Vector Multiplication on Machine 1 (Left) and Machine 2 (Right)

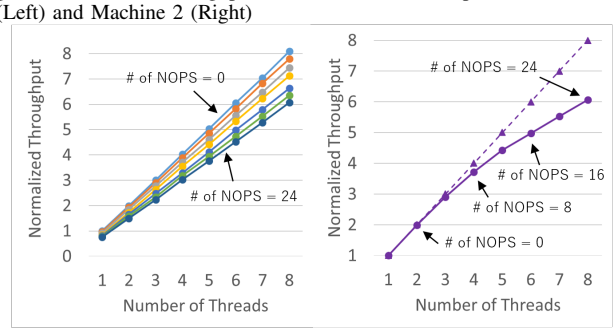

Fig. 5. Normalized Throughput of Matrix-Vector Multiplication in the Emulated Environment. Left: Each line corresponds to the number of NOPs from 0 to 24 incremented by 4. Right: Each point from $x = 2$ to $x = 8$ corresponds to the number of NOPs from 0 to 24 incremented by 4 (# NOPs = 0 at $x = 2$, # NOPs = 4 at $x = 3$, ..., # of NOPs = 24 at $x = 8$).

Figure 4 shows the normalized throughput of a matrix-vector multiplication program (well-known to be memory-access-intensive). The size of the matrix is 8K × 8K and is larger than the LLC of both machines. The program is written in C and it uses multiple threads by dividing the input matrix, and each thread calculates a part of the results. The throughput is the number of multiplications done in each time step, thus the expected throughput increases linearly as the number of threads increases. Note that throughput is the inverse of total execution time and is equivalent to it. The lines on the left (blue) show the results on machine 1, and the ones on the right (green) show the results on machine 2. The values are the average of three runs. The program scales better on machine 1, which is as expected, because machine 1 is older than machine 2 and the performance gap between the CPU and the memory subsystem is smaller.

TABLE II
NORMALIZED PERFORMANCE ON MACHINE 2 (NATIVE) AND ON MACHINE 1 WITH DELAY INSERTED (EMULATED ENVIRONMENT)

# of Threads	1	2	3	4	5	6	7	8
Native	1	1.92	2.73	3.49	4.33	4.99	5.57	6.02
Emulated	1	1.99	2.90	3.71	4.42	4.98	5.53	6.07
# of NOPs	0	0	4	8	12	16	20	24

The left-side of Figure 5 shows the normalized performance when the same program is executed with our QEMU on machine 1. We call the combination of machine 1 and the delay-inserting QEMU *the emulated environment*. The number of NOPs inserted is changed from 0 to 24, incremented by 4 (there are $24 \div 4 + 1 = 7$ lines) and all values are normalized by the value at $x = 1$ with 0 NOPs. Note that the y axis starts from 0 (not 1) because the values at $x = 1$ for number of NOPs > 0 are less than 1. A very important finding here is that, in the emulated environment, the throughput scales linearly with regard to the number of threads and it degrades linearly with regard to the number of inserted NOPs. This is because the execution with dynamic binary instrumentation is much slower than the original program, and the performance bottleneck in the memory subsystem is no longer relevant. However, in the native execution, the difference of the throughput between "actual" and "expected" grows with regard to the number of threads, because using more threads causes more severe resource conflict inside the memory subsystem.

Given the finding above, the number of NOPs that best emulates the performance balance should increase with regard to the number of threads used. The right-side of Figure 5 is based on this idea. The graph is plotted by picking some values from the figure on the left: the value for $x = n$ in the right figure is the same as the value for $x = n$ with $(n - 2) \times 4$ NOPs inserted in the left figure. Since n moves from 1 to 8, $(n-2)$ moves from -1 to 6, but we set the number of nops to be zero when $(n - 2) = -1$. This is because the normalized performance at $x = 8$ is best emulated when the number of NOPs is 24.

Table II shows the actual values of the normalized throughput on machine 2 and the emulated environment on machine 1 (in other words, actual values of the right-hand sides of Figure 4 and Figure 5). The fact that the differences between the "Native" and "Emulated" values are small means that our system can emulate the performance balance of the target machine and the performance of a memory-access-intensive application on it. Note that deciding the number of NOPs to insert automatically is one of tasks we set for ourselves in the future work.

IV. DISCUSSION: WHAT OUR SYSTEM CAN BRING

We give two examples on how our system can outperform model-based techniques in the future.

A. Fully Leverage Existing Performance Analyzers

Profilers give fine-grained performance metrics of a given program, such as the amount of time each function uses (either exclusively or including child functions). Comparing these fine-grained metrics among machines is useful to help in understanding why the performance of the same program differs on different machines. For example in the Matrix Factorization application, the `perf` profiler shows that the `_aligned_strided_to_contig_size8` function in the BLAS library consumes constant amount of time (12 %) on machine 1, while on machine 2 the percentage grows with regard to the number of processes used. This is a strong key for the programmer to know that the different performance characterstics of the Matrix Factorization application is due to the different memory access latencies, because the function name suggests that it does some memory alignment and thus has a lot of small memory accesses.

Simulating a profiler with model-based techniques requires the programmer to combine a fine-grained model of execution time and runtime call graphs. On the other hand, we aim to leverage existing profilers as-is, by running them within our emulated environment and help in understanding the performance difference of the target program without any extra effort, such as combining multiple models and information.

B. Dealing with Multi-Threaded Applications

Modeling multi-threaded applications with locks is difficult for model-based techniques because it requires to analyze data dependencies and the probability of lock conflict. What is worse, the relative execution time of critical sections (against other parts of the program) can change when the performance balances of the underlying machines differ, because locks are shared resources among cores/sockets and checking the lock condition often requires DRAM accesses. Dealing with this issue requires queueing-theory-based analysis, whose parameters are hard to decide systematically for each target program. On the other hand, our system automatically propagates the effect of prolonged/shortened critical sections to the other parts by design because the whole of the code is actually executed.

V. RELATED WORK

Many studies tackle the performance prediction problem using performance models of the target program. Zhai *et al.* [10] predicts the total execution time of a target parallel program by combining a model of the communication pattern and the serial version of the program. Sengupta *et al.* [11] predicts the total execution time when the target program is executed with non-volatile memory, which is more energy efficient but has longer latency than DRAM. They retrieve memory traces of the target program and calculate how long these memory traces are delayed by the longer latency of Non-volatile memory by considering the number of memory accesses in each interval of the traces. Wang *et al.* [6] improves the prediction of memory IO throughput of multi-threaded programs by considering the hit-ratio of the cache inside the DRAM modules. As explained in Section I-C, the fundamental

shortcoming of model-based techniques is that they can only predict what they model. For example, to know how the difference of memory IO throughput between two machines impacts the total execution time of the target program requires the combination two models shown above, which is very costly and not straight-forward.

Quartz [12] predicts the performance of a given application when it is run on a machine equipped with non-volatile memory modules, by emulating larger read/write latencies of non-volatile memory modules compared to normal DRAMs. Although our idea is similar to Quartz, the largest difference is that we insert delay at the instruction level while they do at the epoch (time-interval) level. Inserting delay at the instruction level allows finer-grained performance analysis in the emulated environment. For example, the execution time of a function is prolonged with regard to the number of memory accesses in the fuction with our method, while in Quartz it is not the case because an epoch can include multiple function calls.

VI. Future Work

Future work includes two directions:

1) **Deciding the number of NOPs to insert systematically** is required to complete the system. It requires not only the specifications of the target machines (e.g. the number of cycles for a DRAM access, the size of the last level cache, etc.), but also some cutting-edge challenges such as estimating the resource conflict level inside the DRAM module (e.g. bus, row-buffer, etc.).
2) **New virtualization techniques** are needed to achieve the two examples in Section IV. For one thing, the overhead of system calls should be hidden because they are largely delayed in dynamic binary instrumentation and thus invalidate the profiling results of the whole program execution. Fortunately, actually decreasing the overhead of system calls in dynamic binary instrumentation is not necessary. A possible solution can be to run the program natively first, cache the results and the number of memory accesses in system calls, and then use that information for emulation.

VII. Conclusion

In this paper, we tackle performance prediction of a given program on different machines by virtually tweaking the performance balance of the underlying machine. Our system is based on the idea that inserting a small amount of NOP instructions after memory-access-related instructions can emulate the performance balance of the CPU and the memory subsystem of the target machine. We showed that our approach can emulate the normalized throughputs that an large matrix-vector multiplication workload shows in different machines, and gave examples on how our system can outperform existing model-based mechanisms in the future.

Acknowledgments

This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770 – 778.

[2] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin, "A fast parallel sgd for matrix factorization in shared memory systems," in *7th ACM Conference on Recommender Systems (RecSys)*, 2013, pp. 249–256.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[4] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *4th International Conference on Algorithmic Aspects in Information and Management (AAIM)*, 2008, pp. 337–348.

[5] "Nas parallel benchmarks," http://www.nas.nasa.gov/publications/npb.html.

[6] W. Wang, T. Dey, J. W. Davidson, and M. L. Soffa, "Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 380–391.

[7] "Pin - a dynamic binary instrumentation tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[8] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notice*, vol. 42, no. 6, pp. 89–100, Jun. 2007.

[9] S. Hajnoczi, "Towards multi-threaded device emulation in qemu," KVM-Forum 2014.

[10] J. Zhai, W. Chen, and W. Zheng, "Phantom: Predicting performance of parallel applications on large-scale parallel machines using a single node," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010, pp. 305–314.

[11] D. Sengupta, Q. Wang, H. Volos, L. Cherkasova, J. Li, G. Magalhaes, and K. Schwan, "A framework for emulating non-volatile memory systems with different performance characteristics," in *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2015, pp. 317–320.

[12] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *16th Annual Middleware Conference (Middleware)*, 2015, pp. 37–49.