# PANDA: Runtime Profiler for Near Memory Computing Scheduler

Salessawi Ferede Yitbarek, Animesh Jain, Patipan Prasertsom, Jasjit Singh

***Abstract*—The increasing gap between DRAM access speed and processor speed continues to be one of the fundamental performance bottlenecks. Due to the advent of 3D stacked chips, it is now possible to integrate simple processing units and DRAM chips in the same 3D IC and reduce the effect of this bottleneck. Due to the thermal constraints of 3D stacked systems, it is not feasible to integrate aggressive out-of-order cores into these 3D stacks. This leads us to a design where we have low power in-order cores placed in the 3D package and aggressive out-of-order cores externally connected to memory package. Workloads that require high bandwidth, low latency memory access would benefit by running on the near memory cores while compute intensive workloads are better off scheduled on an aggressive out of order core located outside of the memory package. In this work, we try to address this scheduling challenges using a runtime profiler that analyzes running programs to identify regions of an application that would benefit from near memory scheduling. We also demonstrate that the overhead of runtime profiling can be greatly reduced using a sampling scheme we implemented.**

## I. Introduction

A lot of research has been directed towards addressing performance gap between processor and memory speed. One of the methodologies that has been proposed to address this gap has been "Processing In Memory" (PIM). PIM chips integrate processor logic into memory devices which offers a new opportunity for bridging the growing gap between processor and memory speeds, especially for applications with high memory-bandwidth requirements.

The idea of integrating memory and logic in the same die has not been viable due to technology constraints. However, as 3D stacked IC manufacturing has become feasible, systems that integrate data processing logic and memory in the same 3D package are emerging.

One major challenge in 3D stacking is that it introduces strict thermal constraints. The heat produced by the processing core(s) at the bottom of the stack affects the retention capability of the DRAM layers on top of it [7]. Because of this constraint, current 3D stacking technology does not allow us to integrate high performance, power hungry cores into the package. Therefore this leads us to use low power in-order cores.

To effectively utilize these emerging systems, we need to address the challenges of workload scheduling. Workloads that require high bandwidth, low latency memory access would benefit by running on the near memory cores while compute intensive workloads are better scheduled on an aggressive out of order core located outside of the memory package.

In this work, we attempt to build a runtime system that performs online profiling of any x86 binary and maps different phases of the application to the appropriate core. In particular, the following points were explored

- We create a runtime profiler that can estimate important program characteristics such as available ILP and cache miss profile
- We demonstrate that we can reduce the overhead of runtime profiling by implementing a sampling scheme on frequently executed portions of the code
- We explore the effectiveness of the collected program features in identifying the suitable core type for each phase of application.

## II. Background and Related Work

### A. Processing In Memory and 3D Stacking

The notion of processing in memory(PIM) received a lot of attention in the late 90s and early 2000s as a viable solution to overcome the memory wall. A number of projects have explored the possibility of building smart memory systems by putting logic and memory on the same die[2][3][5]. This approach however suffered from two major limitations: (1) Logic implemented on a DRAM process was very slow. (2) Introducing logic onto the memory die reduced the storage density and increased cost per storage. Very little research was done on this area in subsequent years.

With the recent advent of commercially feasible 3D chip stacking technology, there has been renewed interest in integrating processing and memory in a single package. Recent work has shown that by deploying multiple low energy-per-instruction cores beneath a layer of DRAM chips, it is possible to get significant speedups on data-intensive workloads while operating within the thermal constraints imposed by 3D stacking[8]. The integration of such general purpose cores makes programming PIM systems approachable. The work in [8] also shows that the MapReduce programming model is amenable for programming many-core PIM systems. Similarly, the feasibility of stacking memory on top of GPGPUs has been shown[7] . Another study[11] has explored the performance gains and the energy savings from integrating application specific integrated circuits for data intensive sparse matrix and fast Fourier transform(FFT) operations.

### B. Single-ISA Heterogeneous Multicore Scheduling

Single-ISA Heterogeneous Multicore comprises of cores that are functionally equivalent but micro-architecturally dissimilar. These cores are tailored in such a way that different cores are reflective of different types of workloads. The near memory computing paradigm that we are considering leads us to a heterogeneous multicore system and also towards challenges of scheduling workloads onto these heterogeneous cores. Fine grained sampling of the performance of applications on different cores to guide scheduling decisions is infeasible due to the high overhead. Hence, schemes that predict the performance of workloads on different cores without sampling are necessary. As a result different hardware and software techniques have been proposed for predictively mapping workloads to the appropriate core type[10][9]. These techniques have demonstrated that available ILP, cache miss profile, and MLP of workloads serve as good predictors of performance on different cores.

### C. Software Based Runtime Profiling and Scheduling

Previous software scheduler implementations relied on offline profile data due to the high overhead of runtime profiling[9].

Building a runtime system that does online profiling needs to have low overhead and still be able to measure relevant program characteristics to a reasonable degree of accuracy.

## III. Profiler Design

We present our runtime profiling and analysis tool, PANDA (**P**rofiling & **A**nalysis tool for **N**ear-**D**ata **A**rchitecture), implemented in Pin[6]. PANDA profiles the application for ILP and cache misses data by instrumenting a program binary to analyze instruction dependencies and collect memory statistics. In this project, two program analysis approaches, Runtime Binary Analysis and Performance Counter Analysis, are explored. We implemented a sampling technique to improve the profiler's performance.

### A. Implementation 1: Runtime Binary Analyzer

As the specification of hardware varies from one processor to another, it is difficult to precisely estimate the ILP and cache misses of each core in the system. However, it is still possible to obtain useful information by performing architecture independent profiling and analysis based on solely on the property of the program binary. The analysis is done at the trace[1]-level granularity as it exposes a reasonable number of independent instructions. Doing so in a finer granularity (Basic Block/Instruction level) not only provide insufficient information but also incurs infeasible runtime overhead.

**ILP**: To estimate the amount of Instruction Level Parallelism of a program, PANDA scans the instructions within each instruction trace for register dependency. Based on the information, PANDA then estimates IPC in each portion of the trace and infers its representative IPC.

**Implementation** For each instruction trace that Pin fetched, PANDA analyzes all registers dependency among all the instructions and constructs a Data Dependency Graph (DDG). The constructed DDG is then divided into stages, where each stage only contains independent instructions. PANDA then analyzes each stage of execution to estimate average IPC of a trace. To keep the analysis simple, PANDA assumes infinite hardware resource and uniform instruction latencies. This assumption is proven to be sufficient for estimation purpose as demonstrated in a previous work[10]. The values of IPC calculated are the maximum possible values of IPC in each stage of the DDG, which will then be aggregated and averaged to find the approximate value of IPC for the trace. Figure 1 provides an example of such analysis.

**Cache Misses**: To estimate the number of cache misses, we set a threshold on memory reuse distances. PANDA then analyze the memory access under the assumption that if the reuse distance between a memory access and the previous access to the same memory address is greater than the said threshold, the access is considered a cache miss.

**Implementation**: PANDA instruments each memory instruction and inserts a call to a function that keeps tracks of memory reuse distance and number of cache misses in a trace. The number of cache misses is estimated by implementing an LRU scheme over a fixed-size list of memory addresses. All requests above a certain reuse threshold are considered to be misses. In this project, the threshold is chosen to be 512. As the current size of the cache can hold large number of cache lines, smaller reuse threshold will not be a good representation. Larger reuse threshold, on the other hand, will increase the size of the address list data structure to the point where it will start affecting the execution of the target program due to cache contentions and bookkeeping overheads.

[1]Trace: A block of code that has a single entry point and possibly multiple exits

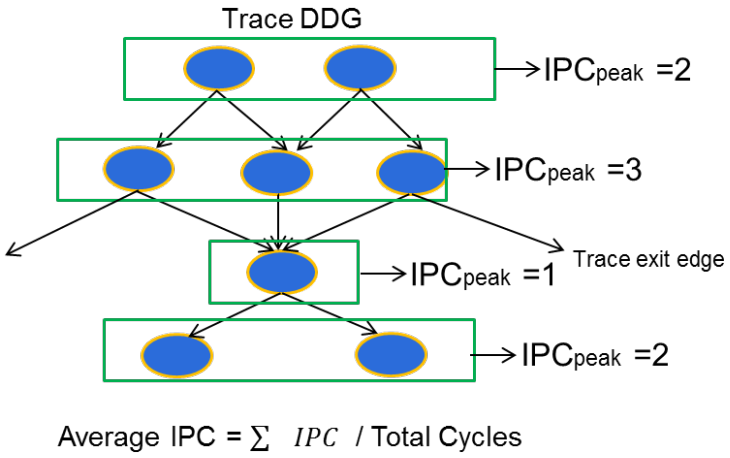


Fig. 1. **Quantifying ILP**. Runtime profiler assumes an infinitely wide processor and identifies how many instructions can be computed at a given level. This gives a rough estimate of IPC for a trace. We use IPC numbers to represent ILP.

During program execution, if a memory access trapped by PANDA refers to an addressed that is not presented in the address list, that memory address is either never accessed before or accessed with the reuse distance higher than the threshold that was set. In both cases, a memory access will be considered a cache miss.

### B. Implementation 2: Performance Counter Based Profiler

Current hardware systems provide built-in set of special-purpose registers which store the statistics of hardware related activities like dynamic instruction count, last-level cache misses, etc. Users can use these performance counters to conduct low-level performance tuning or analysis.

We use these performance counters to extract IPC numbers and last-level cache misses dynamically. The idea is to use these performance counters to capture dynamic characteristics of a code region. This information is then used in conjunction with the heuristics that are already developed offline to decide which code region is suitable for in-order core.

**Implementation**: We used perfmon library and Pin instrumentation tool to read performance counters at runtime. Similar to the approach of runtime binary analysis, we extract traces using Pintool as the target granularity for scheduling is at trace level. A lightweight approach would have been to insert the performance instrumentation at the start and exit of a trace. However, a typical trace can exit from a side path that is not instrumented. Therefore, instrumentations are done at a granularity of a basic block, and the trace information is updated by aggregating the results of basic blocks.

Specifically we monitored the following four hardware events: i) **LLC MISSES** represent the number of DRAM requests made by the processor. If the number of memory requests made by the processor is high, then a high-bandwidth memory system might benefit that code region. ii) **Instructions** represent the dynamic instruction count for a trace. iii) **cycle count** represent the dynamic cycle count for a trace. We can use Instructions and cycle count hardware event to calculate IPC for a trace. iv) **L1D MISSES** represents the L1 Data cache misses. We used this to perform a sanity check with the sniper data.

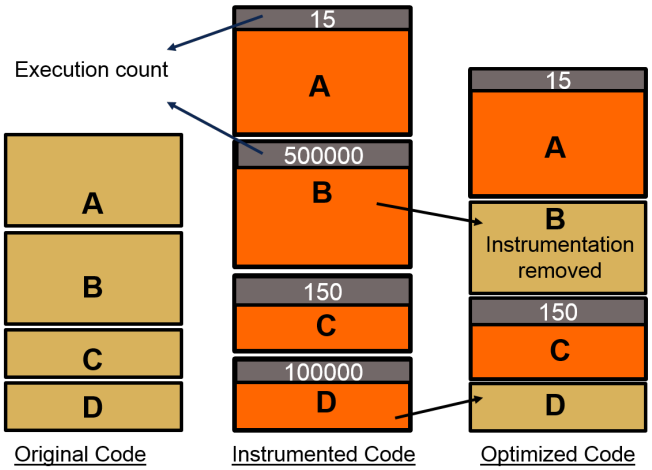


Fig. 2. **Reducing instrumentation overhead by sampling**.

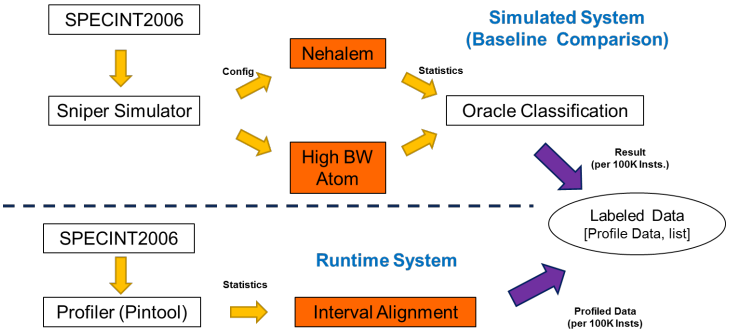


Fig. 3. **Heuristic development process.**

### C. Sampling

One of the challenges in deploying dynamic binary instrumentation system is the runtime overhead it incurs. The techniques we proposed above results in an overhead that is greater than 200x. To mitigate this problem, we implement a sampling technique. The key idea is that it is enough to measure execution characteristics of frequently executed traces for sufficient numbers of iterations. Miss rates and other measured values will start to stabilize as the program progresses, eliminating the need to continuously monitor frequently executed sections of the code.

To realize this sampling technique, a counter is inserted at the entry point of each trace in order to keep track of its execution frequency (Figure 2 middle). After a given trace has executed a sufficient number of times, we remove all instrumentations from it (Figure 2 right). After the removal process, the de-instrumented traces will run at a near native speed. This ensures that the frequently executed portions of the code do not experience significant overhead. This optimization step is not applied to infrequently executed portions of the code as they do not noticeably degrade the application performance.

## IV. Offline Heuristic Development

Figure 3 provides an overview of the heuristic evaluation process. In order to determine code regions that will be able to take advantage of near memory computation, we evaluate all SPECint benchmarks using the Sniper architectural simulator[1] using two hardware configurations: Gainestown (Intel's Nehalem Architecture) and a high-bandwidth memory interface connected to a Silvermont (Intel's Atom Architecture). To reduce the simulation time, we use SimPoint[4] to find representative portions of each application and only consider them for our analysis.

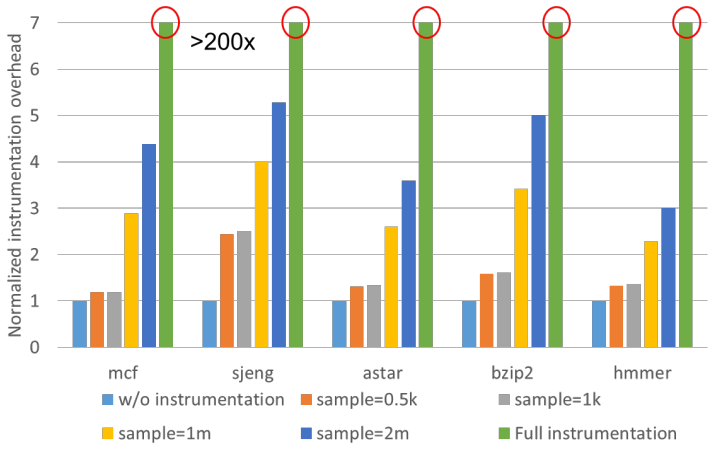


Fig. 4. **Instrumentation Overhead for Runtime Binary Profiler.** We observe that for a sample size of 1 million, the instrumentation overhead is reduced to 4-5x.

Based on the collected statistics, we tag every 100,000 instruction execution interval with the most suitable core type. Similarly, we tag every 100,000 instruction execution interval with profile information generated by our tools.

To discover the effective scheduling heuristics, we combine the profile data and the labels from the architectural simulator.

In general, a big out-of-order core provide a higher performance than a small in-order core at a cost of higher power consumption. Hence, in the core selection process, we address this point by setting an "acceptable" slowdown level for a small core. If the small core slow down for an interval is lower than a threshold (in this case, it is set to 1.75), then we consider that it can be tolerated and running the interval on a small core will be more beneficial as it will provide energy benefit. If the slowdown is larger than a threshold, we consider the interval to be better suited running on a big core.

## V. Experimental Results

In this section we present: i) Sampling accuracy and overhead results ii) Heuristic evaluation. As stated earlier, we target heterogeneous system with an aggressive out-of-order core and an in-order core coupled with a high-bandwidth memory. We use Sniper simulator [1] for offline heuristics discovery. For runtime profiler, we analyze the two profilers described in Section III.

**Sampling Accuracy and Overhead** We now discuss accuracy and instrumentation overhead associated with the two profiling techniques we proposed.

**Runtime Binary Analyzer**. As stated earlier, we need a fine-tuned sampling size in order to capture a stable runtime characteristic of a trace without generating high performance overhead. In Figure 4, we show how sampling size affects the accuracy of profiling. Increasing the sample size from 1 thousand to 1 million results in an average miss rate deviation of 16%. However, we observe that increasing the sample size from 1 million to 2 million reduce the average miss rate deviation down to 5%. This shows that runtime characteristics start to stabilize when the sample size is over a million.

We compare runtime overhead of several SPEC benchmarks for different sample sizes. Figure 4 shows that runtime overhead for full instrumentation without any sampling is exorbitantly high (>200x).

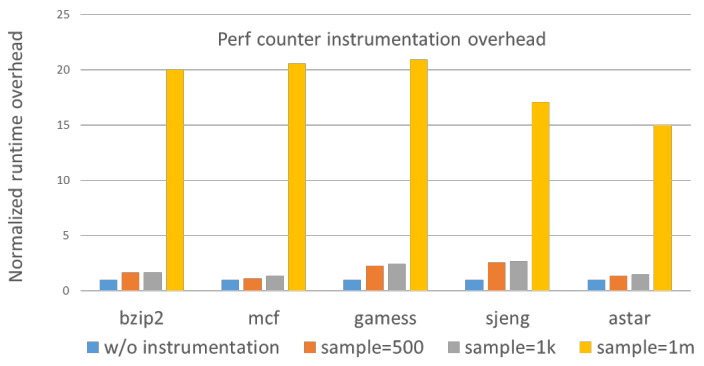


Fig. 5. **Instrumentation Overhead for Performance Counter Profiler.** We observe that performance counter profiler shows exorbitantly high performance overhead making it infeasible for deployment.

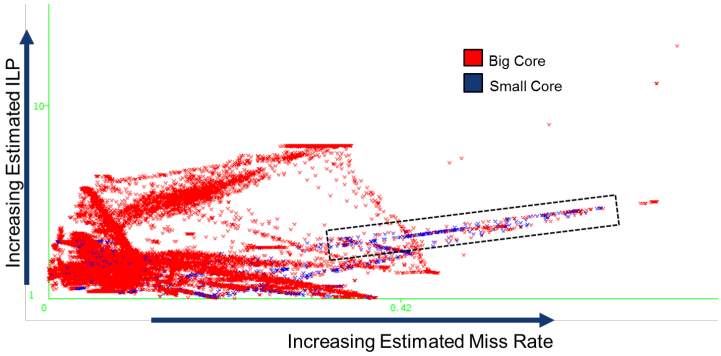


Fig. 6. **Labeled data from runtime binary analysis**. TThe dashed box represent a heuristic which schedules every code region with the ILP and miss rate numbers lying in that region on small core.

We also observe that the runtime overhead for a sample size of 1 million is reduced to 4-5x. Therefore, we conclude that a sample size of 1 million is a good enough sampling size with an acceptable performance overhead.

**Performance counter based profiler**: Figure 5 shows the runtime overhead for this technique. This kind of instrumentation at basic block level results in huge performance overhead. The figure shows that even for a sample size of 1 million, the overhead can be above 20x. We were unable to complete a few runs with sample size of 2 million due to timing constraints.

In a nutshell, the runtime binary analyzer with a sample size of 1 million is the most promising configuration.

**Heuristic evaluation** We use the oracle data obtained from Sniper tool as explained in the previous section and correlate it with the runtime binary profiler data. We plot the runtime characteristics obtained from the profiler as shown in Figure 6 (around 90,000 execution intervals where each code region is 100,000 instructions). Each execution interval is labeled with the most suitable core. The figure shows an interesting trend. We observe that if we schedule the execution intervals which lie in the rectangular dashed box on a small core, then in most cases it would be in congruence with oracle data prediction. This region has a high amount of memory requests but not a very high ILP. This characteristic makes it suitable to run it on an in-order core coupled with a high-bandwidth memory. A deeper analysis needs to be done to quantify performance gains and classification accuracy. This has not been done in this project due to time constraints.

## VI. Conclusion and Future Work

In this project we compared two viable implementations of a runtime profiler system - one that analyzes data dependence and address reuse distances and another one that relies on hardware performance counters. Our PMU based implementation has inherent inefficiencies and as a result the overhead on the original binary was quite high. In addition, the PMU data was affected by the extra instrumentation code and did not yield any promising results. On the other hand, our binary analysis profiler coupled with a sampling technique exhibited much lower overhead and higher accuracy. We expect to get improved performance by using binary instrumentation tools with lower runtime overhead(e.g. DynamoRIO). Our preliminary results show that program signature data collected through our binary analysis tools follow a pattern that would allow classification with a simple thresholding scheme.

However, better analysis of classification errors and deeper analysis using multi-threaded, data centric workloads is needed to fully evaluate the merits of our system.

## References

[1] T.E. Carlson, W. Heirman, and L. Eeckhout. “Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation”. In: *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. 2011, pp. 1–12.

[2] Jeff Draper et al. “The Architecture of the DIVA Processing-In-Memory Chip”. In: *International Conference on Supercomputing(ICS’02)* (2002).

[3] Joseph Gebis, Sam Williams, and Christos Kozyrakis. “VIRAM1 : A Media-Oriented Vector Processor with Embedded DRAM”. In: ().

[4] Greg Hamerly et al. “Simpoint 3.0: Faster and more flexible program analysis”. In: *Journal of Instruction Level Parallelism*. 2005.

[5] Yi Kang et al. “FlexRAM: Toward an advanced intelligent memory system”. In: *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. IEEE. 2012, pp. 5–14.

[6] Chi-Keung Luk et al. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: PLDI ’05. 2005.

[7] Dong Ping et al. “TOP-PIM: throughput-oriented programmable processing in memory”. In: *International Symposium on High-Performance Parallel and Distributed Computing(HPDC14)* (2014).

[8] Seth H Pugsley et al. “NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads”. In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2014).

[9] Daniel Shelepov et al. “HASS: a scheduler for heterogeneous multicore systems”. In: *ACM SIGOPS Operating Systems Review* (2009).

[10] Kenzo Van Craeynest et al. “Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE)”. In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA ’12. 2012.

[11] Qiuling Zhu et al. “A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing”. In: *2013 IEEE International 3D Systems Integration Conference (3DIC)* (2013).