Pinpointing Performance Inefficiencies via Lightweight Variance Profiling

Pengfei Su
College of William and Mary
Williamsburg, Virginia, USA
psu@email.wm.edu

Shuyin Jiao
College of William and Mary
Williamsburg, Virginia, USA
sjiao@wm.edu

Milind Chabbi
Scalable Machines Research
milind@ScalableMachines.org

Xu Liu
College of William and Mary
Williamsburg, Virginia, USA
xl10@cs.wm.edu

ABSTRACT

Execution variance among different invocation instances of the same procedure is often an indicator of performance losses. On the one hand, instrumentation-based tools can insert calipers around procedures and identify execution variance; however, they can introduce high overheads. On the other hand, sampling-based tools insert no instrumentation and have low overheads; however, they cannot synchronize samples with procedure entry and exit.

In this paper, we propose FVSampler, a lightweight, sampling-based variance profiler. FVSampler employs hardware performance monitoring units in conjunction with hardware debug registers to sample and monitor whole procedure instances (invocation till return) and collect hardware metrics in each sampled procedure instance. FVSampler, typically, incurs only 6% runtime overhead and negligible memory overhead making it suitable for HPC-scale production codes. We evaluate FVSampler with several parallel applications and demonstrate its effectiveness in pinpointing execution variance. Guided by FVSampler, we tune data structures and algorithms to obtain significant speedups.

CCS CONCEPTS

• **General and reference** → **Measurement**; **Metrics**; **Performance**.

KEYWORDS

Variance profiling, lightweight measurement, PMU, debug register

ACM Reference Format:
Pengfei Su, Shuyin Jiao, Milind Chabbi, and Xu Liu. 2019. Pinpointing Performance Inefficiencies via Lightweight Variance Profiling. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3295500.3356167

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356167

1 INTRODUCTION

High-performance computing (HPC) software has become increasingly complex; it includes large amounts of source code, sophisticated control and data flow, a hierarchy of component libraries, and growing levels of abstractions. Application developers can easily introduce performance inefficiencies embedded deep in the code bases that are difficult to identify. Such performance inefficiencies prevent software from enjoying the full system capacity. Performance tools are necessary to pinpoint performance bottlenecks and guide code optimization.

Application developers primarily think of code in terms of functions (aka procedures) which form mental boundaries of functionality. It is natural that when developers investigate performance problems, they often want to see the execution metrics at function level granularity. Almost all performance tools facilitate function level attribution; in fact, most tools offer finer-grained attribution such as loops or statements with call path attribution. A recent line of work has investigated procedure instance level variance as a major cause for performance problems such as long tail latency [16–18, 20, 29] particularly in the enterprise cloud systems. This paper targets the procedure instance level execution variance in the HPC domain. Variance is a concern in HPC domain as well, as we show with a motivating example in Section 1.1 and several case studies in our evaluation section.

A prerequisite of profiling for variance among procedure instances is the ability to place monitoring calipers around procedure entry and exit. This allows comparing the metrics from two or more instances of invocation of the same procedure within the same execution. Instrumentation-based tools [6, 14, 19, 35, 40, 41, 48] avail themselves to procedure instance level metrics because the instrumentation can be placed at the entry and exit of a procedure; in fact, even finer-grained placement such as statements or instructions is also possible. They can count resources consumed by any invocation instance of the same code region albeit the overhead can be non-trivial (~2×).

Sampling-based tools [1, 9, 11, 15, 31, 48, 59], on the other hand, use interrupt-based mechanism supported by hardware performance monitoring units (PMU) or operating system (OS) timers, attribute samples to code regions, and highlight hotspots based on the number of samples taken in the same code region. Expecting a PMU sample to be delivered precisely at the entry of a procedure

instance and the immediate next sample to be delivered precisely at the exit of that procedure instance is wishful thinking but impractical for almost any PMU or timer-based sampling tool. Each sample is a point in time, one sample cannot be compared with another sample quantitatively. Hence, identifying execution variance of the same procedure across different invocation instances is seemingly impossible[1]. There is little variance between two samples since each one is delivered after the same number of preconfigured events. Furthermore, variance among procedure instances is not statistically significant in a sampling-based profiler if the sampling interval is larger than the execution time of the procedure itself. *In summary, sampling-based tools, until now, have not been able to synchronize samples with procedure boundaries.*

Measuring the variance across procedure invocation instances requires starting and stopping measurements at procedure entry and exit. This act of starting and stopping measurements at *every* procedure entry and exit is equivalent to placing instrumentation, which defeats the purpose of lightweight sampling. Hence, there is a dilemma, how can we enjoy the low overhead of sampling and yet collect meaningful measurements at procedure instance boundaries so that we can compare execution variance across two or more invocations of the same procedure? We would like to emphasize that we are interested in the variance of two or more execution instances of the same procedure in a single execution[2]. However, we would like to collect such variance for a large number of procedures that the program executes and we would like to do so in a single profiling session.

We address the aforementioned problem in FVSampler, a lightweight sampling-based variance profiler with the ability to show procedure instance level execution variance. FVSampler employs PMUs to sample function call (entry) and then uses debugs registers to intercept the return (exit) from the same function invocation and measures metrics between these two points. The metrics can be any of the supported PMU events, e.g., CPU cycles, cache misses, energy consumption, to name a few. The key differentiating aspect of FVSampler when compared to a large class of profilers is its ability to intercept function call and return with no instrumentation (source or binary) and prior knowledge of the program, which makes it useful in production. A thorough evaluation on several parallel applications shows that quantifying variance on per sampled function invocation offers new avenues into understanding performance losses; mitigating the causes of variance enhances performance.

In the rest of this section, we first describe a motivating example, showing that identifying execution variance in HPC code bases yields unique optimization opportunities. We then summarize the contribution of this paper and overview the paper organization.

[1]One may be able to approximately infer procedure boundaries by looking at consecutive samples taken in the same procedure, however, this method is inaccurate for a small procedure called in a loop when consecutive samples across multiple invocations of the same procedure are not interleaved by a sample in another procedure.

[2]Comparing total samples taken by two different procedures or procedures from two different threads or processes is straightforward and available in almost all HPC profilers, sampling or otherwise.

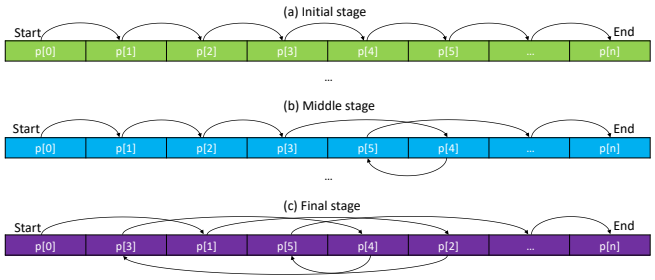


Figure 1: Access order and storage order of an array of particles (p[]) in GTC. (a) At the program start, particles are stored in cell order, which exactly matches access order. (b) and (c) As the execution progresses, particles move from one cell to another, resulting in the mismatch between access order and storage order.

1.1 Motivation

NERSC-8 GTC [39], a particle-in-cell code, is used for Gyrokinetic Particle Simulation of Turbulent Transport in Burning Plasmas. A previous study [36] on GTC shows that cache misses in different invocation instances of the procedure that accesses an array of particles in sequential order varies significantly and increases as the execution progresses. With the source code analysis, we notice that at the program start, all particles are stored in cell order, which exactly matches access order, as shown in Figure 1a. However, as the program continues, particles move from one cell to another, resulting in the mismatch between access order and storage order (loss of data locality), as shown in Figure 1b and 1c. Periodically sorting particles in cell order can avoid the loss of data locality and improve the program performance by more than 20%. However, no existing sampling-based tools can identify procedure instance execution variance since they cannot distinguish whether two samples from the same procedure belong to the same instance of that procedure. As a result, they offer little help in optimizing this problematic procedure in GTC. Instrumentation-based tools can show procedure instance execution variance by instrumenting function call and return albeit the overheads are quite large. For example, when we employ Intel Pin [35] to capture each procedure instance in GTC by instrumenting call and return instructions, a 5× runtime overhead is introduced and even worse an 8× runtime overhead is introduced with call path collection enabled. Compile-time instrumentation can result in lower overhead but will not be able to instrument the library code.

1.2 Contribution Summary

We make the following contributions:

- We develop a technique to overcome a critical missing piece in sampling-based tools — synchronize samples with procedure boundaries to monitor whole procedure instances.
- We develop a lightweight sampling-based variance profiler — FVSampler — that combines PMUs and debug registers available in commodity CPU processors to quantify variance across different invocations of the same function without requiring code instrumentation.

- We address the challenges raising due to combining the usage of PMUs and the limited number of debug registers.
- We show that FVSampler monitors fully optimized, unmodified binary executables and provides rich information to guide code optimization, such as calling contexts, variance metrics and their distributions, and source code attribution.
- We demonstrate the effectiveness of FVSampler by optimizing several parallel applications under the guidance of FVSampler, yielding significant speedups.

1.3 Paper Organization

The rest of this paper is organized as follows. Section 2 reviews the related work and distinguishes FVSampler. Section 3 offers the background knowledge necessary to understand FVSampler. Section 4 highlights the methodology we employ to capture function entry and exit. Sections 5 and 6 depict the design and implementation of FVSampler. Section 7 evaluates FVSampler's overhead and shows several case studies. Section 8 presents our conclusions and future work.

2 RELATED WORK

Tracing Tools. HPCToolkit [1], perf [31], gprof [15], CrayPAT [11], and Intel VTune [9], Oracle Solaris Studio [42], OpenSpeedShop [47], and PGPROF [54] use interrupt-based sampling techniques supported by PMUs or OS timers to sample performance events and present them in chronological order. Unlike FVSampler, these tools do not capture function entry and exit and do not pinpoint function-level variance. Intel Pin [35], TAU [48], Scalasca [14], DynamoRio [6], Valgrind [40], and Dyninst [41] show procedure level execution variance via exhaustive or selective code instrumentation. Compared to exhaustive instrumentation, FVSampler incurs much lower overhead in both runtime and memory; compared to selective instrumentation, which needs to know the interesting functions for study, FVSampler does not require any prior knowledge of the program.

Variance Diagnosis Tools. X-Ray [4] pinpoints performance inefficiencies by employing dynamic binary instrumentation to identify basic block level performance variance. Spectroscope [46] diagnoses performance changes in distributed systems by comparing request flows between two time periods (the period before the change and the period after the change). Yoon *et al.* [60] combine outlier detection and causality analysis to detect performance anomalies on individual transactions in online transaction processing systems. VarianceFinder [45] identifies the performance variance of requests under the same call path. Unlike these approaches, FVSampler focuses on identifying function-level variance.

Szebenyi *et al.* [51] use instrumentation to intercept MPI routines and use sampling to profile the remaining code during program execution. Unlike it, FVSampler only uses sampling to profile function invocation instances and does not distinguish libraries from the main executable. Any function called via a call instruction is a potential candidate to be monitored.

VProfiler [19], an instrumentation-based tool, also identifies function-level variance. However, users have to manually annotate code regions of interest before applying VProfiler to the target program. Moreover, VProfiler only identifies latency variance. In contrast, FVSampler is able to identify variance of any PMU event, such as CPU cycles and cache misses.

To the best of our knowledge, FVSampler is the first non-intrusive sampling-based tool to study function-level variance of HPC workloads.

Hardware Debug Register-assisted Tools. A few tools use hardware debug registers to pinpoint performance inefficiencies and correctness issues. None of them is able to quantify the execution variance. Erickson *et al.* [13] employ debug registers [22, 37] to detect data races in the Windows kernel. Jiang *et al.* [21] extend it to the Linux kernel. They sample memory accesses and set watchpoints at the sampled effective addresses to detect conflicting accesses. They use code breakpoints to intercept random instructions and use them to monitor memory accesses for a time window. Liu *et al.* develop DoubleTake [33] and CSOD [32], which use debug registers to identify memory-related vulnerabilities, such as buffer overflows, use after free, and memory leaks. Pesterev *et al.* develop DProf [43], which combines PMU and debug registers to construct the data flow across runtime objects. DProf suffers from the limited number of debug registers; it needs to run a program multiple times to achieve higher coverage. Wen *et al.* [57] combine PMUs and debug registers to identify wasteful memory operations in native languages. Su *et al.* [50] employ the similar techniques to identify wasteful memory operations in managed languages. Chabbi *et al.* [7] apply PMUs and debug registers to identify false sharing in multi-thread or multi-process executions. Wang *et al.* [55] apply PMUs and debug registers to measure reuse distances to quantify whole-program data locality.

Orthogonal to these tools, FVSampler addresses a different problem with a different usage of debug registers.

Software-based Return Address Interpretation. Kasikci *et al.* [24] trace cold code by dynamically rewriting the first instruction of every basic block with the int 3 breakpoint instruction, which causes a trap. This approach can be used to rewrite all the return instructions to capture function exits. However, such binary rewriting does not offer per-thread breakpoints. Maintaining local breakpoints with code caches can incur high overhead. Arnold and Sweeney [3] perform call stack unwinding by replacing the function return address with a trampoline (the address of a handcrafted code snippet). When the modified function returns, the control is first transferred to the trampoline and then transferred back to the program. This software approach is also able to intercept the return from the same function invocation. Unlike these approaches, FVSampler uses hardware debug registers to intercept the return from the same function invocation and targets a completely different problem — variance profiling.

3 BACKGROUND

Hardware Performance Monitoring Unit (PMU). Modern CPUs expose programmable registers (aka PMU) that count various hardware events such as retired instructions, CPU cycles, and cache misses, to name a few. These registers can be configured in sampling mode: when a threshold number of hardware events elapse, PMUs trigger an overflow interrupt. A profiler is able to capture the interrupt as a signal, known as a sample, and attribute the metrics

collected along with the sample to the execution context. These registers can also be configured in counting mode: users can read the number of occurrences of hardware events from PMUs at any time during program execution. PMUs are per CPU core and virtualized by the OS for each thread.

Intel offers Precise Event-Based Sampling (PEBS) [8] in Sandy-Bridge and following generations. PEBS can capture the precise instruction pointer (IP) for the instruction resulting in event counter overflow. AMD Instruction-Based Sampling (IBS) [12] and PowerPC Marked Events (MRK) [49] offer similar capabilities.

Hardware Debug Register. Hardware debug registers [22, 37] enable trapping the CPU execution for debugging when the program counter (PC) reaches an address (breakpoint) or an instruction accesses a designated address (watchpoint). One can program debug registers to trap on various conditions: accessing addresses, accessing widths, and accessing types (write-only and read-or-write). The number of debug registers is limited; an x86 processor has four debug registers and a PowerPC processor has one debug register.

Linux perf_events. Linux offers a standard interface to program and sample PMUs and monitor debug registers via the `perf_event_open` system call [30] as well as the associated `ioctl` system calls. The Linux kernel can deliver a signal to the specific thread whose PMU event counter overflows or debug register traps. A PMU sample is a CPU interrupt caused when an event counter overflows. A watchpoint exception (aka trigger) is a synchronous CPU trap caused when a monitored address is accessed. Both PMU samples and watchpoint exceptions are handled via Linux signals. The user code can extract PMU data and execution contexts at the signal handler.

4 METHODOLOGY

The PMU provides precise events to sample call and return instructions, however, that is not sufficient — PMU samples cannot be configured to deliver one sample at the function entry and another at the return from the same function instance. Our solution is to use PMUs to sample only the call instructions and use debug registers to intercept the returns from the function matching the sampled call instructions.

The point where the PMU delivers an interrupt is at the function entry, that is, right after the call instruction execution in the caller. At this point, the stack pointer (register `rsp` in x86) points to the top of the stack (M[`rsp`]), which holds the return address for the caller to continue (Figure 2a). The callee accesses this return address stored on the stack just when it is about to return. We can intercept the return from the callee by protecting the access to this memory location (M[`rsp`]). We use debug registers to protect the subsequent access to M[`rsp`]. When the callee fetches the return address from M[`rsp`], it triggers a read-or-write watchpoint trap, as shown in Figure 2b. Furthermore, the signal handlers invoked during these two points (PMU sample at a call and watchpoint trap at the return) allow us to record the metrics of interest and the difference in metrics between these two points can be attributed to the function invocation instance. In summary, we can now synchronize sampling with start and end points of functions and since we rely on PMU samples, we have not introduced any source or binary

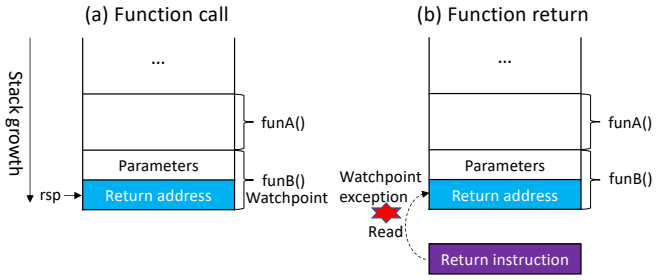


Figure 2: Actions on function call and return. (a) The call instruction in `funA()` (the caller) pushes the parameters of `funB()` (the callee) and the return address on the stack; After the call instruction execution, the return address is on the top of the stack. We set a watchpoint at the stack location (marked in blue) that holds the return address. (b) The return instruction in `funB()` fetches the return address from the stack, which triggers a read-or-write watchpoint trap.

instrumentation; statistically significant functions (i.e., functions with a high invocation frequency) appear in our samples with a high probability.

Before arriving at the final design, we explored two other strategies in capturing function exits. These two approaches used debug registers as breakpoints (trapping on instruction execution) instead of watchpoints (trapping on memory access). In the first approch, we used debug registers to directly monitor *return instructions*, e.g., `retq` in the body of the callee; the return instructions were obtained via an on-the-fly binary analysis. However, it is common that a function has many return instructions but this approach could only monitor four return instructions in a function body with the four available debug registers. This approach would fail to capture the exit from a function if the unmonitored return instruction is executed. In the second approach, we used debug registers to monitor the *return address* — the address of the instruction in the caller that is executed right after the callee returns. This approach, however, failed for recursive functions because different invocation instances of a recursively called function all share the same return address. Consequently, we arrived at the final, correct approach of monitoring the stack location holding the return address of a function invocation.

5 DESIGN AND IMPLEMENTATION

Figure 3 shows the implementation details of how FVSampler uses PMUs to sample function call and uses debug registers to intercept the return from the same function instance. ① FVSampler subscribes to the precise PMU call event in sampling mode and configures debug registers as watchpoints for each thread via the `perf_event` interface. FVSampler also configures other PMUs in counting mode to monitor user-specified events (e.g., CPU cycles, cache misses) as for the variance metrics. ② When the PMU counter overflows on sampling function calls, it triggers an interrupt. FVSampler handles the interrupt signal, constructs the calling context at the interrupt via unwinding the execution call stack, and reads the user-specified PMU counters to obtain their current values (V_{call}). ③ FVSampler obtains the stack address M[`rsp`] recorded

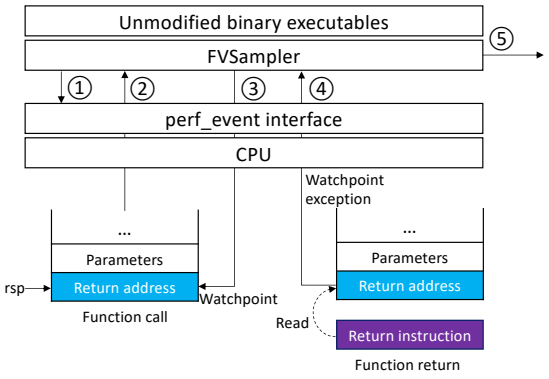


Figure 3: FVSampler's actions in steps to collect variance metrics. FVSampler works on unmodified binary executables and off-the-shelf Linux kernel in commodity CPU architectures.

in register `rsp`, sets a read-or-write watchpoint [3] at M[`rsp`], and resumes the program execution. ④ When the return instruction reads the return address from M[`rsp`] [4], it triggers a watchpoint trap. FVSampler handles the trap signal and reads the user-specified PMU counters to obtain their current values (V_{ret}). ⑤ FVSampler records the difference between V_{ret} and V_{call}, which is the count of the performance events occurring in the current function instance. FVSampler disarms the watchpoint and resumes the program execution until the next PMU overflow. When the signal handler code is executing, we stop all PMU counters so that FVSampler's overhead is not counted towards the metrics collected for the function under investigation.

This scheme assumes flat function calls — we capture each function instance's call and return before monitoring the next one. We need to handle the code with deep call chains.

5.1 Addressing Deep Call Chains

Hardware offers only a small number of debug registers, which becomes a limitation if the PMU delivers a new sample before none of the previously set watchpoints traps. To better illustrate the problem, consider a call chain consisting of three functions: `main()`→`funA()`→`funB()`. Assume the PMU is able to sample both `funA()` and `funB()`, and there is only one debug register available. The first sample occurs when `funA()` is being called by `main()`, which results in setting a watchpoint at the stack address holding the return address of `funA()`. The second sample occurs when `funA()` is calling `funB()`. However, there is no room to monitor the stack address holding the return address of `funB()` since the previously set watchpoint is still active. With this strategy, in a system with N debug registers, at most N function instances can be monitored simultaneously.

FVSampler addresses this problem based on an observation: callees always return before their callers return [5]. Thus, FVSampler maintains a stack $\mathcal{S}$ to save active stack addresses being monitored by watchpoints. We use the same call chain as an example to illustrate our idea, as shown in Figure 4. Upon the sample that captures the call instruction to `funA()`, FVSampler sets a watchpoint at the stack address holding the return address of `funA()` since a debug register is available, as shown in Figure 4a. Upon the next sample that captures the call instruction to `funB()`, FVSampler disarms the watchpoint, pushes the address that the watchpoint is monitoring for `funA()` on $\mathcal{S}$, and reconfigures the debug register to monitor the stack address holding the return address of `funB()`, as shown in Figure 4b. When `funB()` returns later in the execution, it triggers a watchpoint trap. FVSampler handles the trap as normal for variance metrics, disarms the watchpoint, pops out the stack address holding the return address of `funA()` from $\mathcal{S}$, and reconfigures the watchpoint to monitor the stack address holding the return address of `funA()`, as shown in Figure 4c.

With this scheme, only one debug register is needed to handle the deep call chain, which makes our technique widely applicable to both x86 and PowerPC architectures.

5.2 Associating with Calling Contexts

Simply attributing a sample to the corresponding function does not provide insights needed for developer actions. For example, attributing a sample to a common `glibc` function such as `malloc()`, offers little insight since it can be invoked from many places in a complex application. A detailed attribution demands providing the full calling context: `main():line#`→`funA():line#`→`...`→`malloc():line#`. Thus, FVSampler obtains the calling context where a function call occurs. Since the interrupt happens immediately after the function call, the calling context of the interrupt is at the function entry. At the function return, FVSampler need not determine the calling context as it is the same as the one obtained at the function call. FVSampler constructs calling contexts with an on-the-fly binary analysis technique [53], which efficiently maintains calling contexts as a compact calling context tree [2] by merging common prefixes.

5.3 Obtaining Variance Metrics

FVSampler provides two options to present variance metrics. One is to plot the metrics collected from all the sampled instances for a given function in a given calling context. This plot provides the most straightforward view of variance and is able to expose variance patterns (e.g., the increase of cache misses in GTC described in Section 1.1) for optimization actions. The other is to provide a compact view, which computes the mean, standard deviation, and coefficient of variation across the metrics collected from the sampled instances for any function in any calling context. Such compact view can help users quickly locate the problematic functions for further investigation. To avoid recording metrics of every sampled function instance, we leverage Welford's online algorithm [58]. In this section, we briefly describe this algorithm; details about the rigorous proofs can be found in the related paper [56].

When the i^{th} sample of a function occurs, the mean ($\overline{V}_{\{1,\ldots,i\}}$), standard deviation ($SD_{V_{\{1,\ldots,i\}}}$), and coefficient of variation

[3] x86 debug registers do not offer the trap-only-on-read facility.
[4] In a function's execution, only return instructions read the return address from M[`rsp`] and no instructions write values to M[`rsp`]. We do not consider buffer overflows in the security domain.
[5] `longjmp()` is an exception, which is discussed in Section 5.4.

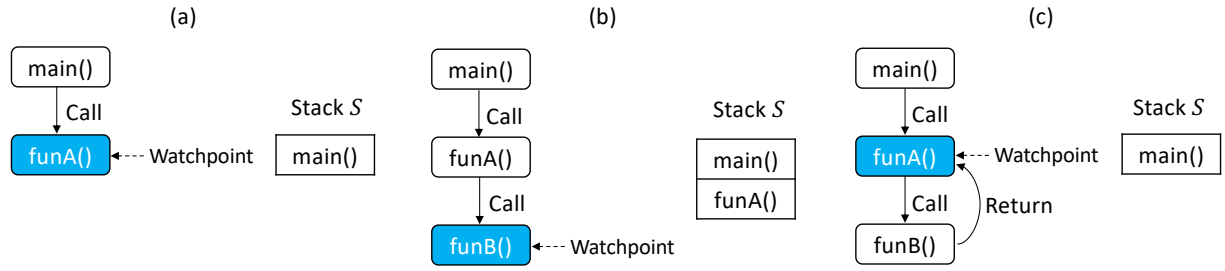


Figure 4: Using one debug register to monitor all sampled function instances. FVSampler maintains a stack $\mathcal{S}$ to save active stack addresses being monitored by watchpoints. (a) When `main()` is calling `funA()`, FVSampler sets a watchpoint at the stack address holding the return address of `funA()`. (b) When `funA()` is calling `funB()`, FVSampler pushes the address the watchpoint is monitoring for `funA()` on $\mathcal{S}$, disarms the watchpoint, and sets it at the stack address holding the return address of `funB()`. (c) When `funB()` is returning to `funA()`, FVSampler pops out the address holding the return address of `funA()` from $\mathcal{S}$ and resets the watchpoint at it.

($CV_{V_{\{1,\ldots,i\}}}$) of the variance metric ($V_{\{1,\ldots,i\}}$) across the first i samples are calculated by the following equations:

$$\overline{V}_{\{1,\ldots,i\}} = \frac{(i-1)\overline{V}_{\{1,\ldots,i-1\}} + V_i}{i}$$

$$SD_{V_{\{1,\ldots,i\}}} = \sqrt{\frac{(V_i - \overline{V}_{\{1,\ldots,i-1\}})(V_i - \overline{V}_{\{1,\ldots,i\}}) + (i-2)SD^2_{V_{\{1,\ldots,i-1\}}}}{i-1}}$$

$$CV_{V_{\{1,\ldots,i\}}} = \frac{SD_{V_{\{1,\ldots,i\}}}}{\overline{V}_{\{1,\ldots,i\}}}$$

From these equations, we can see that computation on these metrics enjoys an incremental fashion, with no need to record all samples. In this paper, we employ the coefficient of variation metric to quantify procedure instance execution variance.

5.4 Discussions

Handling Parallelism. FVSampler works for MPI programs as it monitors each MPI process independently. FVSampler also works for multithreaded programs since PMUs and debug registers are virtualized by the OS for each thread. FVSampler does not handle the user-level threading where a function call and its corresponding return are executed on two different OS threads. A solution to user-level threading would require minimal support from the runtime — the user-level thread switching should save and restore the debug register state.

***Handling* `longjmp()`.** `setjmp()`/`longjmp()` provide inter-procedure jumps, which deviates from the typical calling conventions. FVSampler intercepts them by overloading their calls. When `longjmp()` executes, FVSampler disarms the active watchpoint. FVSampler also clears the watchpoint stack $\mathcal{S}$ because we do not know which stack frame `longjmp()` jumps to.

Understanding the limitation of sampling. Like any sampling-based tool, FVSampler captures statistically significant functions (i.e., functions with high invocation frequency) and misses some insignificant ones. It satisfies the needs for studying variance because variance is meaningful only on functions with high invocation frequency. Seldom called functions (e.g., `main()`) are less interesting. FVSampler will miss some functions that are not invoked via a call instruction, e.g., functions that are inlined or called via a tail call.

6 USAGE OF FVSAMPLER

6.1 FVSampler's Workflow

FVSampler consists of three components: a runtime profiler, a post-mortem analyzer, and a GUI. The runtime profiler accepts fully optimized binary executables and collects profiles, which has been described in Section 5. The post-mortem analyzer and GUI analyze the runtime profiles and associate them with the program source code for intuitive guidance. The rest of this section focuses our discussion on the post-mortem analyzer and GUI.

Post-mortem Analyzer. As the runtime profiler produces per-thread profiles, the analyzer needs to coalesce the profiles for the entire execution. The coalescing procedure follows the rule: two invocation instances of the same function from different threads are merged *iff* they have the same calling context. All the metrics are also merged across threads. The scheme is similar for profiles from different processes. The calling context profiles can scale the analysis of program execution to a large number of cores. The coalescing overhead grows linearly with the number of threads and processes run in the monitored program. FVSampler leverages the reduction tree technique [52] to parallelize the merging process. FVSampler takes less than one minute to merge per-thread profiles in all of our studied programs.

GUI. The GUI is built atop an existing graphical interface, `hpcviewer` [38], which enables navigating calling contexts and the corresponding source code ordered by the monitored metrics in a top-down and bottom-up manner. Additionally, the GUI provides an option to plot the execution variance of any sampled function in the timeline. Figure 5 in Section 7.1 shows an example of the GUI and we defer the explanation of the GUI details to that section.

6.2 FVSampler's Optimization Guidance

Our optimization decision on a function is based on the execution time and variance (i.e., coefficient of variation metrics), as shown in Table 1. Only functions with both high execution time and variance are worth efforts for further performance analysis. In all of our case studies, we investigate a function *iff* it accounts for more than 10% CPU cycles over the entire program and has larger than 20% *intra-thread variance* or 10% inter-thread variance. Once

Table 1: Optimization decisions based on the execution time and variance. We would like to focus our optimization efforts on functions with both high execution time and variance.

Execution time	Variance	Guidance
High	High	Actions should be taken to reduce variance for performance
High	Low	Performance is unrelated to variance
Low	High	Reducing variance yields little benefit to the whole program
Low	Low	No action on variance optimization

FVSampler pinpoints a problematic function, it plots the metrics collected from all its sampled instances in the timeline. The variance pattern can effectively guide unique code optimization (e.g., Section 7.1).

7 EVALUATION

We evaluate FVSampler on a machine with two 18-core Intel Xeon E5-2699 v3 CPUs (Haswell) of 2.30GHz frequency running Linux 4.8.0. The machine has a private 32KB L1 cache, a private 256KB L2 cache, a shared 40MB L3 cache, and 128GB main memory. FVSampler subscribes to the precise PMU event `BR_INST_RETIRED.NEAR_CALL` to sample call instructions.

Overhead. Runtime overhead is measured as the ratio of the runtime of a program monitored with FVSampler to the runtime of its native execution. Table 2 shows the runtime overhead of FVSampler on two HPC benchmark suites — NERSC-8 [39] and CORAL-2 [28] as well as five HPC benchmarks — LULESH-2 [23], Sweep3D [26], MASNUM [44], Sequoia AMG2006 [27], and PARSEC-2.1 dedup [5]. Programs are compiled with `gcc-5.4.1 -O3` except MASNUM compiled with `icc-18.0.2 -O3`. MPI programs are compiled with MPICH-3.0.4 [25]. All MPI programs are run with 36 processes and all OpenMP programs are run with 36 threads, which are pinned to cores. We tune the sampling period to ensure that at least 30 samples are collected per second per thread. We use the PMU event `PERF_COUNT_HW_INSTRUCTIONS` in counting mode to count the number of instructions executed by each sampled function instance. We run each program five times and report the average runtime overhead. In Table 2, we can see that FVSampler typically incurs 6% runtime overhead. FVSampler can incur more overhead when profiling short-running programs (e.g., < 1 second) due to the fixed overhead of setting up PMUs and debug registers. Table 3 shows per-sample overhead and per-watchpoint-trap overhead, respectively. We can see that FVSampler typically incurs 44 microseconds overhead per sample and 11 microseconds overhead per watchpoint trap. In addition, FVSampler incurs average 7MB memory overhead per thread in all these programs. Such low overhead makes FVSampler appropriate for production runs.

Case Studies. Table 4 summarizes the performance issues found by FVSampler via function-level execution variance analysis. All programs are compiled with `gcc-5.4.1 -O3` and run with 36 threads on a single node except MASNUM that is compiled with `icc-18.0.2 -O3` and run with 36 MPI processes on a cluster. We quantify the performance improvement in execution time of the entire program. In the rest of this section, we describe the insights FVSampler offers to users and the optimizations on the problematic functions for each program shown in Table 4.

Table 2: FVSampler's runtime overhead in the unit of times (×). Some benchmarks are not covered due to compile-time or runtime errors.

Benchmark		Language	Programming model	Native runtime (sec)	Overhead
NERSC-8	AMG	C	MPI+OpenMP	42.14	1.07x
	GTC	Fortran	MPI+OpenMP	50.29	1.04x
	MILC	C	MPI+OpenMP	81.63	1.05x
	MiniFE	C++	MPI+OpenMP	53.17	1.08x
	PSNAP	C	MPI	49.66	1.05x
	SMB	C	MPI	113.43	1x
	SNAP	Fortran	MPI+OpenMP	43.91	1.06x
	STREAM	C/Fortran	MPI+OpenMP	35.33	1.06x
CORAL-2	CLOMP	C	MPI+OpenMP	21.21	1.06x
	MDTest	C	MPI	55.49	1.04x
	PENNANT	C++	MPI+OpenMP	24.76	1.09x
	Quicksilver	C++	MPI+OpenMP	27.61	1.08x
LULESH-2		C++	MPI+OpenMP	36.59	1.08x
Sweep3D		Fortran	MPI	34.78	1.06x
MASNUM		Fortran	MPI	67.12	1.12x
Sequoia AMG2006		C	MPI+OpenMP	44.13	1.05x
PARSEC-2.1dedup		C	Pthreads	20.42	1.11x
Median		–	–	–	**1.06x**
GeoMean		–	–	–	**1.06x**

Table 3: FVSampler's per-sample overhead and per-watchpoint-trap overhead in the unit of microseconds.

Benchmark		Overhead (microsecond)	
		Per sample	Per watchpoint trap
NERSC-8	AMG	24	8
	GTC	50	9
	MILC	44	11
	MiniFE	16	7
	PSNAP	13	7
	SMB	54	9
	SNAP	52	14
	STREAM	43	12
CORAL-2	CLOMP	154	27
	MDTest	71	12
	PENNANT	60	13
	Quicksilver	26	9
LULESH-2		59	11
Sweep3D		102	11
MASNUM		13	8
Sequoia AMG2006		24	13
PARSEC-2.1dedup		14	8
Median		**44**	**11**
GeoMean		**45**	**11**

Table 4: Overview of performance improvement guided by FVSampler.

Program	Inefficiency		Optimization	
	Call site of the problematic function	Symptom	Solution	Speedup
MASNUM [44]	propagat.inc(96, 103)	Linear search	Locality-friendly search	1.54×
Sequoia AMG2006 [27]	par_relax.c(1654, 1658)	Load imbalance	Reducing the granularity of parallel work	1.08×
NERSC-8 MiniFE [39]	SparseMatrix_functions.hpp(465, 474)	Poor data structure	Replacing C++ `set` with `unordered_set`	1.96×
PARSEC-2.1 dedup [5]	encoder.c(120, 226, 840, 891, 1003)	Poor hashing algorithm	Reducing hash collisions	1.08×

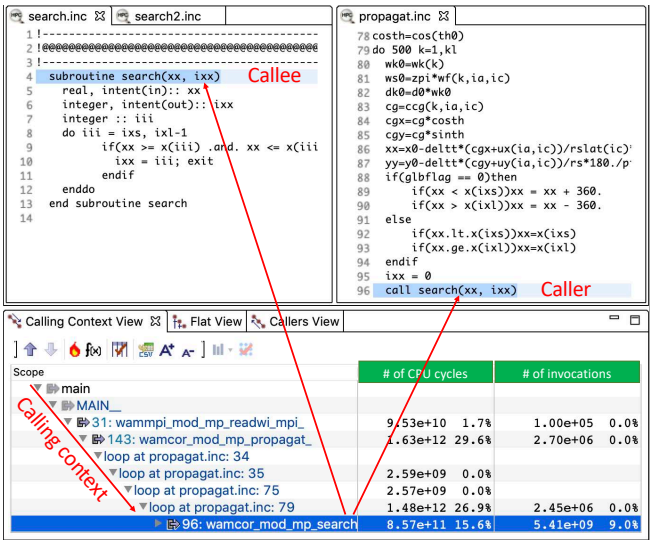


Figure 5: FVSampler's report for MASNUM, showing a problematic function — `search()` with full calling context.

7.1 MASNUM

MASNUM [44], one of the 2016 ACM Gordon Bell Prize finalists, forecasts ocean surface waves and climate change. It is written in Fortran and parallelized with MPI. We deploy FVSampler to the Stampede2 cluster located at Texas Advanced Computing Center (TACC). Stampede2 consists of 4,200 Intel Xeon Phi 7250 (Knights Landing) nodes and 1,736 Intel Xeon Platinum 8160 (SkyLake) nodes. We use only the SkyLake nodes to run MASNUM. Each SkyLake node consists of two 24-core sockets at 2.10GHz clock rate and 192 GB DDR4 main memory. Given the default input size of MASNUM, we run MASNUM with six nodes and totally 36 MPI processes. FVSampler reports that function `search()` is invoked in two call sites, which accounts for 27% of the total CPU cycles. The variance of the number of instructions executed in `search()` is high — 31%. One of the call sites with its full calling context as seen through FVSampler's GUI is highlighted in Figure 5. The GUI consists of three panes: the top pane shows the program source code, the bottom left pane shows the calling contexts, and the bottom right pane shows the metrics.

Figure 6a shows the number of instructions executed in different invocation instances of `search()` under the same calling context in one MPI process. The other 35 processes have similar patterns (not shown). In this figure, we make two observations. (1) The number of instructions executed in different invocations of `search()` has a clear periodical distribution pattern. (2) In each interval period, the number of instructions gradually increases and the number of instructions executed in adjacent invocations is similar, which indicates adjacent invocations of `search()` have similar workloads.

To understand the root cause of such execution pattern, we investigate the implementation of `search()`, as shown in the top pane in Figure 5. We find that `search()` is invoked in a loop and in each invocation, it performs a linear search (lines 8-12 in the file search.inc) for a given input `xx` over an immutable non-decreasing array `x` to determine the location of `xx`. With further investigation, we notice that the input `xx` has good value locality, that is, the parameter values are similar in adjacent invocations of `search()`. Consequently, in array `x`, the number of elements that are required to compare with `xx` in adjacent invocations are similar, which shows up as tiny horizontal bars (meaning approximately the same number of instructions in adjacent invocations) in Figure 6a.

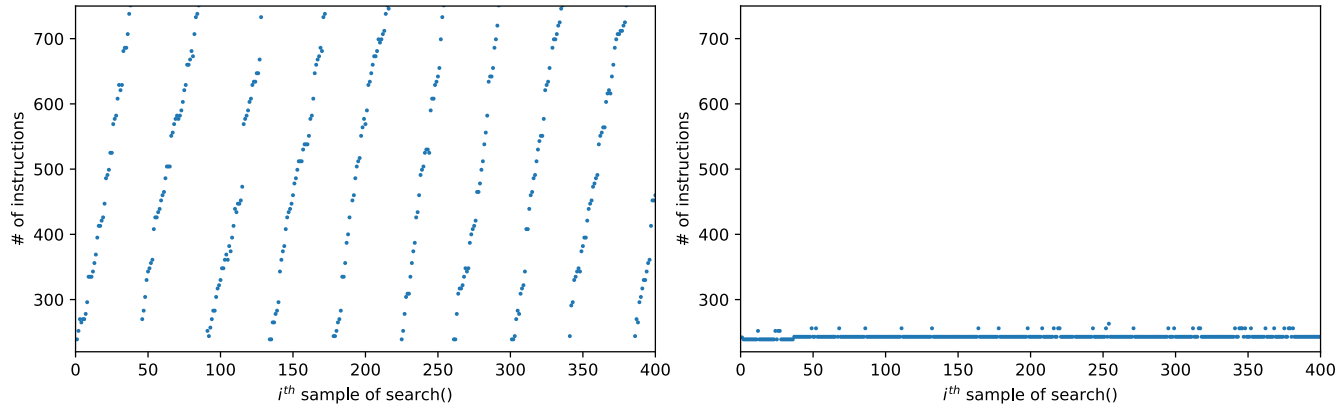
To improve serach(), we replace the linear search with a locality-friendly search. We memoize the location index of input `xx` when the current search instance finishes; in the next search, we begin at the previously recorded location index and alternate the linear search in both directions to the start and end of array `x`. This optimization yields a 1.54× speedup for the entire program and reduces the variance of the number of instructions executed in `search()` to 2%, as shown in Figure 6b, which is much flatter than the one shown in Figure 6a.

It is worth noting that changing the linear search to a binary search yields less than 2% performance improvement. Because the binary search, although reduces the number of instructions executed, hurts the data locality.

7.2 Sequoia AMG2006

Sequoia AMG2006 [27] is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. We study an optimized version from Liu and Mellor-Crummey [34]. The code is written in C and parallelized with MPI+OpenMP. We run AMG2006 on a $30 \times 30 \times 30$ grid.

FVSampler reports that function `hypre_BoomerAMGTraverse()` consumes 10% of the total CPU cycles and 36% of the total function invocations, as shown in Figure 7. FVSampler further identifies that the number of instructions executed in `hypre_BoomerAMGTraverse()` varies significantly on different threads, as shown in Figure 8. With the source code study, we find that `hypre_BoomerAMGTraverse()` is invoked in a loop nest and the outer loop is a statically scheduled OpenMP loop, which divides the iterations into equal-sized chunks and assigns them to each thread. It appears that each thread has an equal amount of work because each chunk consists of an equal number of iterations.



(a) Original. (b) Optimized.

Figure 6: Variance of the number of instructions executed in different invocation instances of `search()` in MASNUM. In the original MASNUM, `search()` performs a linear search over an immutable non-decreasing array and in the optimized MASNUM, we performs locality-friendly search over that array. We can clearly see that the variance shown in Figure (a) is much higher than the one shown in Figure (b).

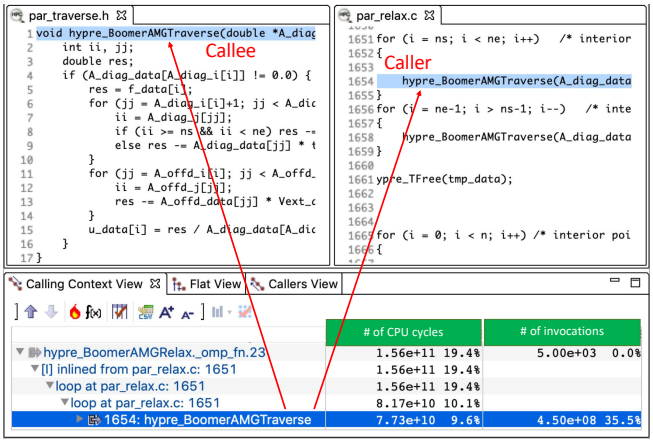


Figure 7: Inter-thread variance in Sequoia AMG2006. The number of instructions executed in `hypre_BoomerAMGTraverse()` varies significantly on different threads. `hypre_BoomerAMGTraverse()` takes different branches, which results in execution variance.

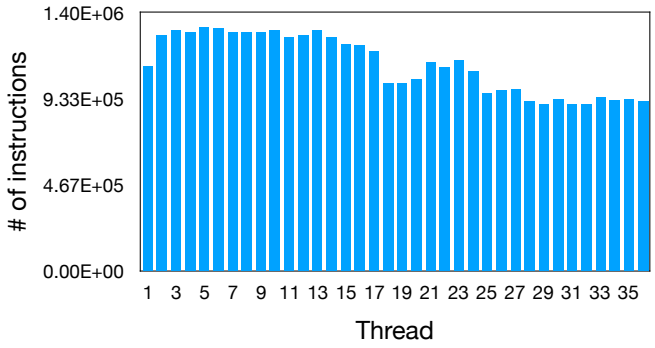


Figure 8: The number of instructions executed in `hypre_BoomerAMGTraverse()` on each thread.

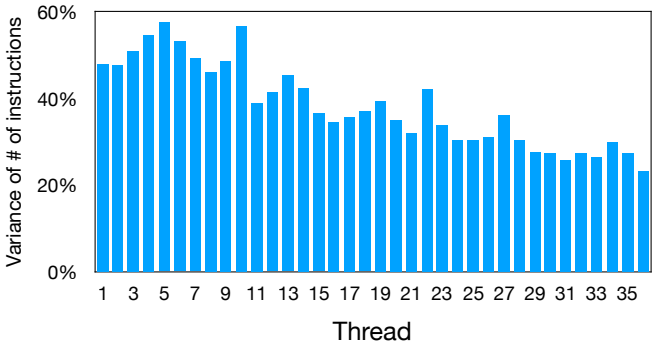


Figure 9: Variance of the number of instructions executed in different invocation instances of `hypre_BoomerAMGTraverse()` on each thread.

When investigating the function body, we find `hypre_BoomerAMGTraverse()` employs branches, which results in execution variance depending on the taken branch. This execution variance among threads is a symptom of load imbalance. The most straightforward optimization is to redistribute the iterations to different threads. However, as Figure 9 shows, the execution variance of `hypre_BoomerAMGTraverse()` inside each thread is also high. Thus, it is difficult to assess the workload of each iteration and achieve load balance via static scheduling.

To solve the load imbalance, we reduce the chunk size to $\frac{1}{5}$ of the original chunk size and employ dynamic scheduling to balance the work across threads. With this optimization, the variance of the work (number of instructions) assigned to each thread is reduced from 14% to 3%. FVSampler also identifies other functions with the similar issue and guides the similar optimization. Finally, the entire program gains a 1.08× speedup.

```
1  void impose_dirichlet(..., const std::set<typename MatrixType::
       GlobalOrdinalType>& bc_rows) {
2    ...
3    for(size_t i=0; i<A.rows.size(); ++i) {
4      ...
5      A.get_row_pointers(row, row_length, cols, coefs);
6      Scalar sum = 0;
7      for(size_t j=0; j<row_length; ++j) {
8  ▶     if (bc_rows.find(cols[j]) != bc_rows.end()) {
9          sum += coefs[j];
10         coefs[j] = 0;
11       }
12     }
13   }
14 }
```

Listing 1: Call site of `std::set::find()` in NERSC-8 MiniFE, which accounts for 22% of the total CPU cycles.

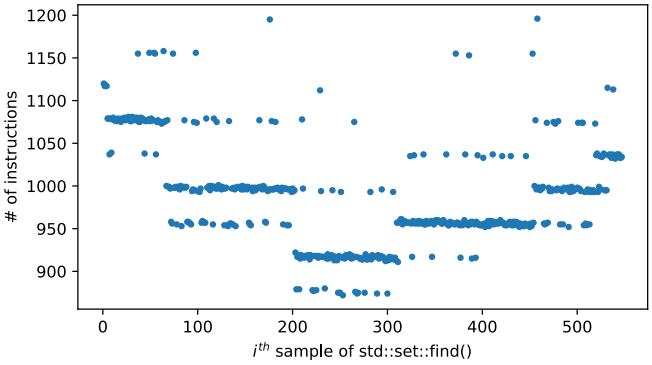


Figure 10: Variance of the number of instructions executed in different invocation instances of `std::set::find()` in NERSC-8 MiniFE. `std::set` in C++ is implemented as a Red-Black tree where a lookup operation requires one comparison in the best case and $O(\log n)$ comparisons in the worst case. Consequently, the number of instructions executed in different invocation instances of `std::set::find()` varies from one to $O(\log n)$.

7.3 NERSC-8 MiniFE

NERSC-8 MiniFE [39] employs the implicit finite-element method (FEM) to solve problems of engineering and mathematical physics. The code is written in C++ and parallelized with MPI+OpenMP. We apply FVSampler to evaluate it with the default input. Listing 1 highlights one of the hottest function — `std::set::find()` at line 8, which accounts for 22% of the total CPU cycles and is executed only on the master thread. FVSampler further reports the number of instructions executed in different invocation instances of `std::set::find()`, as shown in Figure 10. We can see that the work (number of instructions) performed by different instances varies significantly. The underlying implementation of `std::set` in C++ is a Red-Black tree where a lookup operation requires one comparison in the best case and $O(\log n)$ comparisons in the worst case. Hence, the number of comparisons involved in `std::set::find()` varies from one to $O(\log n)$, which shows up as the large execution variance.

To improve the lookup operation, we replace `std::set` with `std::unordered_set`. The latter uses a hash table to store elements, which requires expected $O(1)$ comparisons to look up an element. With this optimization, the execution variance reduces significantly, yielding a 1.96× speedup for the entire program.

```
1  struct hash_entry *hashtable_search(struct hashtable *h, void *k){
2    struct hash_entry *e;
3    unsigned int hashvalue, index;
4    hashvalue = hash(h,k);
5    index = indexFor(h->tablelength,hashvalue);
6    e = h->table[index];
7  ▶ while (NULL != e) {
8  ▶   if ((hashvalue == e->h) && (h->eqfn(k, e->k))) return e;
9  ▶   e = e->next;
10 ▶ }
11   ...
12 }
```

Listing 2: Inefficient implementation of `hashtable_search()` in PARSEC-2.1 dedup. Excessive hash collisions in linear hashing result in non-uniform linked lists.

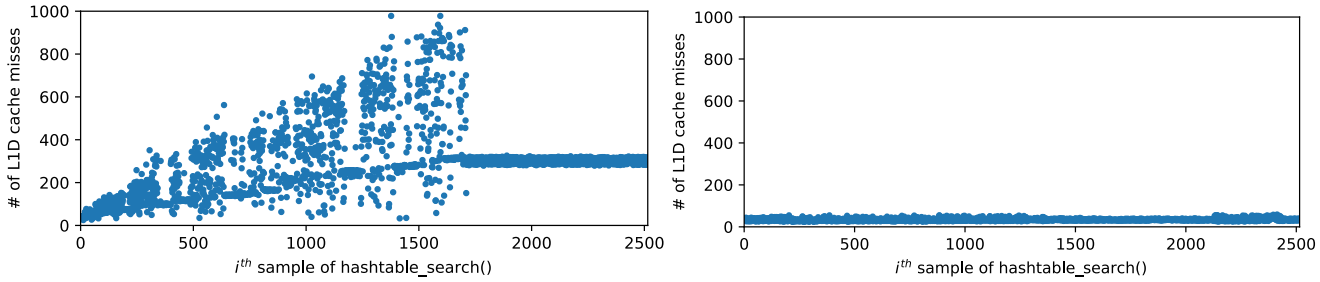
7.4 PARSEC-2.1 dedup

PARSEC-2.1 dedup [5] compresses data via deduplication. It is written in C and parallelized with `Pthreads`. With the native input, FVSampler reports that function `hashtable_search()` accounts for 11% of the total CPU cycles. Figure 11a shows the variance of `L1D` cache misses in different invocation instances of `hashtable_search()`. Listing 2 shows the implementation of `hashtable_search()`, which is invoked in a loop (not shown). In each invocation, `hashtable_search()` searches for an item in a linked list associated with a hash table entry. With further investigation, we notice that only ~2% of hash buckets are occupied, resulting in excessive collisions. Given different search items, the length of linked list traversal can be different, resulting in execution variance (measured in cache misses).

To optimize, one can improve the hash algorithm by uniformly distributing hash keys across all buckets to reduce the variance in traversing the linked list due to hash collisions. In this case study, we adopt Curtsinger and Berger's approach [10] to balance the hash key distribution, which reduces the variance of `L1D` cache misses in `hashtable_search()` from 56% to 16%, as shown in Figure 11b. The whole program gains a 1.08× speedup.

8 CONCLUSIONS AND FUTURE WORK

This paper presents FVSampler, a lightweight variance profiler for HPC applications. FVSampler advances the state-of-the-art in sampling profilers by demonstrating the ability to synchronize profiling samples precisely at procedure boundaries. Thus, FVSampler abandons code instrumentation for function-level monitoring. FVSampler adopts hardware performance monitoring units to sample function call and uses hardware debug registers to intercept the return from the same function invocation instance. FVSampler further collects the performance events, e.g., CPU cycles, instruction instances, cache misses, occurring in each sampled function instance and computes the variance metrics across different instances of the same function. FVSampler is able to pinpoint both intra-thread and inter-thread variance, which helps isolate performance problems in complex codes. FVSampler incurs low



(a) Original. **(b) Optimized.**

Figure 11: Variance of L1D cache misses in different invocation instances of `hashtable_search()` in PARSEC-2.1 dedup. `hashtable_search()` searches for an item in a linked list associated with a hash table entry. In the original dedup, only ~2% of hash buckets are occupied, resulting in excessive collisions. Given different search items, the length of linked list traversal can be significantly different, resulting in execution variance. In the optimized dedup, hash keys are uniformly distributed across buckets to reduce the variance in traversing the linked list.

runtime and memory overheads, which makes it attractive for production HPC codes. Guided by FVSampler, we are able to optimize several parallel applications, yielding up to a 1.96× speedup.

Our future direction is to explore execution variance beyond the procedure level. We will investigate a finer granularity of a series of basic blocks and also a coarser granularity of a series of procedures for a semantic interval.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments. This work is supported by Google Faculty Research Award, as well as the Thomas F. and Kate Miller Jeffress Memorial Trust, Bank of America, Trustee and any specified Program donor (if applicable). Shuyin Jiao is the corresponding author.

REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency Computation : Practice Expererience* 22, 6 (Apr 2010), 685–701.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 85–96.

[3] M. Arnold and P. F. Sweeney. 1999. Approximating the calling context tree via sampling. Technical Report, IBM.

[4] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 307–320.

[5] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[6] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Cambridge, MA, USA. AAI0807735.

[7] Milind Chabbi, Shasha Wen, and Xu Liu. 2018. Featherlight On-the-fly False-sharing Detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 152–167.

[8] Intel Corp. 2010. Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide. https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf.

[9] Intel Corp. 2018. Intel VTune. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[10] Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding Code That Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 184–197.

[11] Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. 2008. Cray Performance Analysis Tools. In *Tools for High Performance Computing*. Springer Berlin Heidelberg, 191–199.

[12] Paul J. Drongowski. 2007. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. https://pdfs.semanticscholar.org/5219/4b43b8385ce39b2b08ecd409c753e0efafe5.pdf.

[13] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 151–162.

[14] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca Performance Toolset Architecture. *Concurr. Comput. : Pract. Exper.* 22, 6 (April 2010), 702–719.

[15] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN '82)*. ACM, New York, NY, USA, 120–126.

[16] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 161–175.

[17] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. 2017. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 625–638.

[18] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. 2017. Reining in Long Tails in Warehouse-Scale Computers with Quick Voltage Boosting Using Adrenaline. *ACM Trans. Comput. Syst.* 35, 1, Article 2 (March 2017), 33 pages.

[19] Jiamin Huang, Barzan Mozafari, and Thomas F. Wenisch. 2017. Statistical Analysis of Latency Through Semantic Profiling. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 64–79.

[20] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L. Cox. 2016. TPC: Target-Driven Parallelism Combining Prediction and Correction to Reduce Tail Latency in Interactive Services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 129–141.

[21] Yunyun Jiang, Yi Yang, Tian Xiao, Tianwei Sheng, and Wenguang Chen. 2016. DRDDR: A Lightweight Method to Detect Data Races in Linux Kernel. *The Journal of Supercomputing* 72, 4 (April 2016), 1645–1659.

[22] Mark Scott Johnson. 1982. Some Requirements for Architectural Support of Software Debugging. In *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS I)*. ACM, New York, NY, USA, 140–148.

[23] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still. 2013. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 919–932.

[24] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. 2014. Efficient Tracing of Cold Code via Bias-free Sampling. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 243–254.

[25] Argonne National Laboratory. 2014. MPICH wiki. https://wiki.mpich.org/mpich/index.php.

[26] Lawrence Livermore National Laboratory. 1995. Sweep3D Benchmark Code. http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html.

[27] Lawrence Livermore National Laboratory. 2013. ASC Sequoia Benchmark Codes. https://asc.llnl.gov/sequoia/benchmarks.

[28] Lawrence Livermore National Laboratory. 2018. CORAL-2 Benchmarks. https://asc.llnl.gov/coral-2-benchmarks.

[29] Z. Lai, Y. Cui, M. Li, Z. Li, N. Dai, and Y. Chen. 2016. TailCutter: Wisely cutting tail latency in cloud CDN under cost constraints. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9.

[30] Linux. 2012. perf_event_open - Linux man page. https://linux.die.net/man/2/perf_event_open.

[31] Linux. 2015. Linux Perf Tool. https://perf.wiki.kernel.org/index.php/Main_Page.

[32] Hongyu Liu, Sam Silvestro, Xiaoyin Wang, Lide Duan, and Tongping Liu. 2019. CSOD: Context-sensitive Overflow Detection. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 50–60.

[33] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and Precise Error Detection via Evidence-based Dynamic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 911–922.

[34] Xu Liu and John Mellor-Crummey. 2014. A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 259–272.

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200.

[36] G Marin, G Jin, and J Mellor-Crummey. 2008. Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code. *Journal of Physics: Conference Series* 125 (jul 2008), 012087.

[37] R. E. McLear, D. M. Scheibelhut, and E. Tammaru. 1982. Guidelines for Creating a Debuggable Processor. In *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS I)*. ACM, New York, NY, USA, 100–106.

[38] John Mellor-Crummey, Robert J. Fowler, Gabriel Marin, and Nathan Tallent. 2002. HPCVIEW: A Tool for Top-down Analysis of Node Performance. *J. Supercomput.* 23, 1 (Aug. 2002), 81–104.

[39] NERSC. 2016. NERSC-8 / Trinity Benchmarks. http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks.

[40] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100.

[41] University of Maryland and University of Wisconsin. 2017. Putting the Performance in High Performance Computing. https://www.dyninst.org.

[42] Oracle Corp. 2017. Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html.

[43] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. 2010. Locating Cache Performance Bottlenecks Using Data Profiling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, New York, NY, USA, 335–348.

[44] Fangli Qiao, Wei Zhao, Xunqiang Yin, Xiaomeng Huang, Xin Liu, Qi Shu, Guansuo Wang, Zhenya Song, Xinfang Li, Haixing Liu, Guangwen Yang, and Yeli Yuan. 2016. A Highly Effective Global Surface Wave Numerical Simulation with Ultra-high Resolution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 5, 11 pages.

[45] Raja R. Sambasivan and Gregory R. Ganger. Submitted. Automated Diagnosis Without Predictability Is a Recipe for Failure. In *Presented as part of the*. USENIX.

[46] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing Performance Changes by Comparing Request Flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 43–56.

[47] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. 2008. OpenSpeedShop: An open source infrastructure for parallel performance analysis. *Sci. Program.* 16, 2-3 (April 2008), 105–121.

[48] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (May 2006), 287–311.

[49] M. Srinivas, B. Sinharoy, R. J. Eickemeyer, R. Raghavan, S. Kunkel, T. Chen, W. Maron, D. Flemming, A. Blanchard, P. Seshadri, J. W. Kellington, A. Mericas, A. E. Petruski, V. R. Indukuru, and S. Reyes. 2011. IBM POWER7 performance modeling, verification, and evaluation. *IBM JRD* 55, 3 (May-June 2011), 4:1–4:19.

[50] Pengfei Su, Qingsen Wang, Milind Chabbi, and Xu Liu. 2019. Pinpointing Performance Inefficiencies in Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 818–829.

[51] Z. Szebenyi, T. Gamblin, M. Schulz, B. R. d. Supinski, F. Wolf, and B. J. N. Wylie. 2011. Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs. In *2011 IEEE International Parallel Distributed Processing Symposium*. 640–651.

[52] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. 2010. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11.

[53] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. 2009. Binary Analysis for Measurement and Attribution of Program Performance. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 441–452.

[54] The Portland Group. 2011. PGPROF Profiler Guide Parallel Profiling for Scientists and Engineers. http://www.pgroup.com/doc/pgprofug.pdf.

[55] Q. Wang, X. Liu, and M. Chabbi. 2019. Featherlight Reuse-Distance Measurement. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 440–453.

[56] B. P. Welford. 1962. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics* 4, 3 (1962), 419–420.

[57] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for Software Inefficiencies with Witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 332–347.

[58] Wikipedia. 2019. Algorithms for calculating variance. https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf.

[59] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. 2019. Can We Trust Profiling Results?: Understanding and Fixing the Inaccuracy in Modern Profilers. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*. ACM, New York, NY, USA, 284–295.

[60] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. DBSherlock: A Performance Diagnostic Tool for Transactional Databases. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1599–1614.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We evaluate FVSampler on a machine with two 18-core Intel Xeon E5-2699 v3 CPUs (Haswell) of 2.30GHz frequency running Linux 4.8.0. The machine has a private 32KB L1 cache, a private 256KB L2 cache, a shared 40MB L3 cache, and 128GB main memory. FVSampler is built with gcc-5.4.1 -O3.

The benchmarks include two HPC benchmark suites — NERSC-8 and CORAL-2 as well as several HPC benchmarks — LULESH-2, Sweep3D, MASNUM, Sequoia AMG2006, and PARSEC-2.1 dedup. Programs are compiled with gcc-5.4.1 -O3 except MASNUM that is compiled with icc-18.0.2 -O3. MPI programs are compiled with MPICH-3.0.4. All MPI programs are run with 36 processes and all OpenMP programs are run with 36 threads, which are pinned to cores.

In addition, We also deploy FVSampler to the Stampede2 cluster located at Texas Advanced Computing Center (TACC). Stampede2 consists of 4,200 Intel Xeon Phi 7250 (Knights Landing) nodes and 1,736 Intel Xeon Platinum 8160 (SkyLake) nodes. We use only the SkyLake nodes to run MASNUM. Each SkyLake node consists of two 24-core sockets at 2.10GHz clock rate and 192 GB DDR4 main memory.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: No author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

`https://github.com/WitchTools/FVSampler`

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel Xeon E5-2699 v3 CPUs (Haswell) @ 2.30GHz, Intel Xeon Platinum 8160 (SkyLake) @ 2.10GHz

Operating systems and versions: Ubuntu 14.04 running Linux kernel 4.8.0

Compilers and versions: gcc-5.4.1, icc-18.0.2

Applications and versions: NERSC-8, CORAL-2, LULESH-2, Sweep3D, MASNUM, Sequoia AMG2006, PARSEC-2.1 dedup

Libraries and versions: MPICH-3.0.4

Paper Modifications: We propose FVSampler, a lightweight, sampling-based variance profiler. FvSampler is able to quantify variance across different invocations of the same function without requiring code instrumentation. Guided by FvSampler, we are able to optimize several parallel applications by mitigating the causes of variance, yielding up to a 1.96× speedup.

Output from scripts that gathers execution environment information.

```
Distributor ID:        Ubuntu
Description:        Ubuntu 14.04.5 LTS
Release:        14.04
Codename:        trusty
+ uname -a
Linux ksunserver3 4.8.0 #1 SMP Tue Jul 24 00:36:17 EDT
↪  2018 x86_64 x86_64 x86_64 GNU/Linux
+ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                72
On-line CPU(s) list:   0-71
Thread(s) per core:    2
Core(s) per socket:    18
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Stepping:              2
CPU MHz:               1201.379
BogoMIPS:              4590.66
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              46080K
NUMA node0 CPU(s):     0-17,36-53
NUMA node1 CPU(s):     18-35,54-71
+ cat /proc/meminfo
MemTotal:       131626784 kB
MemFree:        100988292 kB
MemAvailable:   129709472 kB
Buffers:         2453660 kB
Cached:         25138376 kB
SwapCached:        11972 kB
Active:         15095556 kB
Inactive:       12675996 kB
Active(anon):     176440 kB
Inactive(anon):    15596 kB
Active(file):   14919116 kB
Inactive(file): 12660400 kB
Unevictable:           32 kB
Mlocked:               32 kB
SwapTotal:      133944316 kB
SwapFree:       133596456 kB
```

```
Dirty:                20 kB
Writeback:             0 kB
AnonPages:        169156 kB
Mapped:            67236 kB
Shmem:             12268 kB
Slab:            2429240 kB
SReclaimable:    2203460 kB
SUnreclaim:       225780 kB
KernelStack:       20848 kB
PageTables:        31468 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    199757708 kB
Committed_AS:    3747948 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
HardwareCorrupted:     0 kB
AnonHugePages:    135168 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
CmaTotal:              0 kB
CmaFree:               0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:      301508 kB
DirectMap2M:     5718016 kB
DirectMap1G:    130023424 kB
+ inxi -F -c0
collect_environment.sh: line 14: inxi: command not
↪  found
+ lsblk -a
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
ram11    1:11   0    64M  0 disk
ram2     1:2    0    64M  0 disk
loop1    7:1    0         0 loop
ram0     1:0    0    64M  0 disk
ram9     1:9    0    64M  0 disk
ram7     1:7    0    64M  0 disk
loop6    7:6    0         0 loop
ram14    1:14   0    64M  0 disk
ram5     1:5    0    64M  0 disk
loop4    7:4    0         0 loop
ram12    1:12   0    64M  0 disk
ram3     1:3    0    64M  0 disk
loop2    7:2    0         0 loop
ram10    1:10   0    64M  0 disk
ram1     1:1    0    64M  0 disk
loop0    7:0    0         0 loop
sda      8:0    0 743.2G  0 disk
├─sda2   8:2    0     1K  0 part
├─sda5   8:5    0 127.8G  0 part [SWAP]
└─sda1   8:1    0 615.5G  0 part /
ram8     1:8    0    64M  0 disk
loop7    7:7    0         0 loop
ram15    1:15   0    64M  0 disk
ram6     1:6    0    64M  0 disk
loop5    7:5    0         0 loop
ram13    1:13   0    64M  0 disk
ram4     1:4    0    64M  0 disk
loop3    7:3    0         0 loop
+ lsscsi -s
collect_environment.sh: line 16: lsscsi: command not
↪  found
+ module list
collect_environment.sh: line 17: module: command not
↪  found
+ nvidia-smi
collect_environment.sh: line 18: nvidia-smi: command
↪  not found
+ lshw -short -quiet -sanitize
+ cat
H/W path                Device     Class
↪  Description
=====================================================┘
↪  =========
                                   system
                                   ↪  UCSC-C220-M4S
                                   ↪  ()
/0                                 bus
↪  UCSC-C220-M4S
/0/0                               memory         64KiB
↪  BIOS
/0/22                              memory
↪  128GiB System Memory
/0/22/0                            memory         16GiB
↪  DIMM 2133 MHz (0.5 ns)
/0/22/1                            memory          DIMM
↪  [empty]
/0/22/2                            memory          DIMM
↪  [empty]
/0/22/3                            memory         16GiB
↪  DIMM 2133 MHz (0.5 ns)
/0/22/4                            memory          DIMM
↪  [empty]
/0/22/5                            memory          DIMM
↪  [empty]
/0/22/6                            memory         16GiB
↪  DIMM 2133 MHz (0.5 ns)
/0/22/7                            memory          DIMM
↪  [empty]
/0/22/8                            memory          DIMM
↪  [empty]
/0/22/9                            memory         16GiB
↪  DIMM 2133 MHz (0.5 ns)
/0/22/a                            memory          DIMM
↪  [empty]
```

```
/0/22/b                              memory         DIMM
↪  [empty]
/0/22/c                             memory         16GiB
↪  DIMM 2133 MHz (0.5 ns)
/0/22/d                              memory         DIMM
↪  [empty]
/0/22/e                              memory         DIMM
↪  [empty]
/0/22/f                             memory         16GiB
↪  DIMM 2133 MHz (0.5 ns)
/0/22/10                             memory         DIMM
↪  [empty]
/0/22/11                             memory         DIMM
↪  [empty]
/0/22/12                            memory         16GiB
↪  DIMM 2133 MHz (0.5 ns)
/0/22/13                             memory         DIMM
↪  [empty]
/0/22/14                             memory         DIMM
↪  [empty]
/0/22/15                            memory         16GiB
↪  DIMM 2133 MHz (0.5 ns)
/0/22/16                             memory         DIMM
↪  [empty]
/0/22/17                             memory         DIMM
↪  [empty]
/0/3c                                memory
↪  576KiB L1 cache
/0/3d                                memory
↪  576KiB L1 cache
/0/3e                                memory
↪  4608KiB L2 cache
/0/3f                               memory         45MiB
↪  L3 cache
/0/40                                processor
↪  Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
/0/41                                memory
↪  576KiB L1 cache
/0/42                                memory
↪  576KiB L1 cache
/0/43                                memory
↪  4608KiB L2 cache
/0/44                               memory         45MiB
↪  L3 cache
/0/45                                processor
↪  Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
/0/100                               bridge         Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DMI2
/0/100/1                             bridge         Xeon
↪  E7 v3/Xeon E5 v3/Core i7 PCI Express Root Port 1
/0/100/1/0              eth2         network        I350
↪  Gigabit Network Connection
/0/100/1/0.1            eth0         network        I350
↪  Gigabit Network Connection
/0/100/1.1                           bridge         Xeon
↪  E7 v3/Xeon E5 v3/Core i7 PCI Express Root Port 1
/0/100/2                             bridge         Xeon
↪  E7 v3/Xeon E5 v3/Core i7 PCI Express Root Port 2
/0/100/2/0                           bridge         VIC
↪  82 PCIe Upstream Port
/0/100/2/0/0                         bridge         VIC
↪  PCIe Downstream Port
/0/100/2/0/0/0                       generic        VIC
↪  Management Controller
/0/100/2/0/1                         bridge         VIC
↪  PCIe Downstream Port
/0/100/2/0/1/0                       bridge         VIC
↪  PCIe Upstream Port
/0/100/2/0/1/0/0                     bridge         VIC
↪  PCIe Downstream Port
/0/100/2/0/1/0/0/0      em1          network        VIC
↪  Ethernet NIC
/0/100/2/0/1/0/1                     bridge         VIC
↪  PCIe Downstream Port
/0/100/2/0/1/0/1/0      em2          network        VIC
↪  Ethernet NIC
/0/100/2/0/1/0/2                     bridge         VIC
↪  PCIe Downstream Port
/0/100/2/0/1/0/2/0                   bus            VIC
↪  FCoE HBA
/0/100/2/0/1/0/3                     bridge         VIC
↪  PCIe Downstream Port
/0/100/2/0/1/0/3/0                   bus            VIC
↪  FCoE HBA
/0/100/2.2                           bridge         Xeon
↪  E7 v3/Xeon E5 v3/Core i7 PCI Express Root Port 2
/0/100/2.2/0            scsi0        storage
↪  MegaRAID SAS-3 3108 [Invader]
/0/100/2.2/0/2.0.0      /dev/sda     disk
↪  797GB UCSC-MRAID12G
/0/100/2.2/0/2.0.0/1    /dev/sda1    volume
↪  615GiB EXT4 volume
/0/100/2.2/0/2.0.0/2    /dev/sda2    volume
↪  127GiB Extended partition
/0/100/2.2/0/2.0.0/2/5  /dev/sda5    volume
↪  127GiB Linux swap / Solaris partition
/0/100/3                             bridge         Xeon
↪  E7 v3/Xeon E5 v3/Core i7 PCI Express Root Port 3
/0/100/5                             generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Address Map, VTd_Misc,
↪  System Management
/0/100/5.1                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Hot Plug
/0/100/5.2                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 RAS, Control Status and
↪  Global Errors
/0/100/5.4                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 I/O APIC
```

```
/0/100/11                           generic
↪  C610/X99 series chipset SPSR
/0/100/11.4                         storage
↪  C610/X99 series chipset sSATA Controller [AHCI
↪  mode]
/0/100/16                           communication
↪  C610/X99 series chipset MEI Controller #1
/0/100/16.1                         communication
↪  C610/X99 series chipset MEI Controller #2
/0/100/1a                        bus          C610/X99
↪  series chipset USB Enhanced Host Controller #2
/0/100/1c                           bridge
↪  C610/X99 series chipset PCI Express Root Port #1
/0/100/1c.3                         bridge
↪  C610/X99 series chipset PCI Express Root Port #4
/0/100/1c.3/0                       display        MGA
↪  G200e [Pilot] ServerEngines (SEP1)
/0/100/1d                        bus          C610/X99
↪  series chipset USB Enhanced Host Controller #1
/0/100/1f                           bridge
↪  C610/X99 series chipset LPC Controller
/0/100/1f.2                         storage
↪  C610/X99 series chipset 6-Port SATA Controller
↪  [AHCI mode]
/0/1                                generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 0
/0/3                                generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 0
/0/4                                generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 0
/0/6                                generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 1
/0/7                                generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 1
/0/8                                generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 1
/0/9                             generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/a                             generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/b                             generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/c                                generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/d                                generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/e                                generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/f                                generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/10                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/11                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/12                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/13                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/14                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/15                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/16                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/17                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/18                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/19                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/1a                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/1b                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/1c                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/1d                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/1e                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/1f                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/20                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/21                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/23                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/24                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/25                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/26                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 PCIe Ring Interface
/0/27                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 PCIe Ring Interface
/0/28                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
/0/29                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
/0/2a                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
```

```
/0/2b                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Home Agent 0
/0/2c                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Home Agent 0
/0/2d                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Home Agent 1
/0/2e                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Home Agent 1
/0/2f                            generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  0 Target Address, Thermal & RAS Registers
/0/30                            generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  0 Target Address, Thermal & RAS Registers
/0/31                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel Target Address Decoder
/0/32                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel Target Address Decoder
/0/33                            generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Channel 0/1 Broadcast
/0/34                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO Global Broadcast
/0/35                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 0 Thermal Control
/0/36                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 1 Thermal Control
/0/37                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 0 ERROR Registers
/0/38                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 1 ERROR Registers
/0/39                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/3a                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/3b                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/46                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/47                            generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  1 Target Address, Thermal & RAS Registers
/0/48                            generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  1 Target Address, Thermal & RAS Registers
/0/49                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel Target Address Decoder
/0/4a                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel Target Address Decoder
/0/4b                            generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Channel 2/3 Broadcast
/0/4c                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO Global Broadcast
/0/4d                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 0 Thermal Control
/0/4e                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 1 Thermal Control
/0/4f                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 0 ERROR Registers
/0/50                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 1 ERROR Registers
/0/51                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/52                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/53                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/54                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/55                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Power Control Unit
/0/56                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Power Control Unit
/0/57                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Power Control Unit
/0/58                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Power Control Unit
/0/59                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Power Control Unit
/0/5a                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 VCU
/0/5b                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 VCU
/0/2                               bridge         Xeon
↪  E7 v3/Xeon E5 v3/Core i7 PCI Express Root Port 2
/0/5                               generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Address Map, VTd_Misc,
↪  System Management
/0/5.1                             generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Hot Plug
/0/5.2                             generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 RAS, Control Status and
↪  Global Errors
/0/5.4                             generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 I/O APIC
/0/5c                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 0
/0/5d                              generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 0
```

```
/0/5e                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 0
/0/5f                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 1
/0/60                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 1
/0/61                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 QPI Link 1
/0/62                        generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/63                        generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/64                        generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/65                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/66                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/67                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/68                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/69                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/6a                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/6b                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/6c                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/6d                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/6e                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/6f                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/70                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/71                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/72                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/73                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/74                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/75                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/76                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Unicast Registers
/0/77                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/78                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/79                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/7a                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/7b                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/7c                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/7d                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/7e                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 PCIe Ring Interface
/0/7f                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 PCIe Ring Interface
/0/80                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
/0/81                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
/0/82                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
/0/83                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Home Agent 0
/0/84                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Home Agent 0
/0/85                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Home Agent 1
/0/86                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Home Agent 1
/0/87                        generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  0 Target Address, Thermal & RAS Registers
/0/88                        generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  0 Target Address, Thermal & RAS Registers
/0/89                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel Target Address Decoder
/0/8a                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel Target Address Decoder
/0/8b                        generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Channel 0/1 Broadcast
/0/8c                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO Global Broadcast
/0/8d                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 0 Thermal Control
/0/8e                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 1 Thermal Control
```

```
/0/8f                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 0 ERROR Registers
/0/90                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 1 ERROR Registers
/0/91                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/92                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/93                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/94                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/95                         generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  1 Target Address, Thermal & RAS Registers
/0/96                         generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  1 Target Address, Thermal & RAS Registers
/0/97                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel Target Address Decoder
/0/98                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel Target Address Decoder
/0/99                         generic       Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Channel 2/3 Broadcast
/0/9a                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO Global Broadcast
/0/9b                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 0 Thermal Control
/0/9c                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 1 Thermal Control
/0/9d                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 0 ERROR Registers
/0/9e                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 1 ERROR Registers
/0/9f                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/a0                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/a1                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/a2                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/a3                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Power Control Unit
/0/a4                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Power Control Unit
/0/a5                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Power Control Unit
/0/a6                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Power Control Unit
/0/a7                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 Power Control Unit
/0/a8                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 VCU
/0/a9                           generic        Xeon
↪  E7 v3/Xeon E5 v3/Core i7 VCU
```