Generating Miss Rate Curves with Low Overhead Using Existing Hardware

by

Tom Walsh

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Generating Miss Rate Curves with Low Overhead Using Existing Hardware

Tom Walsh

Master of Science

Graduate Department of Computer Science

University of Toronto

2009

Miss Rate Curves (MRCs) for main memory have been proposed as a representation of memory utilization for use in a range of optimizations in the area of memory management. Various techniques exist for their creation; however, all real-world methods of MRC generation must make trade-offs between overhead and accuracy. Proposals for new hardware techniques exist, but have yet to be implemented in actual hardware. We propose the use of the Intel PEBS (Precise Event-Based Sampling) performance monitoring capability for the task of MRC generation on existing commodity hardware.

We use PEBS to generate MRCs and compare them against MRCs generated through instrumentation, finding the PEBS MRCs to be good, but imperfect approximations, while keeping average PEBS overheads below 5%. We were unable to show that PEBS is better or worse than existing techniques, but believe we have succeeded in showing the promise of the use of general purpose performance monitoring hardware for this task and in motivating future research and development in this area.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This paper describes the use of the hardware performance monitoring unit on existing Intel processors to generate miss-rate curves to further our understanding of applications' memory requirements and advance our memory management techniques. Section 1.1 motivates our research in the area of memory management. Section 1.2 describes Miss Rate Curves, which we believe are a valuable tool in addressing some of the challenges described in Section 1.1. Section 1.3 motivates our choice of existing performance hardware as the means by which to generate Miss Rate Curves. Finally, Section 1.4 gives an overview of the remainder of this paper.

1.1 Why Memory Management?

The increasing performance gap between CPUs and main memory, known as the memory wall, means that many applications can process data faster than they can fetch it from memory [17]. When insufficient memory is allocated, however, the cost of a page fault becomes the dominant factor in the cost of a memory access [6]. With insufficient memory we must reduce the cost of paging if we are to beat the memory wall.

Faster storage, such as solid-state disks, may help to partially close the memory-disk gap in the future, but the gap will still remain. More memory can be purchased, but this is

an expensive and power-hungry solution. In scientific or financial computing, algorithms may be tailored to the available memory; however, this assumes dedicated hardware. In a general purpose or server computing environment where multiple applications are frequently being run in parallel, and workloads are often dynamic in nature, memory management is still critical.

Memory is getting larger and cheaper, but recent computing trends towards greater mobility and virtualization reduce the memory available to each application in two ways: by reducing the overall memory and by increasing the number of applications being run.

While memory has always been critical on the desktop, the emergence of mobile computing devices such as laptops and netbooks means that we often have less memory available on our systems, while still demanding high performance. In particular modern streaming and interactive web applications can quickly use up available memory[1]. Users frequently run multiple applications or view multiple web sites simultaneously, switching rapidly between them, and expect this to be near instantaneous. However, cost and power constraints make it unrealistic to put large amounts of memory in a notebook or netbook.

On the server, virtual machines are increasingly popular, consolidating multiple operating system instances and applications onto a single physical machine. This introduces an additional layer of complexity in memory management, as the virtual machine monitor or hypervisor is now responsible for allocating memory between multiple virtual machines. Each virtual machine instance faces increased memory pressure as it only receives a fraction of the host's physical memory. As virtual machines become increasingly dynamic, through virtual machine migration or cloning, as in the Snowflock project[14, 19], the memory load on a single physical machine becomes increasingly dynamic as well. For servers, power is also an increasingly important constraint, and both cost and power con-

[1]Launching the Safari web browser, logging in to facebook.com and playing a video uses over 300MB of memory for the browser alone on Mac OS 10.6

strain the available memory that can be installed into a system. Despite falling memory costs, the cost of memory dominates the cost of large server systems today.

In both the server and mobile spaces, increasingly large numbers of processes are accessing memory, not only producing increased memory pressure on each application, but also increasing the complexity of managing memory. The challenge is to find an efficient means of allocating memory between many processes.

1.2 Why Miss Rate Curves?

We contend that the quality of memory management algorithms is constrained primarily by the information available to the operating system from the hardware. Today's algorithms do the best they can with the very limited information provided by the hardware's "referenced" and "dirty" bits, which tell the operating system whether a page of memory has been used and whether the page has been modified. If the operating system wants to know more detailed information about its pages, such as the reuse distance or the relative ordering of page accesses, the OS must attempt to estimate these properties itself based on the limited information available. New models of memory utilization are required and the hardware must support these abstractions.

For memory allocation, we would like to make a trade-off between the amount of memory allocated and performance. To do so, we would like to know how an application's performance depends on its memory allocation. Miss Rate Curves (MRCs) provide this information. MRCs show the number of page faults of a process as a function of the amount of memory allocated. Chapter 2 describes MRCs and their applications in more detail.

Miss Rate Curves may be the new model that we are looking for, but first it is necessary to show that they can be produced efficiently and they can be used to improve our memory management algorithms.

1.3 Why Performance Monitoring Hardware?

There are two primary challenges related to MRCs. The first is how to generate accurate MRCs with low enough overhead for practical online use. The second is identifying the best algorithms and mechanisms for online MRC-based optimization. This work focuses on the first challenge, as we believe a solid understanding of MRC properties is a prerequisite for developing good MRC-based algorithms and good MRCs are needed in order to develop this understanding.

A number of methods for MRC generation exist, as described in Section 2.3. Generally those that do not rely on specialized hardware have high overheads. Those that rely on specialized MRC-generating hardware are purely theoretical as the hardware does not exist. We attempt to find a middle ground by using existing specialized hardware, the CPU's Performance Monitoring Unit (PMU), to generate MRCs. Our goal is low overhead and good accuracy on currently available real-world hardware.

1.4 Overview

Chapter 2 describes MRCs in more detail, describing what they are, how they are used, and how they are generated. Chapter 3 describes the use of performance monitoring facilities, particularly Intel's Precise Event-Based Sampling (PEBS), and it's applicability to memory-related tasks. Chapter 4 describes, at a high level, the challenges of generating MRCs using PEBS, while Chapter 5 goes into the details of our implementation. Chapter 6 evaluates the success of PEBS-based MRC generation and Chapter 7 concludes by discussing future directions for MRC-based research and for PMU-based memory tracking.

Chapter 2

Miss Rate Curves

Miss Rate Curves (MRCs) have been developed to better understand the memory requirements and utilization of applications or systems in main memory [3, 15, 32, 33], file caches [12, 20, 34], or on-chip L2 cache [21, 27, 28]. They serve both as visualizations to aid human understanding of memory utilization and as input for resource-management algorithms. Their use has been proposed for memory allocation [3, 15, 33], page replacement [3, 32], memory prefetching [3], memory power management [33], and application-specific memory optimization [32].

Technically, the MRCs discussed in this paper are LRU MRCs as they assume an LRU (Least Recently Used) replacement policy. Miss rate curves can be determined for other replacement policies; however, online MRC generation using Mattson's Stack Algorithm as described in Section 4.3 is only possible for stack-based replacement policies such as LRU or MRU (Most Recently Used). For readability, we use "MRC" throughout this paper to mean "LRU MRC", except where prefaced by another replacement policy, as in "MRU MRC".

A number of applications of MRC data have been proposed. These are described in Section 2.2. The adoption of these techniques, however, has been slowed by the cost of obtaining MRC data. Section 2.3 describes the techniques that have been used for

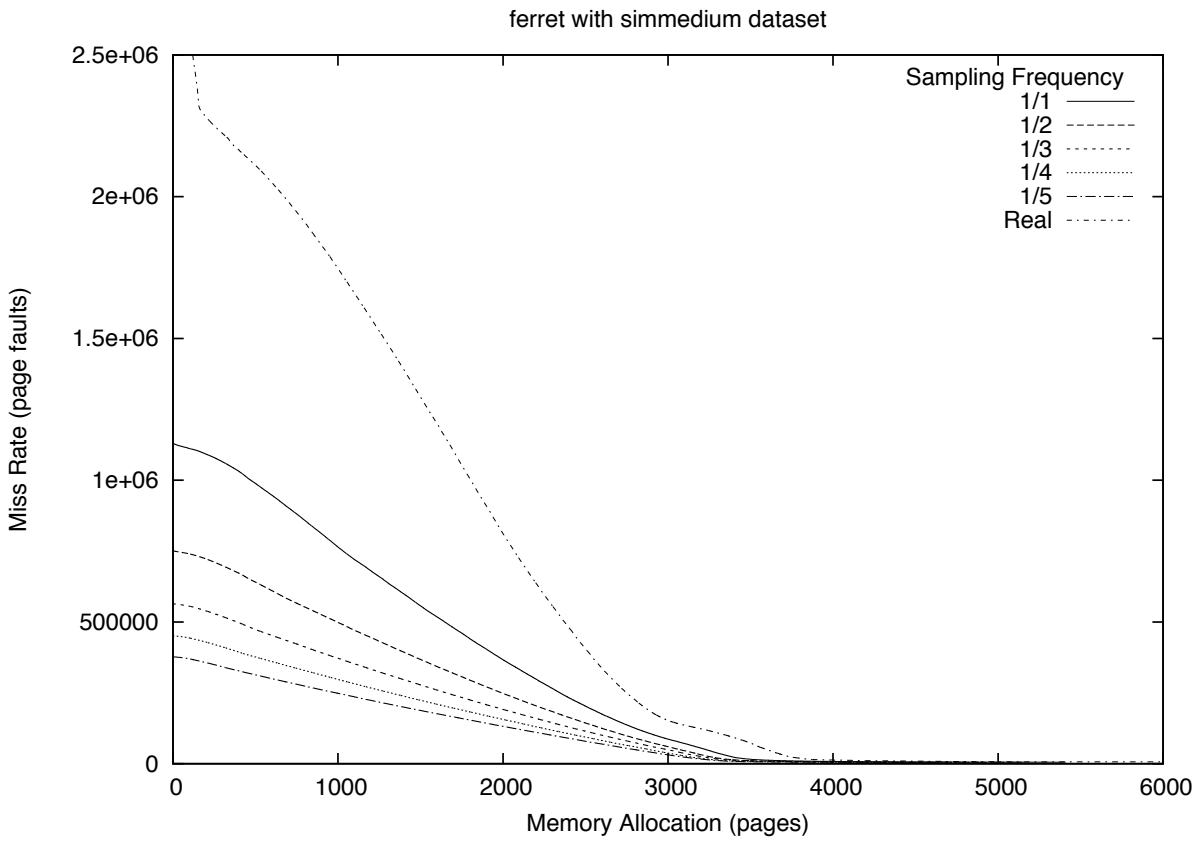


Figure 2.1: A Miss Rate Curve

generating MRCs on-line. In Section 2.4 the holes in our existing knowledge of MRCs are discussed.

2.1 Visualizing Memory Requirements with MRCs

2.1.1 What is an MRC?

An MRC is a curve plotting the rate at which misses occur when a process is executed with a specific amount of memory given to that process. For our purposes, since we are evaluating main memory, misses are page faults and the memory size is recorded in pages. In theory an MRC could be generated by running the process with every possible memory allocation and recording the results. However, for real-world use, more efficient methods are required.

The MRCs in this paper include all misses, both capacity misses (misses due to insufficient memory) and compulsory misses (misses due to accessing a page for the first time). MRCs are not always represented in this manner, as some authors choose to show only capacity misses. The inclusion of compulsory misses helps to show the total size of the MRC, and makes it easier to see the shape of the MRC as it is not obscured by the X-axis.

Another varying property of MRCs is the manner in which the miss rate is shown. Many papers use MPKI (Misses Per thousand(K) completed Instructions). Others chose to show the Miss Ratio rather than the Miss Rate, showing the fraction of all memory accesses that result in misses. We choose to show the absolute number of misses. The curves in all cases are identical, with the only difference being the scaling of the Y-axis.

2.1.2 Understanding Application Memory Requirements

The seemingly basic question in attempting to understand an application's memory requirements is "how much memory does it need?" This question, however, is not so basic, as it raises the question "for what?" For truly optimal performance, many applications need much more memory than can realistically be provided. Simply asking "how much memory does it need?" is not enough. The question is "how much memory does it need for a given level of performance?" or inversely "how will an application perform with a given amount of memory?"

MRCs answer this question. They show, for a given time-slice, the application's "memory performance" for every possible memory size, under the assumption of LRU (Least Recently Used) page replacement. "Memory performance" for our purposes is determined by the rate of page faults. Memory size is shown as the number of pages allocated to the application. In this paper, MRCs show the rate of page faults on the Y-axis and the number of allocated pages on the X-axis. MRCs may also be used for other levels of the memory hierarchy. For instance, in the case of an L2 cache, the MRC

would typically show L2 cache misses on the y-axis and the number of L2 cache lines on the x-axis, under the assumption of a fully-associative cache. Non-LRU memory MRCs are possible, but have only been used in one of the studies we are aware of [3], where MRU (Most Recently Used) MRCs were evaluated.

Given an MRC's precise knowledge about the memory requirements of a region[1] in memory, memory may be managed in a more informed manner than with current techniques. From the MRC, we can determine, for any allocation of memory, how memory-constrained the region will be, and this information can be used in making memory-management decisions. Memory may be allocated to different regions of memory to optimize overall system performance or the performance of specific processes. If a system's memory requirements are determined to be less than the amount of memory provided by hardware, memory may be shut down to save power.

2.2 Applications of MRCs and Memory Traces

This section describes a number of proposed applications of main memory MRCs in computer systems. While it focuses on MRCs, applications are included that are tangentially related due to their dependence upon pre-MRC data, that is data that must be collected for MRC generation. MRCs for caches [21, 27, 28] and file buffers [12, 20, 34] are also useful.

[1]Defining a "region" of memory is an interesting but orthogonal problem. This work avoids the issue by treating each process' address space as a single region. However, for an operating system it may be preferable to have multiple regions per application, isolating large data structures with different access patterns and hence different memory requirements into independent regions, or to combine many smaller processes into a single region. A region may be generally viewed as a chunk of memory (not necessarily contiguous) for which the system is tracking memory utilization as a single unit.

2.2.1 Memory Allocation

Detailed knowledge of the memory usage requirements for the regions of memory active within a computer system may be used to allocate physical memory to individual regions, for optimization of performance and fairness. For performance, the most basic approach has been greedy allocation [3, 15, 33] favoring the process with the greatest marginal utility at each allocation step. This approach is provably optimal for a convex MRC [33], but may result in highly non-optimal allocations for regions with plateaus or large concave portions in their memory requirements. Greedy algorithms with look-ahead [21] have been proposed to see past these plateaus, but have not been thoroughly investigated.

The idea of fair allocation has been proposed [15, 33], but no satisfactory definition of fairness seems to exist, and allocators have not been proposed using MRC data for anything other than performance optimization. Weighted utility functions are in use [33] for prioritizing different memory regions, and require only that the scale of the MRCs used for input be adjusted. Varying the weighting between regions to reflect process priorities is a potential approach for priority-based optimization. However, to our knowledge this has not yet been studied.

In addition to their use for allocation between regions at the operating system level, MRCs have been used in memory allocation between virtual machines at the level of the hypervisor [15].

2.2.2 Page Replacement

An operating system's memory management policies must deal with two orthogonal issues: allocating memory between regions and choosing pages for replacement from within a given region. The previous section described how MRCs may be used to solve the former problem. This section describes how an LRU stack, which we see in Section 4.3 is a pre-requisite data structure for MRC generation, can be used for page replacement.

While MRC-based memory allocation may be achieved without modifying the underlying page replacement policy of the operating system [33], most allocation-based work also includes a simple replacement algorithm based upon the LRU stack [3, 32]. PATH [3] goes further, looking at LRU, MRU, and LIRS [10] replacement as well as adaptively switching between replacement policies.

The page replacement techniques above, with the exception of adaptively switching between LRU and MRU replacement, rely solely upon the LRU stack, rather than MRC data. They are included here because the the LRU stack is a necessary intermediate stage in MRC creation.

2.2.3 Prefetching

PATH [3] uses page access traces, an intermediate stage towards LRU stack and MRC creation, for prefetching sequences of pages showing temporal locality. This approach may either replace or complement existing prefetching techniques based upon spatial proximity.

2.2.4 Power Consumption

One of the original applications of MRCs was reducing power consumption [33]. This is accomplished by moving pages with shorter reuse distance[2] onto the same memory chips and then powering down chips with low utility.

2.2.5 Application-Specific Optimization

CRAMM [32] allows a Java VM to use OS memory usage information (gathered via MRCs) to maximize its heap size without causing thrashing. To do this, it requires an

[2]Reuse distance describes how many other pages are touched between subsequent accesses to a given page. This may be visualized as the distance between the location of a page in the LRU stack when it is accessed and the head of the LRU stack.

interface for passing memory usage information between the OS and applications.

Application-specific page replacement was also suggested in the context of K42 [2], where per-region Page Manager objects potentially allowed for a variety of replacement schemes. A replacement policy based upon communication with the application is a natural extension of such an approach.

2.3 Collecting Memory Utilization Data

Early research required that MRCs be gathered offline by repeatedly re-running an application using varying memory sizes [1, 11, 24, 25]. Memory traces for MRC generation can also be generated through simulation or through instrumentation of memory accesses. Run-time collection of MRC data was first proposed by Zhou [33]. Both a hardware and a software approach were suggested, with the proposed hardware being implemented in a simulator. Since then we have seen several pure-software [15, 32] approaches to gathering MRC data, as well as a hardware-software hybrid [3].

The approach in both hardware and software requires updating an LRU stack for each memory access and applying Mattson's Stack Algorithm, described in Section 4.3. Tracking of page accesses is necessary for the generation of the LRU stack. To track page accesses, Zhou's original hardware implementation [33] snoops on the memory bus. On each access, a hardware LRU stack and MRC are updated. This pure hardware approach is efficient with respect to time , but storing the LRU stack requires 2MB of storage for each 1 GB of memory. The hardware is decidedly single-purpose as only the complete MRC is exposed outside of the hardware.

Zhou's software approach uses memory protection to track page accesses. Since handling protection faults is a relatively expensive operation, the pages are divided into an active and inactive set. The active set contains recently accessed pages and is not page protected, instead relying on regular scanning of the page access bits. Pages in

the inactive set are protected in memory, causing a protection fault on each access. In this manner, protection faults are only triggered in the uncommon case, greatly reducing overhead while slightly reducing the accuracy of the LRU stack. LRU stacks and MRCs are maintained by the OS, with bins used to reduce the overhead of determining stack position. Overhead estimates are 7-10%

CRAMM [32] improves the efficiency of the software approach with two primary optimizations. The active and inactive sets are dynamically re-sized, allowing the page protection fault overhead to be reduced. In addition, an AVL tree is layered over the LRU stack, reducing the overhead on each update. Between these two techniques, overhead is reduced to 1-2.5%. However, it has been argued that to get such a low overhead, the accuracy must be reduced beyond the realm of usefulness for many proposed applications of MRCs [3].

PATH [3] combines the software and hardware approaches. The LRU stack and MRC is maintained by the OS in a manner similar to Zhou's software approach, while specialized hardware was proposed to track page accesses. Overhead is reduced by filtering out repeated accesses to hot pages, with a slight loss of accuracy at the top of the LRU stack, and by buffering page accesses so that the OS is infrequently interrupted. Overhead is conservatively estimated to be roughly 6% and the generated MRCs are believed to be more accurate than those from efficient software-only approaches. Because the proposed hardware exposes page access information to the OS, other memory management optimizations are possible (see 2.2.2, 2.2.3 above) beyond those facilitated by MRC data. BACH [9] also proposes specialized hardware for generating memory traces, but for the purpose of trace-driven analysis, and this scheme has been shown to be constrained by the cost of writing to disk [31].

Qureshi [21] proposes a full hardware approach for generating MRCs at the hardware cache level. However, this approach does not scale up to main memory. Monitoring a 4 MB 16-way cache with 64 byte lines requires maintaining LRU ordering for the 16

lines in each of 4096 sets, whereas monitoring 4GB of main memory requires a single LRU stack containing over a million entries. Since the cache already maintains LRU information within each of its sets, for the cache, the LRU stack information is effectively free, whereas maintaining a large LRU stack is computationally expensive. Qureshi reduces the overhead further by only monitoring a subset of the cache sets.

Lu [15] uses a modified hypervisor to generate MRCs without modifying the operating system. The hypervisor maintains a large page cache. Virtual machines are provided with artificially low memory allocations; however, since pages are "swapped" in from the hypervisor cache, rather than disk, the penalty is relatively low. The overhead is low and proportional to the hypervisor cache size. This approach only produces miss rate data for larger memory allocations. Granularity is also reduced, as MRCs represent the entire virtual machine, rather than the individual process, which may be an advantage or disadvantage depending upon the intended use for MRC data.

2.4 Existing MRC Knowledge

Recent work on MRCs has focused on performance improvement rather than workload characterization. As such, much of the qualitative data about MRCs has gone unpublished. MRC examples are sometimes included for purposes of illustration, but the focus is generally on quantitative evaluation of the performance of MRC-based techniques rather than the MRCs themselves.

A number of limitations exist in the existing studies. In many cases, total system memory is quite low [32, 33] with memory pressure applied synthetically [32], and as such the benchmarks used are not necessarily representative of real-world memory-constrained applications. In some studies, particularly those relying on simulation for validation, the benchmark runs are very short, evaluating only a short time-slice of the process' lifetime [3], or using only small, short-lived benchmarks [32].

The properties of MRCs are not generally agreed upon. For instance, it has been reported that they are usually convex [33] which seems to be supported by some results [15, 32], while the MRCs published elsewhere show otherwise [3, 21]. This is likely a reflection of both the choice of benchmarks and the granularity at which MRC data is collected.

To expand our knowledge of the characteristics of MRCs in real applications, we require a low-overhead, high-accuracy method of MRC generation with available hardware. In Chapter 3 we describe the hardware data sampling mechanisms that may achieve this goal.

Chapter 3

Hardware Data Sampling

Performance Monitoring Units (PMUs) and Hardware Performance Counters (HPCs) have become an important component of all modern CPUs. A basic overview of PMUs is presented in Section 3.1. Section 3.2 describes the specifics of Intel's Precise Event-Based Sampling (PEBS) facilities. Related work using PEBS is described in Section 3.3. Finally, the limitations and advantages of PEBS for the problem of tracking memory accesses and generating a MRC is described in Section 3.4.

3.1 Standard Facilities

On modern CPUs, a limited set of hardware counters is provided for counting performance events. The type of events to be counted and the manner in which they are counted is programmed via a set of control registers.

Typically, processors allow the counting of various events related to normal and abnormal execution, such as CPU cycles, instruction counts (sometimes for specific instruction types as well), branch mispredictions, and a range of memory-related events such as cache and DTLB misses. Frequently there are limitations on how events may be counted: the total number of counters is usually quite small, often less than 10; events are often constrained in terms of which counters can be used to count them or which other events may

be counted simultaneously. Some of these limitations may be addressed through HPC Multiplexing [4].

HPCs can be read on demand or in response to timer or overflow interrupts. HPCs may also be used in conjunction with overflow interrupts to trigger the monitoring of other aspects of the machine state. However, frequent overflow interrupts may dramatically degrade performance as application execution is halted for overflow interrupt handling. In the next section, we see how PEBS addresses this problem.

3.2 Precise Event-Based Sampling (PEBS)

PEBS is an Intel facility for reducing the overhead of overflow-triggered sampling of the processor state; it was introduced with the Netburst microarchitecture. When an HPC is used with PEBS, the overflow interrupt is handled by hardware. On a counter overflow, the complete register state is saved into a buffer and the counter is reset. Execution is only interrupted when the buffer overflows, reducing overhead.

PEBS records a PEBS record, containing the architectural registers and state information into a programmer-configured PEBS buffer in memory. While in practice the PEBS record format is fixed, Intel dedicates 5 bits to specifying what is to be stored in a PEBS record, suggesting that other recording functionality could hypothetically be added. The format of a PEBS record is shown in Figure 3.1 on page 17.

PEBS is limited to tracking one performance-monitoring event at a time, as PEBS works only with a single counter. PEBS is also very limited in terms of which performance-monitoring events can be used. Table 3.1 on page 18 shows the performance-monitoring events that work with PEBS on the Intel Core microarchitecture.

These events are counted during instruction retirement. For example, a cache or DTLB miss will not be counted until the instruction that was waiting on the memory operation is completed and retired. On the Netburst microarchitecture, a PEBS record

Status Register: Maintains status flags used by conditional branch instructions.
Instruction Pointer: Maintains location in program execution.
Result Register: Used for returning results.
Preserved Register: Value preserved across calls.
Parameter Register: Used for passing of parameter values.

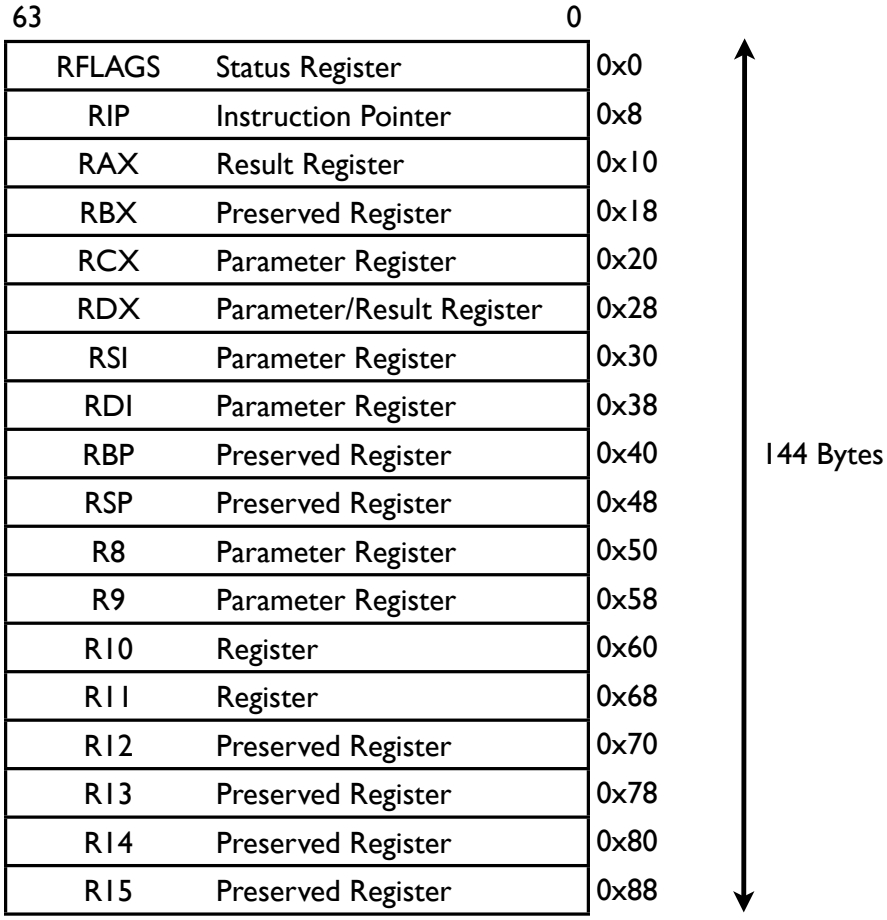


Figure 3.1: 64-Bit PEBS Record Format [8]

Event Name	Description
INSTR_RETIRED.ANY_P	Any retired instruction
X87_OPS_RETIRED.ANY	Any retired floating-point instruction
BR_INST_RETIRED.MISPRED	Any retired branch instruction
SIMD_INST_RETIRED.ANY	Any retired vector instruction
MEM_LOAD_RETIRED.L1D_MISS	Any retired instruction that waited on an L1 cache miss
MEM_LOAD_RETIRED.L1D_LINE_MISS	Any retired instruction that waited on an L1 cache miss and caused the cache line to be requested
MEM_LOAD_RETIRED.L2_MISS	Any retired instruction that waited on an L2 cache miss
MEM_LOAD_RETIRED.L2_LINE_MISS	Any retired instruction that waited on an L2 cache miss and caused the cache line to be requested
MEM_LOAD_RETIRED.DTLB_MISS	Any retired instruction that waited on a data TLB miss

Table 3.1: PEBS Performance Events for Intel Core microarchitecture [8]

captures the CPU state at the onset of the instruction that caused the counter overflow. On the Core/Core2 microarchitecture, a PEBS record captures the state immediately before the next instruction.

3.3 PEBS Memory Applications

PEBS has found a number of applications. This section describes other applications of PEBS for gathering memory addresses. In Section 3.4 the unique elements of our work are discussed.

The PMPT (Performance Monitoring PEBS Tool) [5] integrates the PEBS-generated memory addresses with information collected from malloc() to identify data structures that are contributing to poor cache performance. L2 cache misses are sampled, the data addresses are calculated using methods similar to this work, and this data is cross-referenced against records from a modified version of malloc to identify the data structures

responsible for the L2 cache misses.

TOPP (Trace-based Optimization for Precomputation and Prefetching) [23] uses PEBS-generated memory traces for prefetching. L2 cache misses are sampled using PEBS and pattern recognition techniques are used to perform temporal-locality-based prefetching by generating "prefetch slices" that are executed in idle thread contexts.

Vivek Thakkar [30] uses PEBS-generated memory traces for determining page affinity and performing page migration on ccNUMA architectures. Sampled L2 cache misses are used to generate a trace of page accesses for each node, and pages are then migrated to the nodes that are using them most.

Schneider et al. [26] use PEBS to sample cache misses. They do not calculate the memory addresses, unlike the other works described here. Instead, they use the instruction pointers to optimize the JIT compilation of Java bytecodes.

3.4 PEBS For MRCs

This section highlights some of the unique aspects of our research as well as identifying an alternative methods of MRC generation that we did not attempt.

While most memory-related PEBS work samples L2 misses, we felt that DTLB misses were a better fit for gathering information about pages. The redundancy of having up to 64 L2 misses for each page might alleviate the issue of data loss, described in Section 6.3; however, that redundancy would also drive up overheads as well. The increased overhead could be alleviated by reducing sampling frequencies, but this would increase the risk of systematic data loss due to striding access patterns that only caused a small number of L2 misses on each visited page.

Another unique aspect of our work is the sampling frequencies used. We set PEBS to attempt to capture every single DTLB miss. We know of no other work that pushes PEBS to its limits in this way, and the behaviour of PEBS under these conditions is

undocumented.

Memory traces might also be generated on hardware with a software-managed TLB by instrumentation of the TLB miss handler. This would produce similar data to our approach, without suffering from the data loss problem mentioned above and detailed later. However, the TLB miss handling code is a critical performance path, and adding increased overhead to this code could be incredibly costly overall. PEBS gives us the same data with lower overheads, but with some data loss.

More details of PEBS-based MRC generation are found in Chapter 4 and Chapter 5 with the details of the PEBS configuration in Section 5.1. The success of this approach is detailed in Chapter 6.

Chapter 4

Generating MRCs

In an ideal world, our hardware could provide a ready-to-use MRC as in Qureshi[21] and Zhou[33]'s proposals, or a Memory Trace as in PATH[3]. The goal of this work, however, was to use existing real-world hardware, specifically PEBS, which was not designed specifically for this purpose.

This section describes at a high level how PEBS may be used to generate an MRC at run-time. Section 4.1 describes how to track memory accesses using PEBS. Section 4.2 describes how to extract the memory addresses into a Memory Trace. Section 4.3 describes Mattson's Stack Algorithm [22], which is an algorithm for building a MRC from a Memory Trace. Chapter 5 describes the implementation details for each of these steps.

4.1 Tracking Memory Accesses

There are two limitations of PEBS when it comes to tracking memory accesses. First, due to the limited events that PEBS can monitor, we are forced to use either cache or TLB misses as a proxy for memory accesses. Second, PEBS is a mechanism for coarse-grained sampling, so it was not intended to record an entry for every instance of an event.

While memory accesses is not one of the supported performance events for PEBS on the Intel Core/Core2 microprocessor, it is possible to monitor misses in the L1(data) or

L2 caches or DTLB misses. For our purposes, we can view the caches and the DTLB as acting as a series of filters, removing repeated accesses to recently accessed, or "hot", pages from our stream of memory accesses. At each successive level, more memory accesses are filtered out. This may initially seem to be a problem, but is in fact of benefit to us. It will become clear in Section 4.3 when Mattson's Stack Algorithm is discussed that the impact of these "hot" pages on the MRC is significant primarily for very small memory allocations, while monitoring these hot pages adds both time and space overhead. In fact, the PATH hardware proposal [3] successfully uses additional hardware filters to reduce the size of its trace buffer. We show in Section 6.5 that DTLB misses can be used to produce MRCs with the same characteristics as a true MRC based on memory accesses.

In this work, we attempt to track each memory access that results in a DTLB miss. It is not our belief that PEBS was intended to be used in this manner. PEBS seems to be intended as a sampling mechanism. Most PEBS applications and examples that we have seen use sampling frequencies around 1/100,000 and the behaviour of PEBS at high sampling frequencies is undocumented. In this work, we attempt to push the sampling frequency to 1/1. As we will show, some data loss occurs if a 1/1 sampling frequency is used, and at the outset it was not clear to what extent this would present a problem for our approach. Section 6.3 explores the relationship between data loss and sampling frequencies in PEBS.

4.2 Extracting Memory Addresses

Recall that PEBS provides in its output a copy of the register state. This state does not explicitly contain the address that was used to access memory. However, it is possible to calculate the address from the saved registers. Using the instruction pointer from the PEBS output to search the assembly code of the monitored application, the instruction corresponding to the memory access may be found. Once the instruction's assembly

is available, standard x86 addressing is simulated using the register values provided by PEBS.

An additional complication is that on the Intel Core/Core2 microarchitecture (which we use in our experiments) PEBS provides the register state at the outset of the instruction immediately following the instruction of interest. This complicates the calculation of the true instruction pointer that caused the DTLB miss. Due to x86's variable-length instructions, the instruction pointer of interest cannot be calculated without the application assembly code, but instead must be found by moving backwards in the assembly. Additionally, in the event that the instruction of interest overwrites one of the registers used in the address calculation, it becomes impossible to correctly determine the correct address[1].

The results of processing the PEBS records in this manner is a Memory Trace, which can be used to produce an MRC as explained in Section 4.3.

4.3 Mattson's Stack Algorithm

Mattson's Stack Algorithm [22] is an algorithm for generating a Miss Rate Curve from an access trace. It was developed by Mattson in 1970 and first applied to memory by Kim in 1991 [13].

The algorithm maintains an LRU stack and a set of counters with one counter for each entry in the LRU stack. For each memory access, the counter corresponding to its current distance from the top of the stack is incremented and the page is then moved to the top of the stack to maintain the LRU ordering.

For example, on an access to page 326, the algorithm finds the entry in the stack corresponding to page 326. Suppose the page is found 27 items into the stack, then counter 27 is incremented, and page 326 is moved to the top of the stack.

[1] This situation could be identified by comparing the source and destination registers of the instruction. However, this complicates the implementation and was not implemented in this work.

To generate an MRC, the number of page faults for a given memory allocation is calculated by summing all the counters corresponding to stack distances greater than the number of pages in the given allocation. The expected number of page faults with n pages can be expressed as:

$$faults_n = \sum_{i=n+1}^{N} counter_i,\ where\, N\, is\, the\, maximum\, counter\, number$$

In the real world, searching and modifying a stack on each memory access is time consuming. Various optimizations exist [3, 29, 32, 33]. Customarily stack items are grouped into blocks so that it is possible to quickly find the item using hashing or another lookup scheme and then quickly calculate the stack distance based on which block the item is in. These optimizations were not implemented in this work; however, the existing work in this area makes it clear that efficient implementations of Mattson's Stack Algorithm exist.

Chapter 5 details the implementation decisions that were made in generating MRCs using PEBS.

Chapter 5

Implementation

This chapter describes the implementation details of our system to generate MRCs using PEBS. Section 5.1 describes the use of PEBS to monitor DTLB misses on a modern Linux kernel using Perfmon2 and the libpfm library. Section 5.2 describes the use of objdump to calculate the memory address from a PEBS entry and the use of this information to generate an MRC.

The specific tools used in this system are not requisite for this task. While our implementation is Linux-specific (running in our case on a 2.6.29 kernel), in principle any operating system running on a PEBS-capable processor could be used. The only true requirements are a PEBS-capable processor and access to application binaries.

Figure 5.1 on page 26 provides an overview of our system that may prove useful in understanding this chapter.

5.1 Tracking DTLB Misses Using PEBS

The libpfm library provides a library for user-level access to the perfmon [7] interface, a proposed extension to the Linux kernel providing platform-independent system calls for setup and access to the system's hardware performance monitoring unit. In our experience, this combination is a versatile and usable means of programming performance

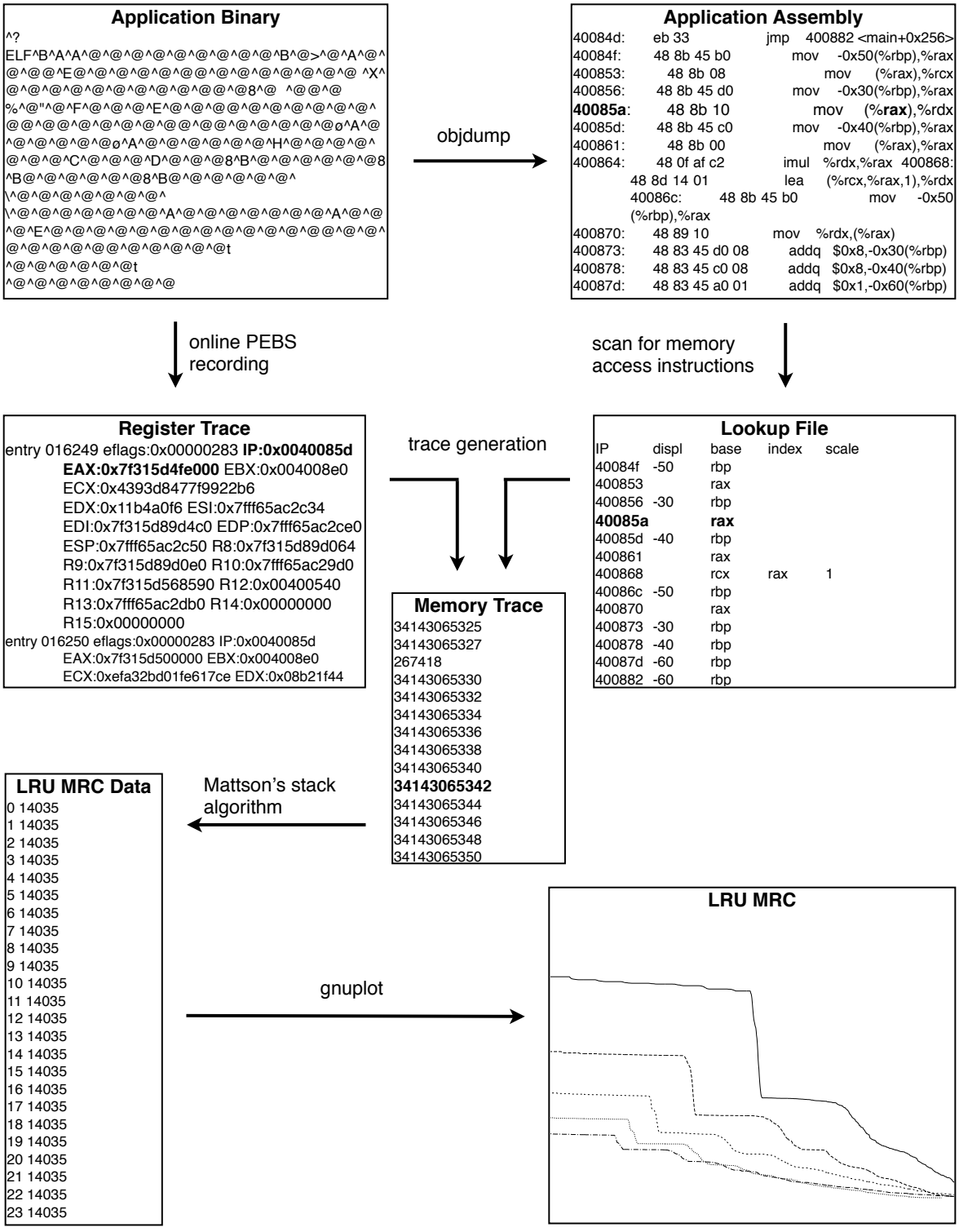


Figure 5.1: System Overview

counters on a Linux system. Of particular value for this project was the built-in PEBS support, which includes mapping the PEBS buffer into user space for fast and efficient processing.

Most of the complexity of using PEBS is handled by libpfm. We enabled PEBS and set our sample event to "MEM_LOAD_RETIRED:DTLB_MISS". Our PEBS sample period was passed in as a command line variable. Setting up the PEBS buffer and mapping it into user space is handled by libpfm, however it is up to the user to process the buffer entries. We simply dump the PEBS entries to a regular file, referred to in Figure 5.1 on page 26 as the Register Trace. This, however, is not particularly efficient, and an implementation intended for use in online optimization would want to process the PEBS entries while still in main memory.

Recall from Section 3.2 that each PEBS entry contains the register state of the processor immediately following the event being sampled. In the Register Trace, we have a record of the register state immediately following each sampled DTLB miss.

5.2 Calculating Memory Addresses from PEBS Entries and Generating MRCs

In order to convert our register state log into a trace of memory addresses, it is necessary to calculate the memory address for each recorded DTLB miss. This section describes how that is done.

The simplest, but not necessarily the most efficient means of viewing the application assembly is using the objdump utility. For each of our benchmarks, we disassemble the application binary using objdump. The entire application is scanned for instructions that access memory, which are easily identified by looking at the addressing mode used. The instruction pointers as well as the registers and immediate values used in the address calculation are recorded to the Lookup File, for later use. In an online implementation, an

in-memory hash-table would dramatically improve lookup time, but its implementation would be orthogonal to the challenge of determining the quality of data available with PEBS.

Next, the Register Trace, described in Section 5.1, is scanned. For each entry, the Lookup File is scanned for the relevant instruction. Each line of the Register Trace contains the instruction pointer of the subsequent instruction to the instruction that accessed memory and caused the DTLB fault. As such we must find the closest instruction preceding the instruction pointer in the Register File. The register values stored in the registers listed in the Lookup File are extracted from the Register Trace and combined with the immediate values stored in the Lookup File to perform an address calculation, following standard x86 addressing rules as follows:

$$Address = Base + (Index \times Scale) + Displacement$$

The resulting memory addresses are then divided by 4096 to give us page numbers which are appended to the Memory Trace in Figure 5.1 on page 26. This Memory Trace is fed into a standard unoptimized implementation of Mattson's Stack Algorithm, as described in Section 4.3 to produce an MRC.

5.2.1 Example

This section provides an example of the process described above. Looking back to Figure 5.1 on page 26, the entries in the Application Assembly, Register Trace, Lookup File, and Memory Trace corresponding to this example are all in bold.

Our tool scans the Application Assembly to identify instructions that address memory. In our example, observe that the instruction address 40085a uses register rax. This information is recorded in the Lookup File. While scanning through the Register Trace, we see an entry with IP: 40085d. The instruction pointers in the Register Trace point

to the following instruction, so we look in the Lookup File for the previous instruction, which is 40085a. The value of rax (which is actually called EAX in the Register Trace) is then extracted from the Register Trace. The address calculation in this case is trivial and the page number is stored in the Address Trace, which is then used as input to generate the MRC.

Chapter 6

Evaluation

This chapter describes our experiments to determine how well our approach works for generating MRCs. Section 6.1 describes our test platform. Section 6.2 describes our experiments on three small hand-written benchmarks. Section 6.3 explores the number of DTLB misses not captured by PEBS for varying sampling frequencies. Section 6.4 evaluates the overhead of high-frequency sampling of DTLB misses using PEBS. Section 6.6 shows a number of interesting MRCs generated during these experiments and discusses some of their properties. Section 6.5 attempts to determine the accuracy of our generated MRCs and their utility for online optimizations.

6.1 Test Platform and Benchmarks

The experiments described below were performed on a server with dual Intel Xeon X5355 quad-core Core2 CPUs and 8 GB of RAM. All benchmarks were run using only a single core, however, in order to minimize non-determinism. We used a Linux 2.6.29 kernel, patched to include the Perfmon2 interface.

All benchmarks used were from the PARSEC 2.1 benchmark suite. PARSEC was chosen because it is free, similar to the SPLASH benchmark suite used in evaluating PATH, and it seemed to be better supported on x86. Not all PARSEC benchmarks were

blackscholes	Option pricing with Black-Scholes Partial Differential Equation (PDE)
bodytrack	Body tracking of a person
canneal	Simulated cache-aware annealing to optimize routing cost of a chip design
facesim	Simulates the motions of a human face
ferret	Content similarity search server
fluidanimate	Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method
freqmine	Frequent itemset mining
streamcluster	Online clustering of an input stream
swaptions	Pricing of a portfolio of swaptions
x264	H.264 video encoding

Table 6.1: PARSEC Benchmarks [18]

used due to incompatibility with our test system[1]. The following benchmarks were used: blackscholes, bodytrack, canneal, facesim, ferret, fluidanimate, freqmine, streamcluster, swaptions, and x264. Swaptions, however, did not produce enough DTLB missed to generate accurate MRCs using our approach. Table 6.1 on page 31 shows descriptions from the PARSEC website of the benchmarks used in our experiments. Each of the PARSEC benchmarks comes with six datasets, from smallest to largest: test, simdev, simsmall, simmedium, simlarge, native. We did not use the test dataset for any of our benchmarks. We used multiple datasets for each benchmark, so as to demonstrate how the MRC varies with dataset size, but did not use every dataset with each benchmark, as some were too small to produce good MRCs and others were too large to run during our experiments.

6.2 Micro-Benchmarks

We wrote three small micro-benchmarks to test our approach and help demonstrate some basic MRC properties. Knowing the MRCs for these three known access patterns helps

[1]The raytrace, vips, and dedup benchmarks did not build correctly on our test system.

in making informed estimates about the behaviour of our PARSEC MRCs in Section 6.6.

The Figures in this section show the PEBS-generated MRCs as well as a "Real" MRC generated by instrumenting every single memory access using Intel's PIN dynamic instrumentation tool [16] and applying Mattson's Stack Algorithm to the resulting memory trace. The "Real" MRC is accurate under the assumption of true LRU replacement with no prefetching or caching, so it may not correspond to true application behaviour.

The rand micro-benchmark randomly increments entries in a 4 MB (1024 page) array of 64-bit integers. In Figure 6.1 on page 33 we see an almost perfectly linear MRC. This is to be expected given that the probability of a a randomly accessed page being in memory is proportionate to the amount of memory given to the application, so with 0 pages, we expect to miss on every access, with 512 pages we expect to miss on half of the memory accesses, and with 1024 pages we expect 0 misses, as the entire application fits within memory.

Our "Real" MRC has an identical shape to the PEBS MRCs; however, its magnitude is much greater. This is likely due to a combination of factors. Our processor's DTLB contains 256 entries, so there is a 25% chance of any given access being hidden from us by the DTLB. More significantly, our benchmark should be expected to generate almost 10 million DTLB misses while running for roughly half a second, and the PEBS hardware is not capable of capturing 20 million records per second. We note, however, that even when PEBS captures less than 0.1% of the DTLB misses with a PEBS sampling frequency of 1/5, the MRC shape is still correct for this benchmark.

The bigarray micro-benchmark sequentially increments entries in a 4 MB (1024 page) array of 64-bit integers, looping through the array 25 times. This benchmark is shown in Figure 6.2 on page 34. Looking at the "Real" case, notice that the MRC has a perfectly flat plateau until the entire benchmark fits within memory. This reflects the poor performance of LRU for sequential looping access patterns, as the least recently accessed pages is the page that will be reused next.

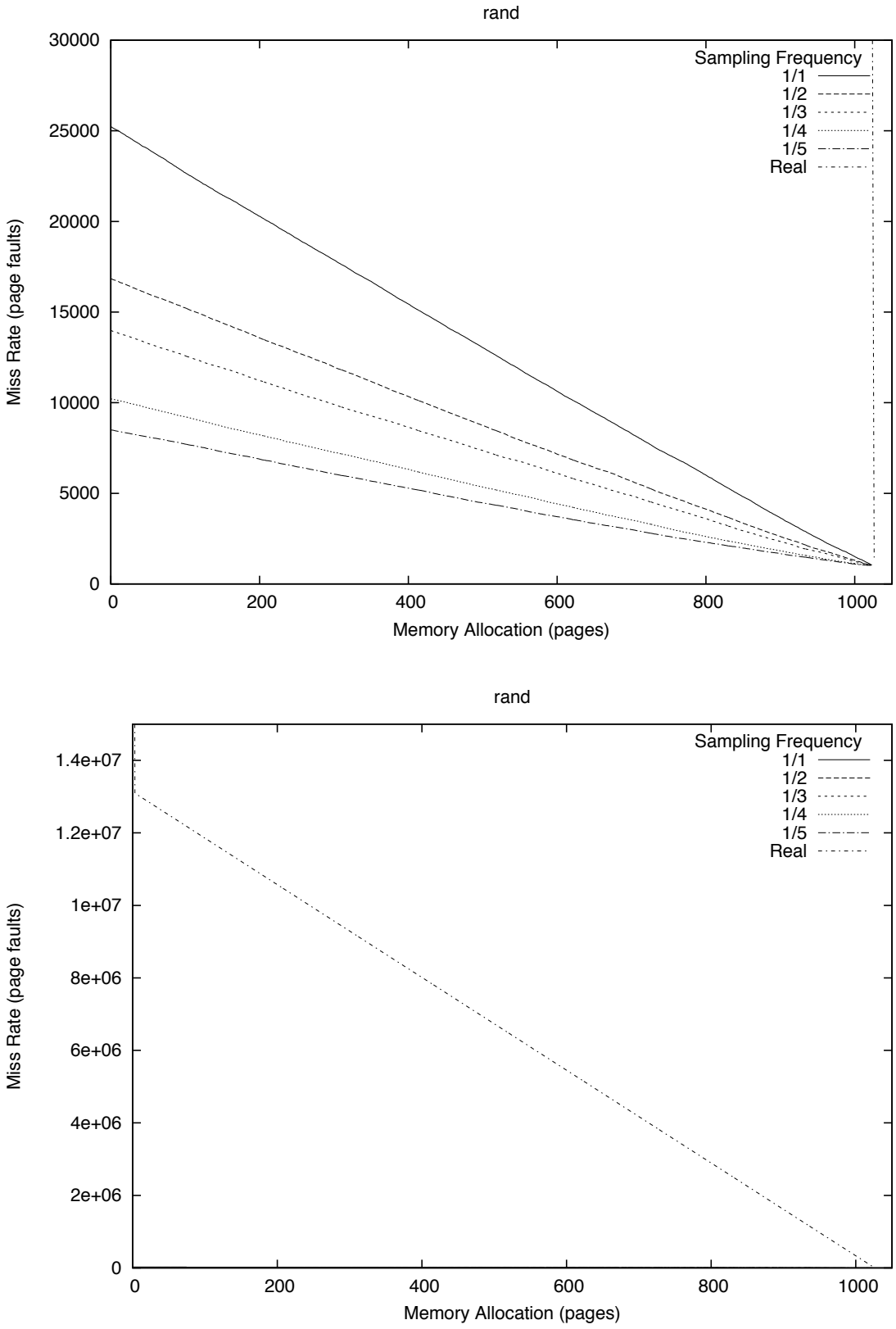


Figure 6.1: Random Access Micro-Benchmark

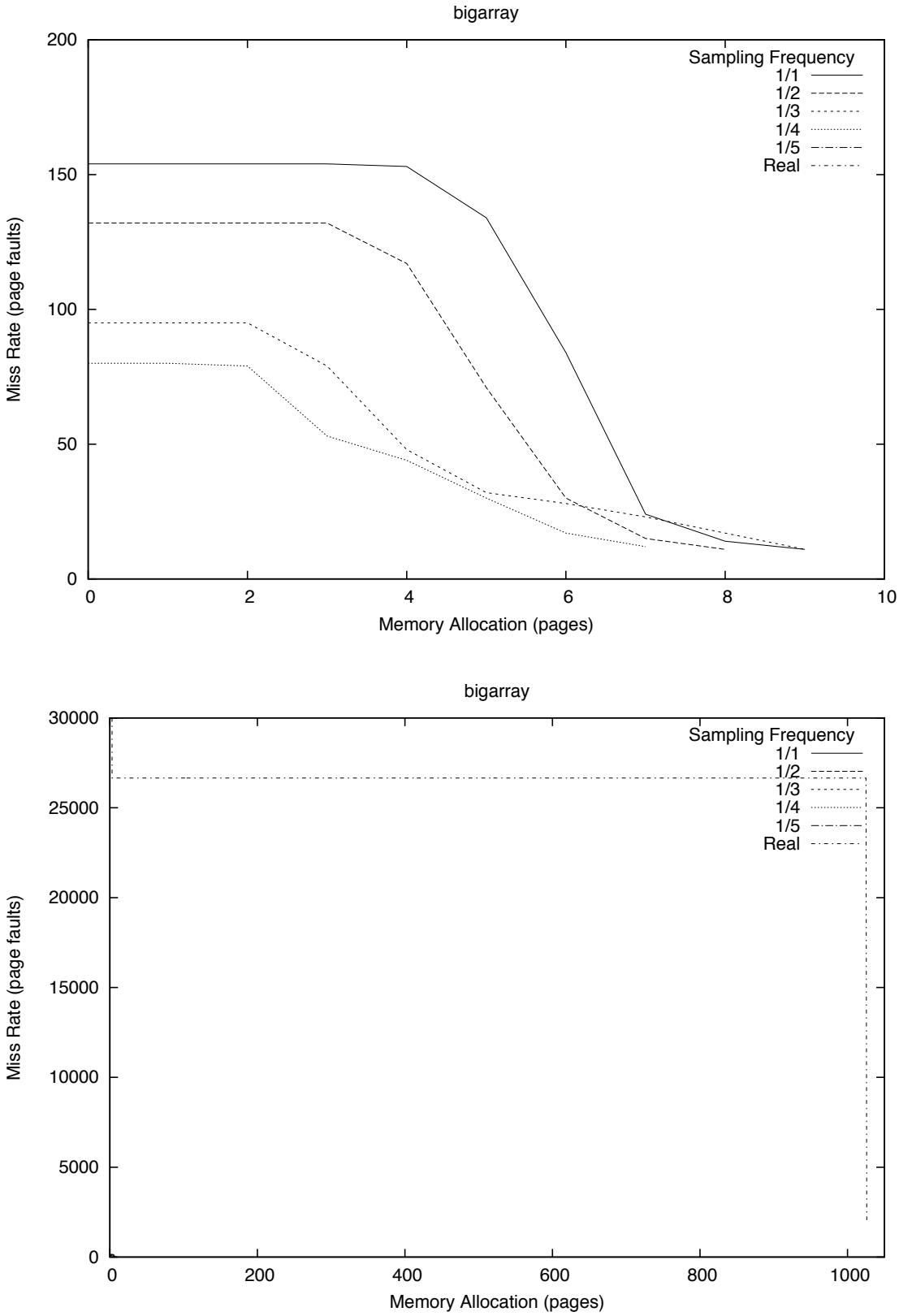


Figure 6.2: Sequential Access Micro-Benchmark

While a looping access pattern is terrible for LRU-based page replacement, a looping access pattern is ideal for both temporal-locality-based prefetching and physical-locality-based prefetching, and our PEBS results reflect this. It seems that our test system is able to prefetch almost every page access, avoiding DTLB misses. In this case, there is reason to believe that our PEBS results may be more reflective of real-world performance than the "Real" MRC.

The randmatmult micro-benchmark, shown in Figure 6.3 on page 36 multiplies a randomly-generated 8 by 65536 matrix with a randomly generated 65536 entry vector. The data structures are 4MB (1024 pages) for the matrix, 512 KB (128 pages) for the vector, and 64 bytes (less than 1 page) for the result vector. Because our algorithm makes alternating accesses to the matrix and the vector, for smaller memory allocations, the matrix and vector each occupy half of the available memory. In the "Real" MRC, it is not until the benchmark has just over 256 pages that the 128 page vector fits entirely within memory, causing a significant reduction in page faults. At just over 1152 pages (1024 + 128) the entire matrix and vector both fit in main memory, and there is another reduction in page faults. For this reason, we see 2 plateaus. The first is when neither data structure is entirely within memory, and the second is when both data structures are entirely within memory[2].

For the PEBS MRCs, the vector's pages are contained almost entirely in the Core2's 256 entry DTLB, as are 128 of the matrices pages, so accesses to these pages are hidden from us. As a result, we see only a single plateau. If PEBS were to capture every single DTLB miss, we would expect the plateau to extend to 896 pages, but instead it drops off just below 600, a result of data loss for this memory-intensive benchmark.

[2]This demonstrates one benefit to defining separate regions for memory management within a process. If the vector and matrix regions were independent regions, a memory allocator could see significant performance improvements between ~128 and ~256 pages by giving all of the available memory to the vector.

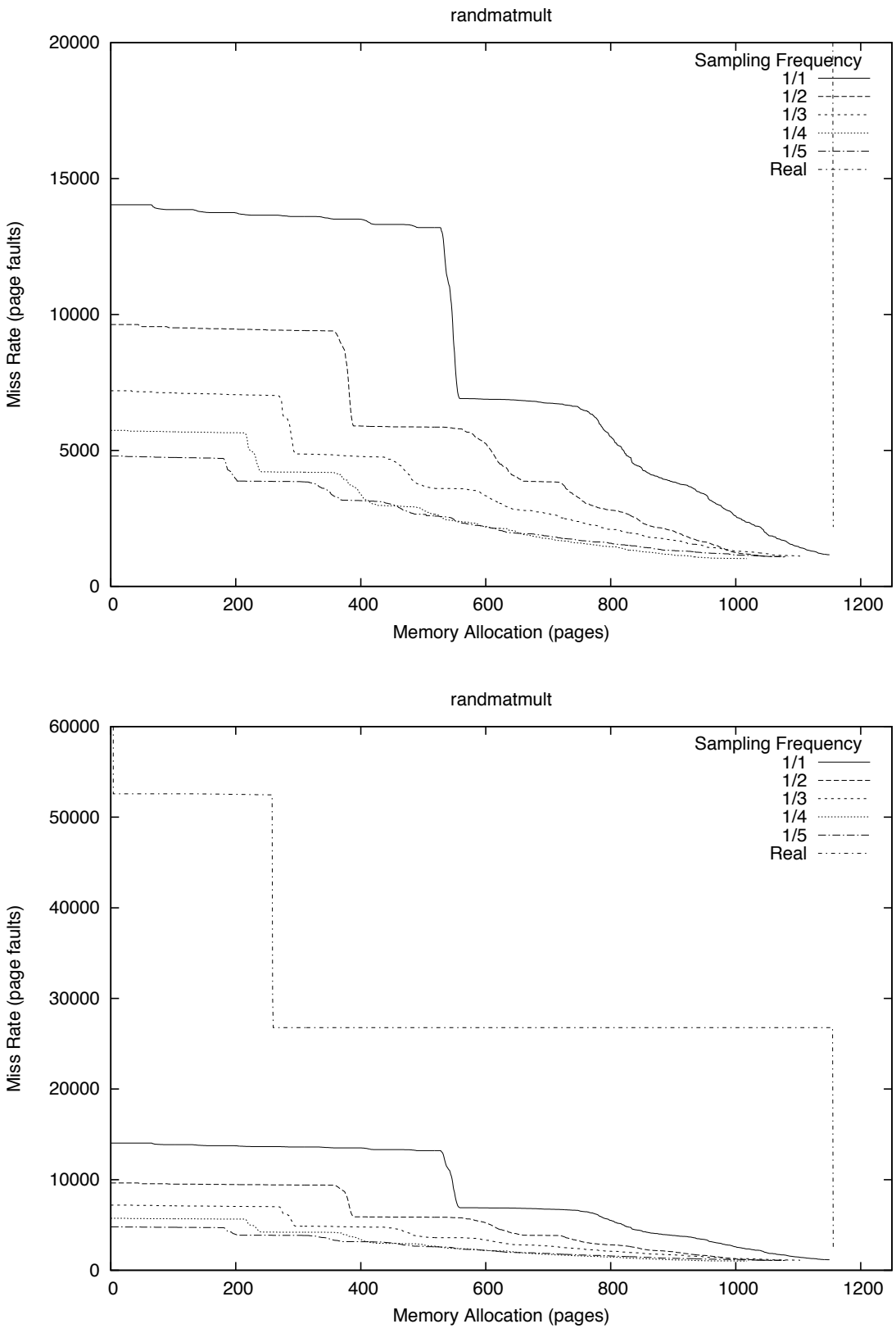


Figure 6.3: Matrix Multiplication Micro-Benchmark

6.3 PEBS Data Loss and Reset Values

In Figure 6.4 on page 38, Figure 6.5 on page 39, Figure 6.6 on page 40, Figure 6.7 on page 41, and Figure 6.8 on page 42 the number of recorded PEBS records for three runs of each benchmark at each sampling frequency are plotted against the sampling frequency used[3]. The dotted line represents the expected number of PEBS records, and is determined by doing a linear extrapolation on the points corresponding to lower sampling frequencies, specifically 1/10 and lower. For higher sampling frequencies, we see the measured PEBS records diverge downwards from this line and this gap represents our rate of data loss. Data losses were up to 50% of all DTLB misses. While these losses affect the accuracy of the memory trace, in Section 6.5 we see that these losses do not appear to affect the overall shape of the MRC.

We see considerable instability on the smaller benchmarks and datasets, as the number of PEBS entries collected is not large enough for smaller deviations to average out. For instance, swaptions produces very few PEBS entries with any dataset size, and as such shows large deviations. However, our larger benchmarks are quite stable. Our freqmine results seem to have systematic instabilities, as multiple runs produced significantly lower numbers of PEBS entries than expected. We believe that some aspect of freqmine's access pattern sometimes causes PEBS to malfunction and stop recording for a period of time. This may be due to an overflow of the reset counter or PEBS buffer, but this is currently just speculation.

6.4 Overhead due to PEBS

Our implementation make excessive use of text files, which is unnecessary and reduces overall performance. In order to isolate the impact of PEBS-based monitoring of DTLB

[3]Sampling frequency refers to the frequency at which PEBS samples DTLB misses. A sampling frequency of 1/10 indicates that PEBS is set to provide a register sample on every 10^{th} DTLB miss. A sampling frequency of 1 indicates that PEBS is attempting to sample every single DTLB miss.

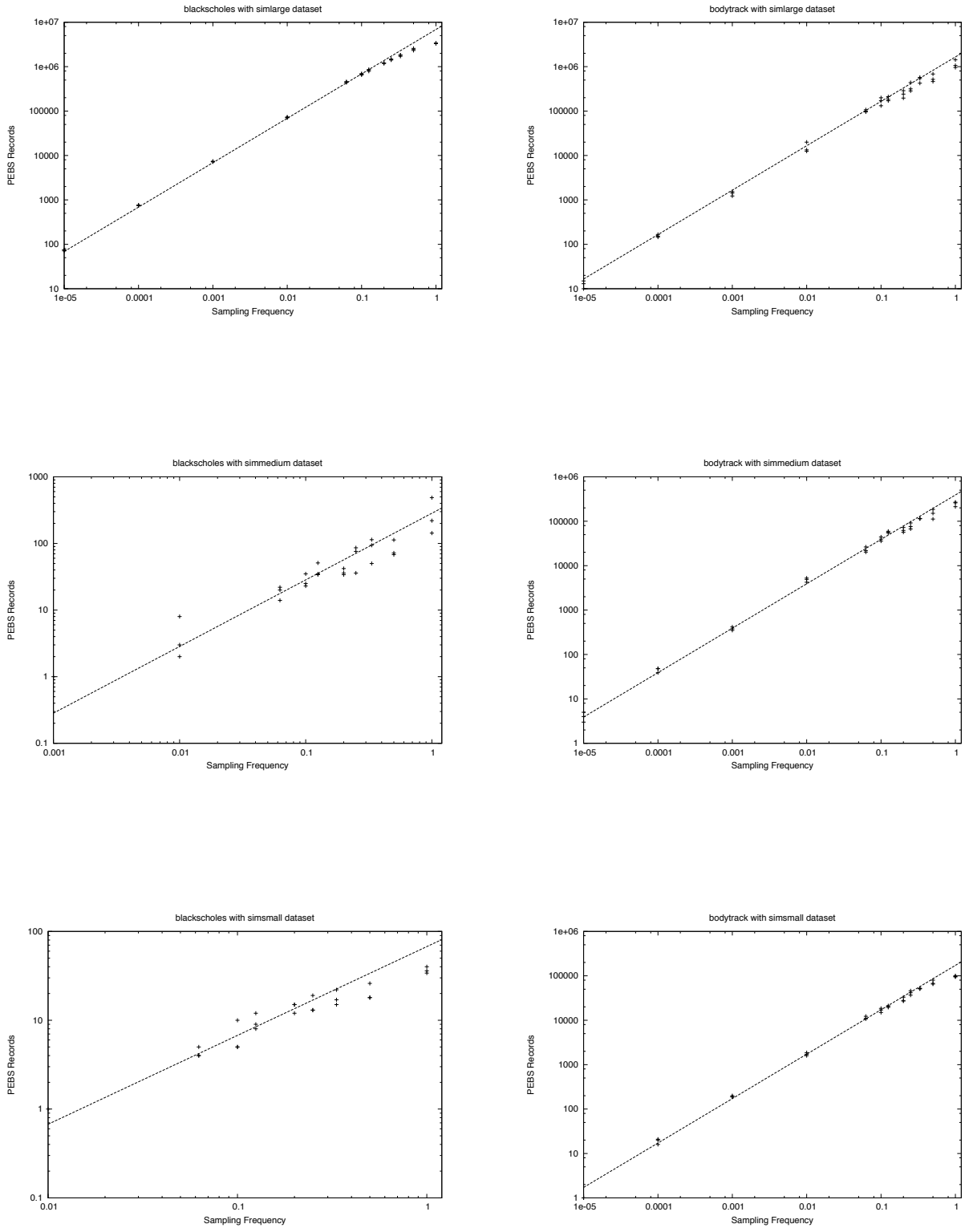


Figure 6.4: Data Loss for the Blackscholes and Bodytrack Benchmarks

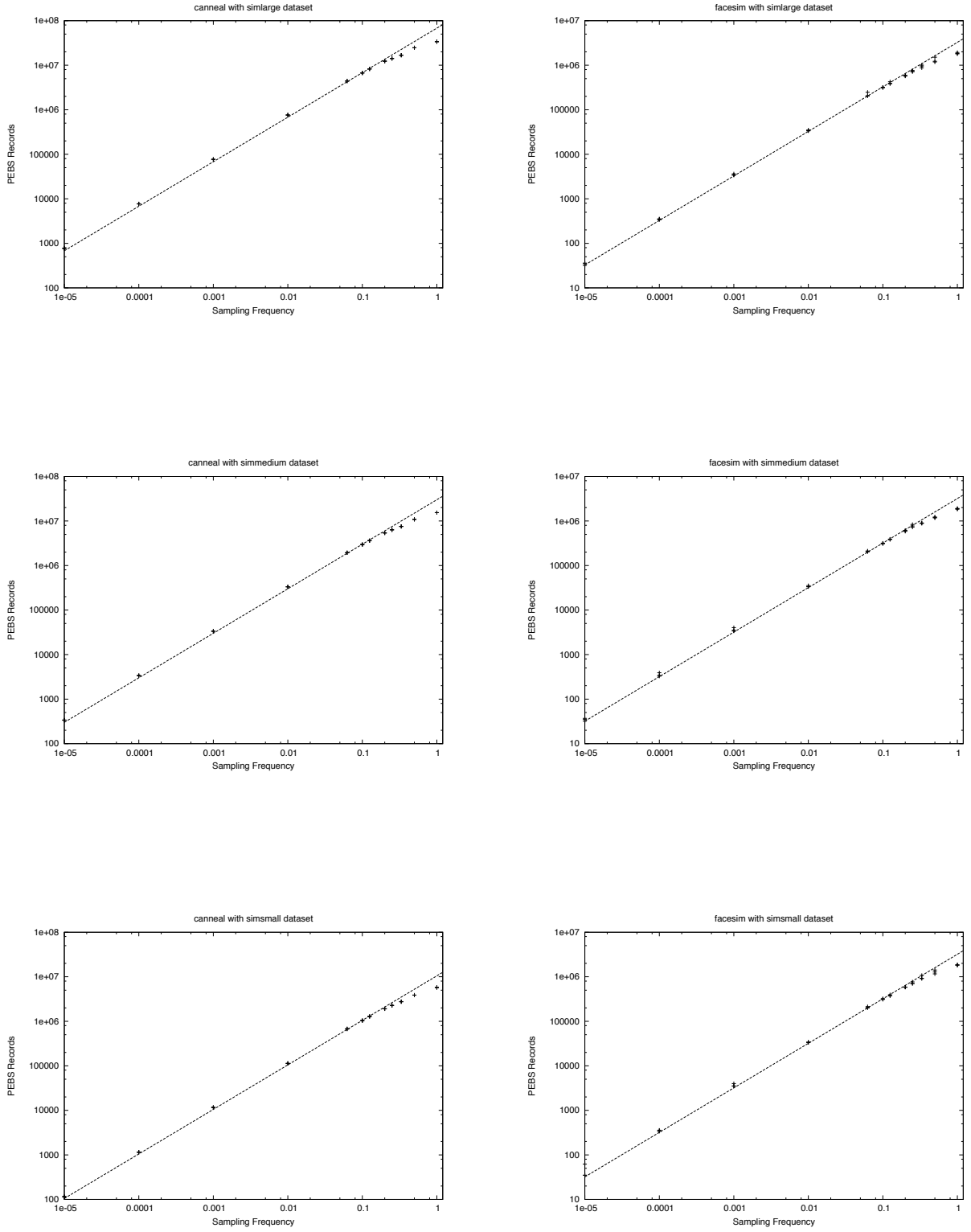


Figure 6.5: Data Loss for the Canneal and Facesim Benchmarks

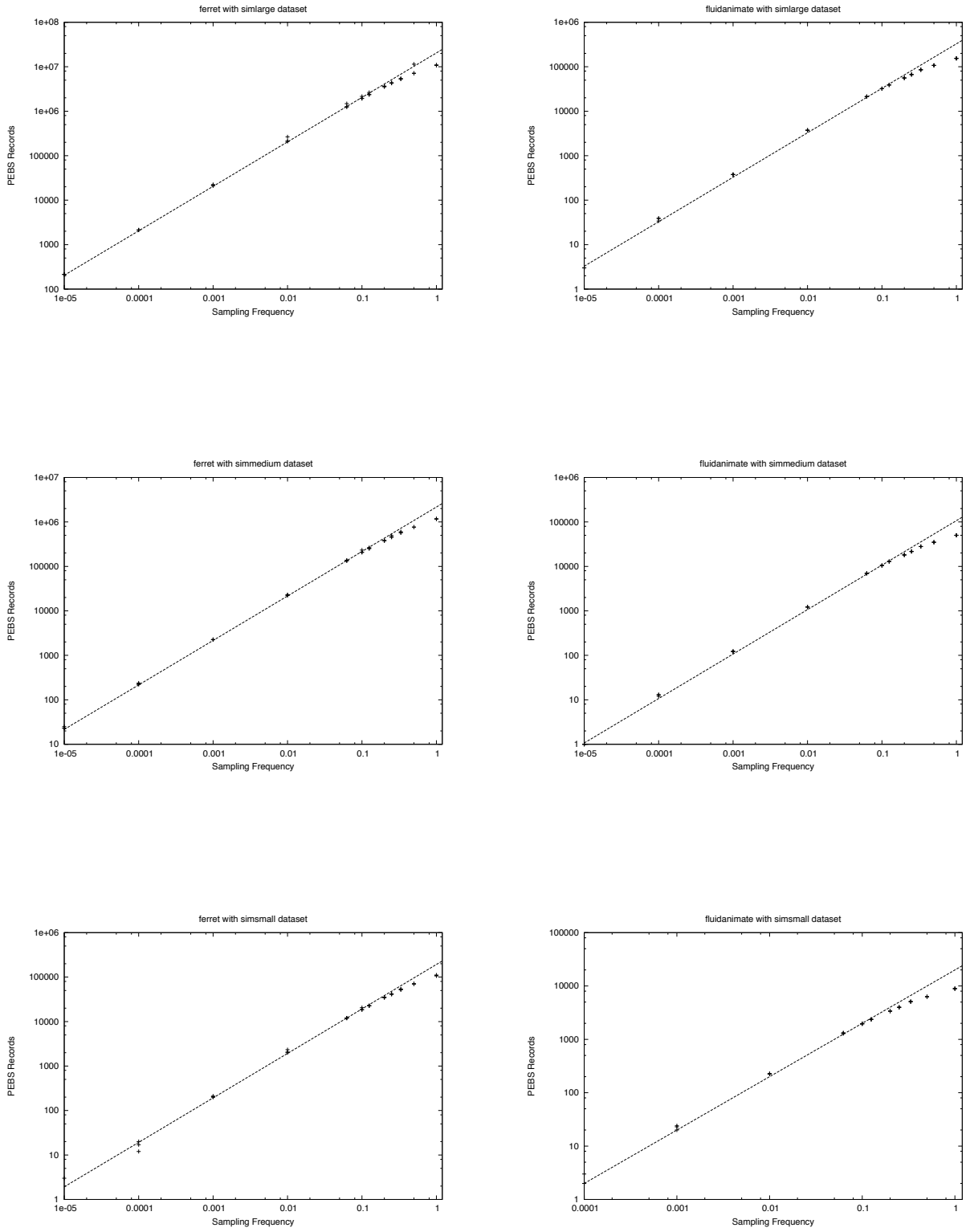


Figure 6.6: Data Loss for the Ferret and Fluidanimate Benchmarks

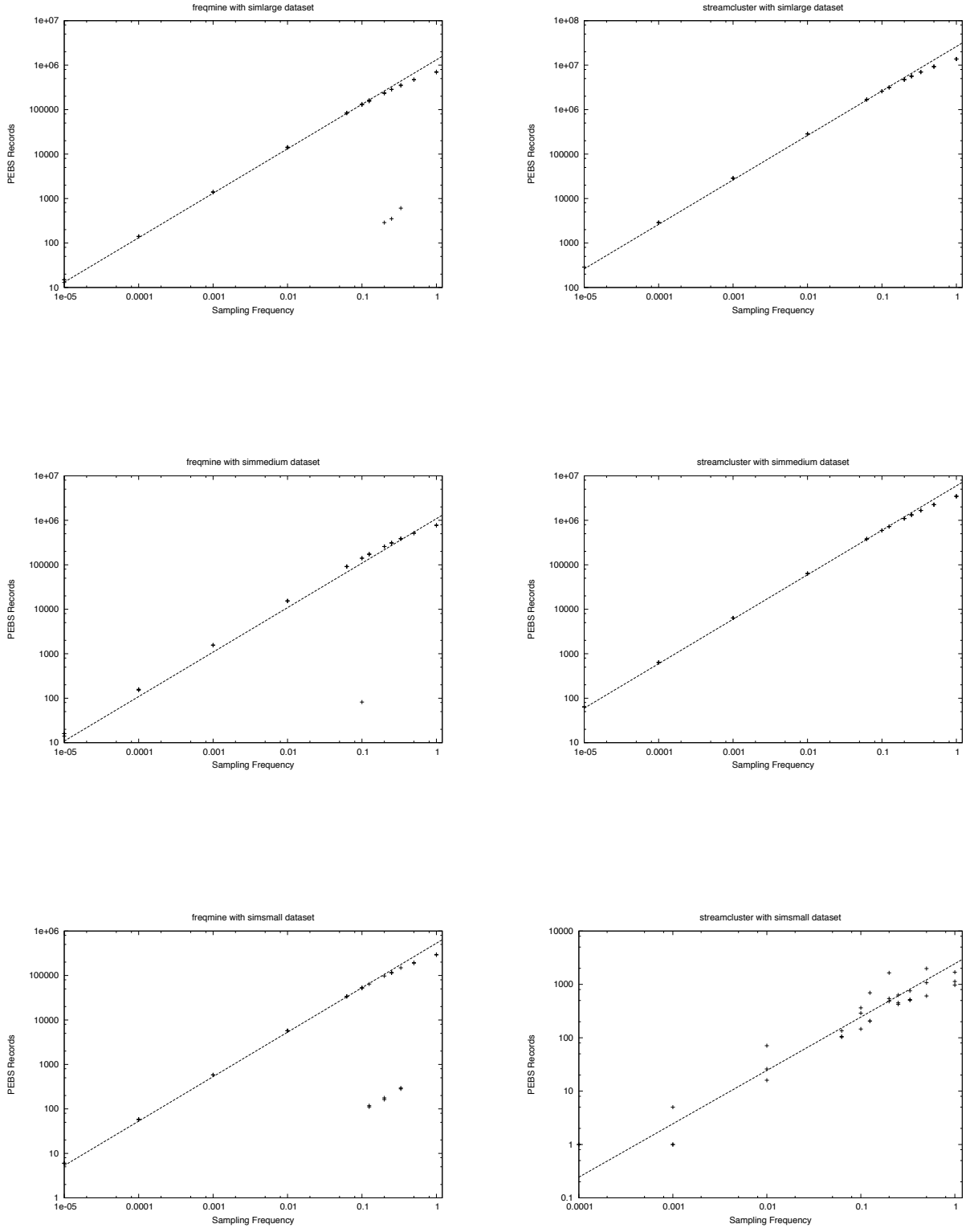


Figure 6.7: Data Loss for the Freqmine and Streamcluster Benchmarks

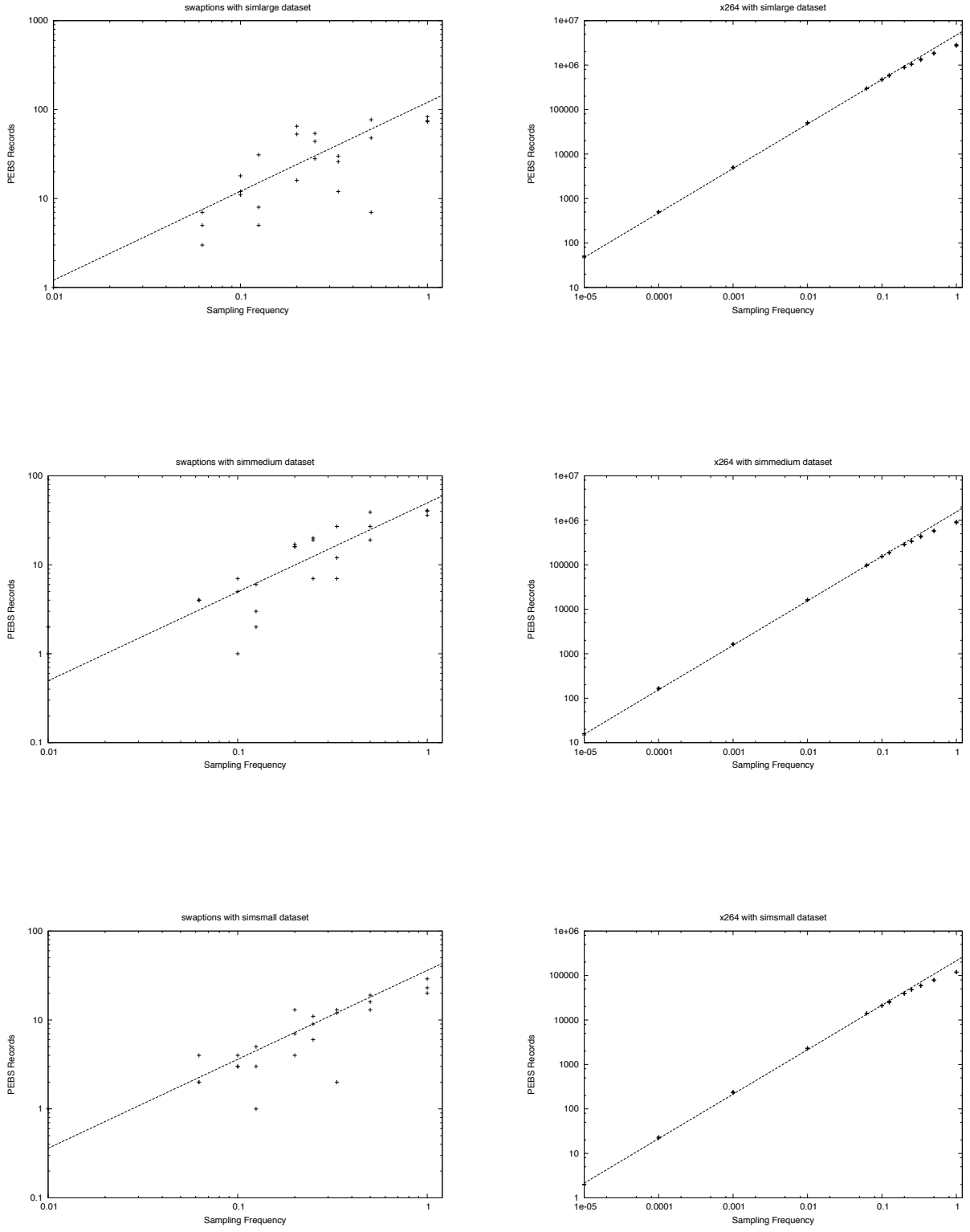


Figure 6.8: Data Loss for the Swaptions and X264 Benchmarks

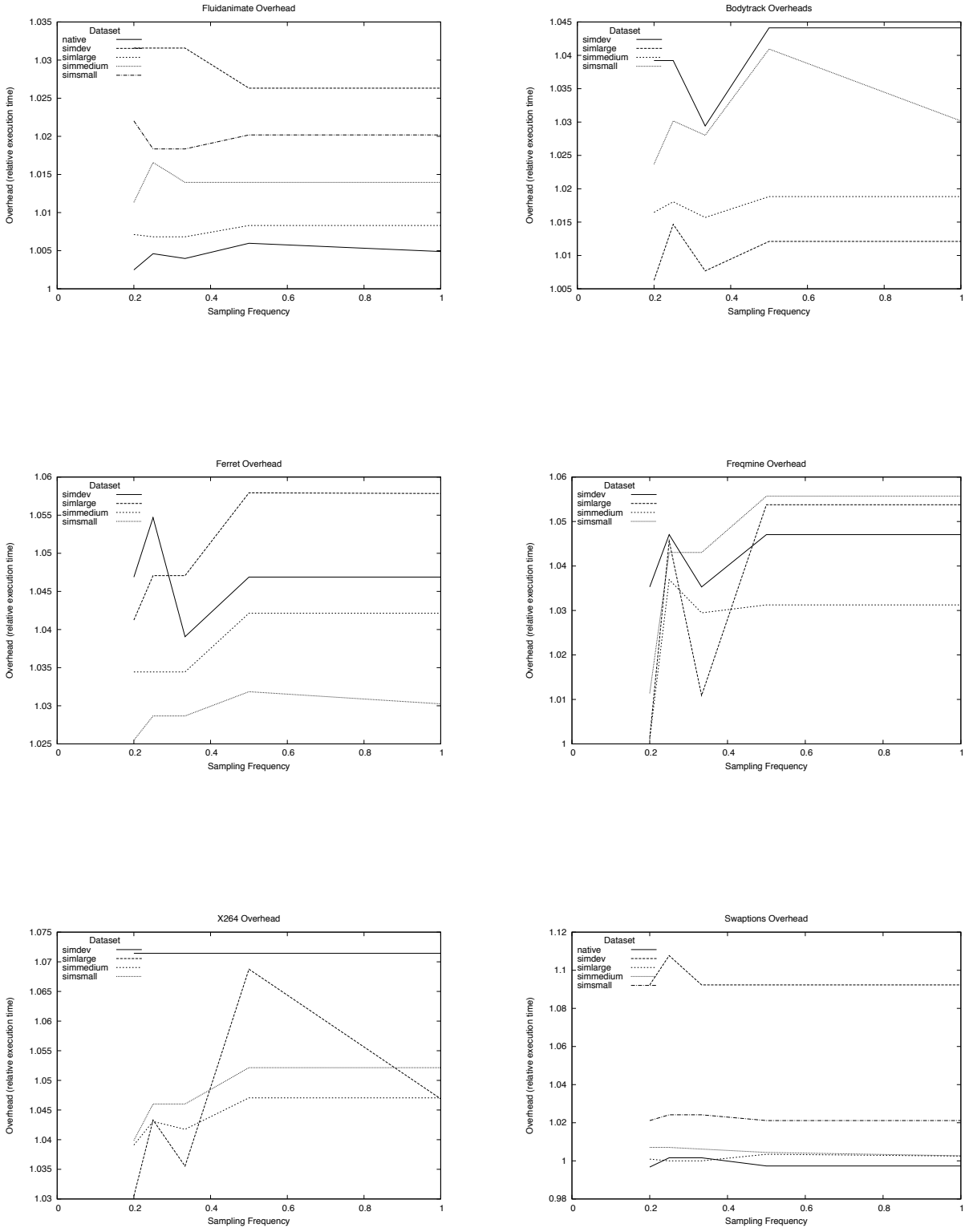


Figure 6.9: PEBS Overheads (1 of 2)

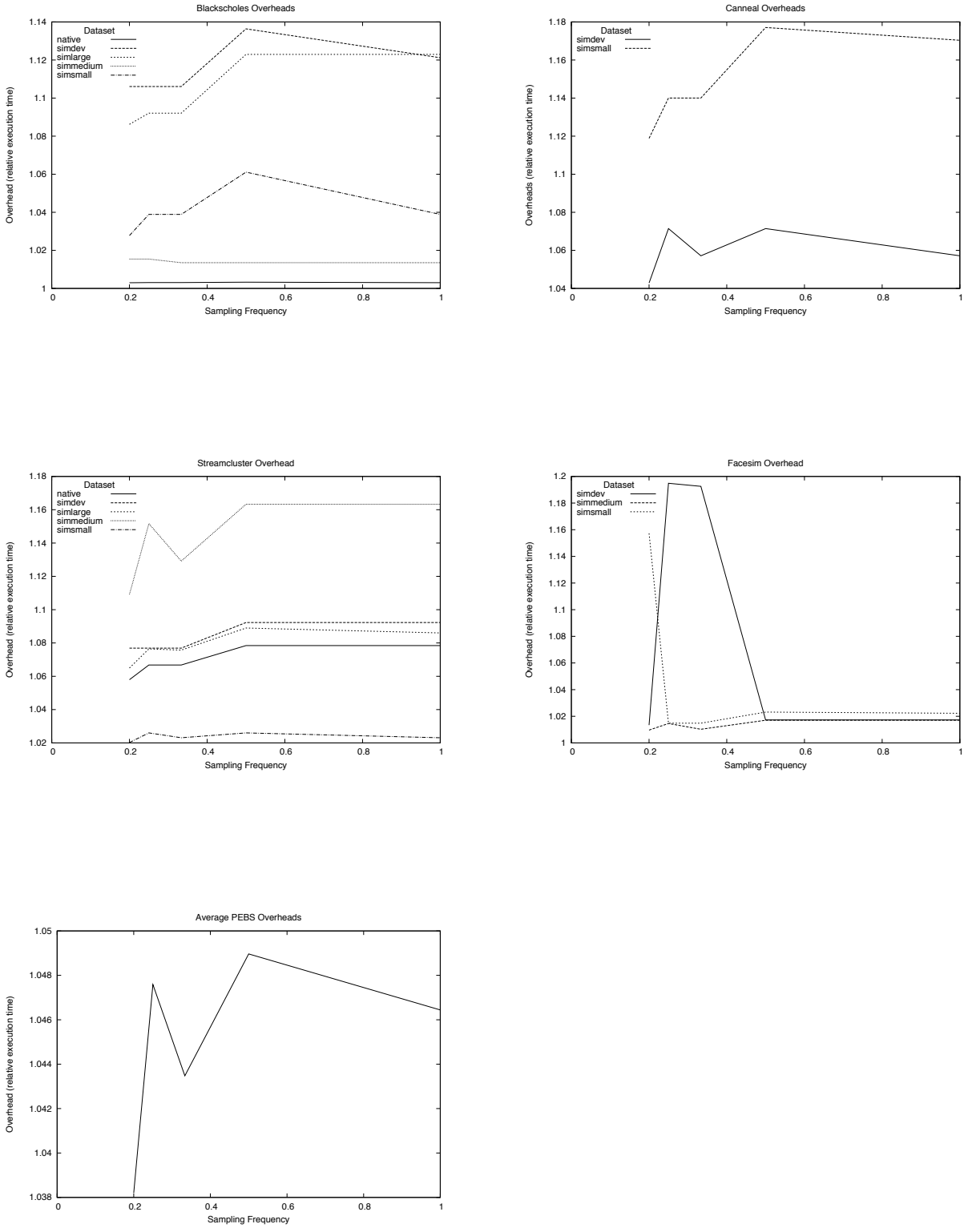


Figure 6.10: PEBS Overhead (2 of 2)

misses from implementation details and the overhead of our unoptimized Mattson's stack algorithm, we ran our PEBS-based monitoring code with file output turned off. The overhead due to PEBS initialization, PEBS-based monitoring, and scanning the PEBS record buffer all remain. Figure 6.9 on page 43 and Figure 6.10 on page 44 shows how the PEBS overhead varies in relation to the sampling frequency. Each data point on these graphs shows the minimum run time of 3 PEBS runs divided by the minimum run time of 3 runs without PEBS.

All of our benchmarks ran with less than 20% overhead, with an average overhead of less than 5%. There did not seem to be a consistent connection between overhead and dataset size. As might be expected, the benchmarks that produced the most PEBS entries (as seen in the figures from Section 6.3) had the highest overheads.

The overhead could likely be reduced further by using techniques developed in RapidMRC [29]. RapidMRC detects changes in MPKI (Misses Per thousand(K) completed Instructions), indicating an application phase[4] change. An MRC is generated at the beginning of each phase. During MRC generation, RapidMRC has overheads of over 300%; however, overall overhead is reduced below 2%. By only using PEBS after a detected phase change, we believe our overhead could be reduced as well.

6.5 Accuracy

Figure 6.11 on page 46, Figure 6.12 on page 47, and Figure 6.13 on page 48 show PEBS-based MRCs for sampling frequencies from 1/5 to 1/1 alongside an MRC generated by using the PIN [16] dynamic instrumentation tool to record all memory accesses and produce an accurate memory address trace.

[4]A phase is a section of a program's execution in which the application's behaviour is relatively consistent, such as when an application is repeatedly performing similar actions. The greater the length of an application phase, the more resources can be used to optimize the program's behaviour in the given phase without the cost outweighing the performance benefits. A standard method of phase detection is looking for changes in the application's MPKI (Missed Per thousand(K) Instructions).

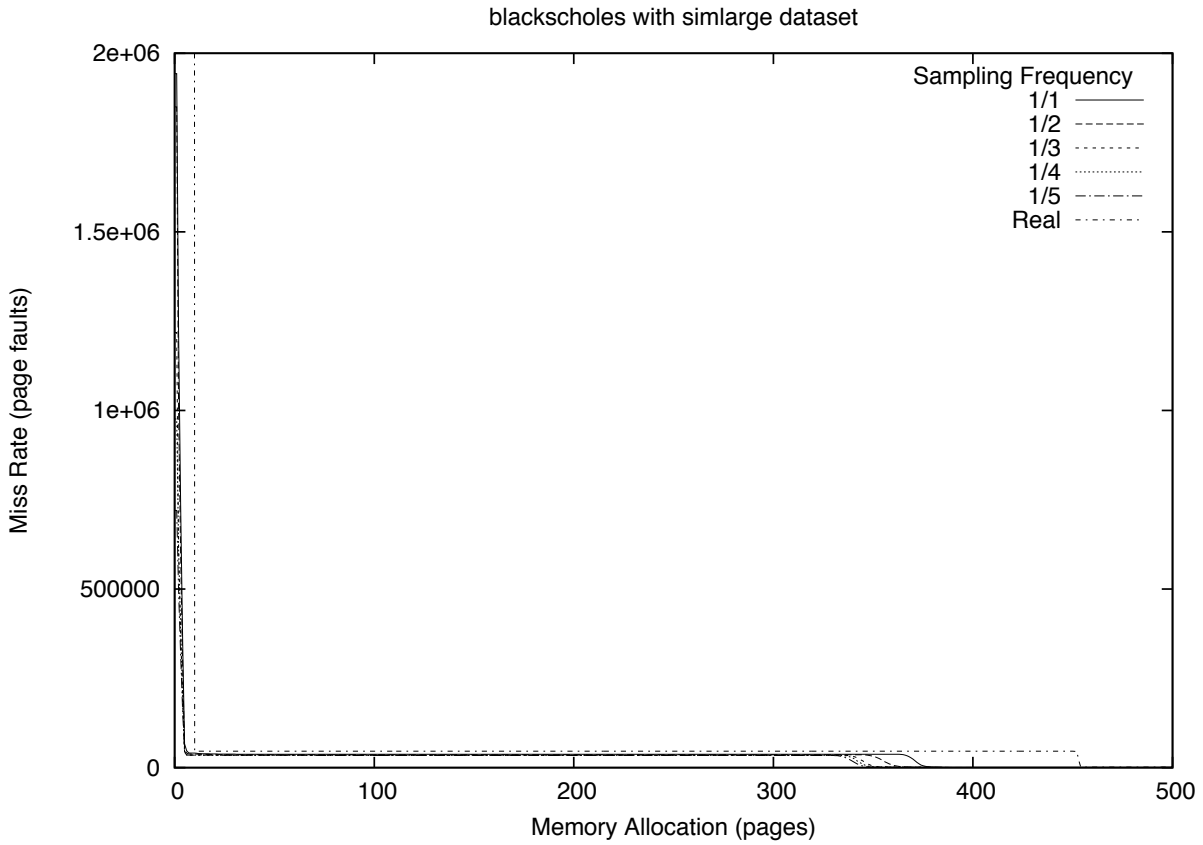


Figure 6.11: Blackscholes MRC showing the PEBS and Real MRCs having similar cliffs. Blackscholes with PEBS did not produce enough data for comparison for smaller datasets.

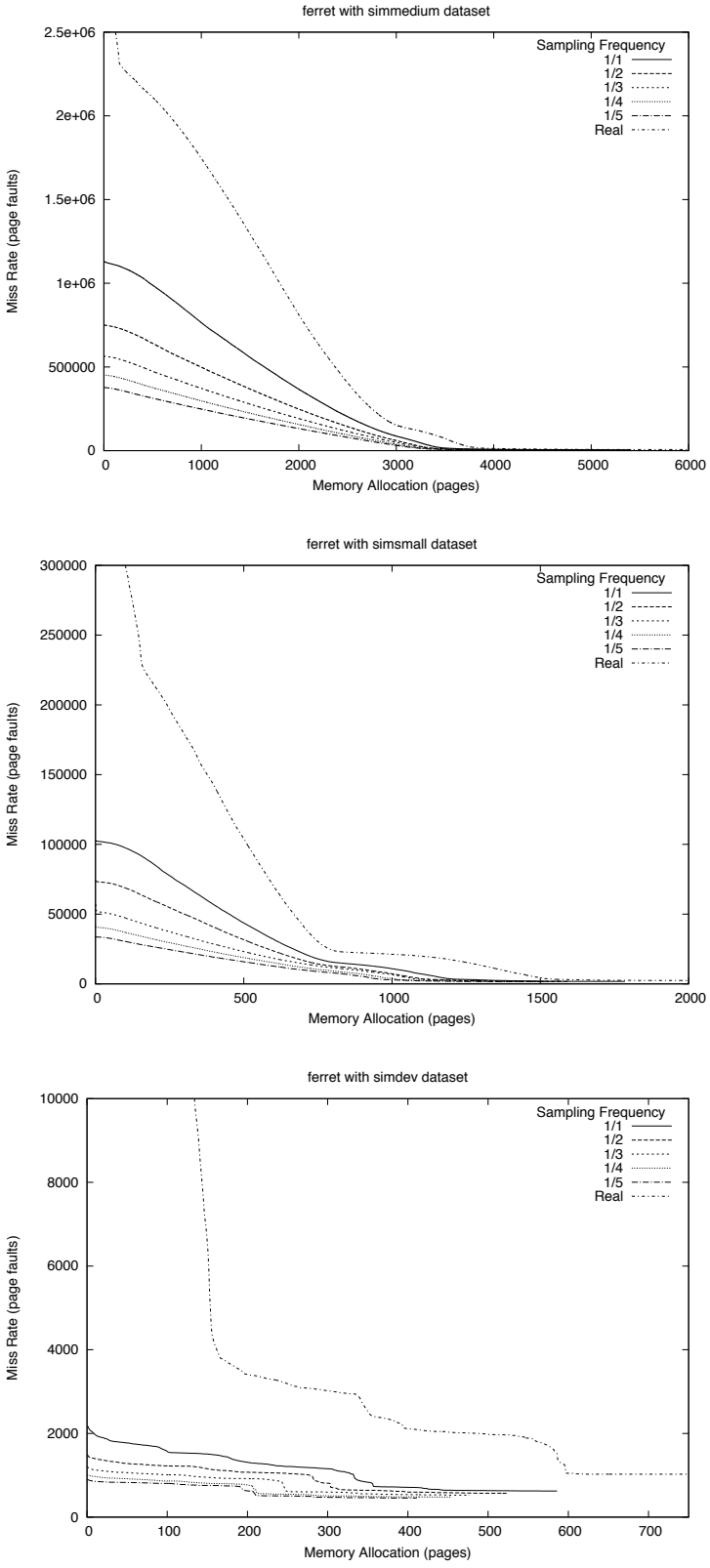


Figure 6.12: Ferret MRCs showing how PEBS accuracy degrades for smaller memory sizes.

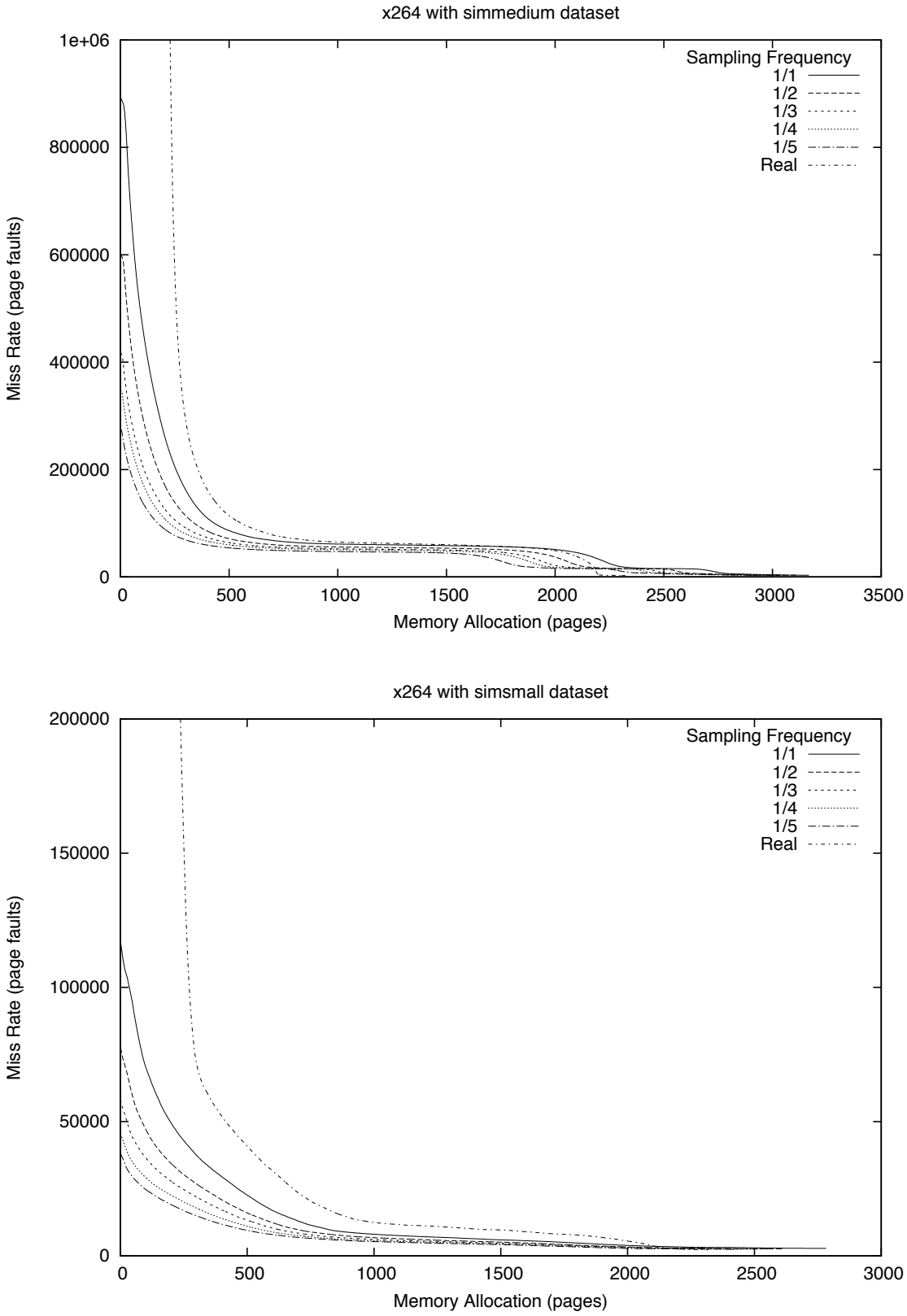


Figure 6.13: X264 MRCs showing similarity in shape between PEBS MRCs and Real MRCs.

The PIN-produced "Real" MRC curves have the same shape as the PEBS-produced curves, although they diverge for very small memory sizes. This is expected and reflects the impact of the DTLB hiding frequently-accessed "hot" pages from us.

For both MRC-based memory allocation and LRU stack-based page replacement, the limited accuracy for small memory sizes should not be of major concern. Memory allocation decisions are based on the segment of the MRC close to a process' current memory allocation, so the accuracy for very small and unrealistic memory allocations is not a concern. Page replacement algorithms identify infrequently accessed pages, so inaccurate ordering of "hot" pages at the top of the stack are of little concern as well.

6.6 MRCs

This section shows all of our PEBS-generated MRCs, showing the diversity of MRCs and the challenges that MRC-based algorithms face.

Figure 6.14 on page 50 shows the blackscholes benchmark, which simulates option pricing using the Black-Scholes Partial Differential Equation. It includes a very steep cliff followed by a long plateau. The large upper plateau for the native dataset is interesting. This plateau is likely a sign of a large frequently accessed data structure whose pages fit within the TLB for the smaller datasets, but which is sufficiently large on the native dataset as to overflow the TLB and influence our MRC. The initial plateau presents a problem for greedy memory allocators. Looking at the native dataset, the marginal utility of additional pages is close to zero below 25,000 pages, hiding the performance gain to be had by giving the process 40,000 pages or more.

The freqmine benchmark, shown in Figure 6.15 on page 51, does frequent item-set mining. It also shows a sharp drop-off followed by a plateau, but the lack of an initial plateau makes it less dangerous to a greedy allocator.

Figure 6.16 on page 52 shows the bodytrack benchmark, a human body tracking sim-

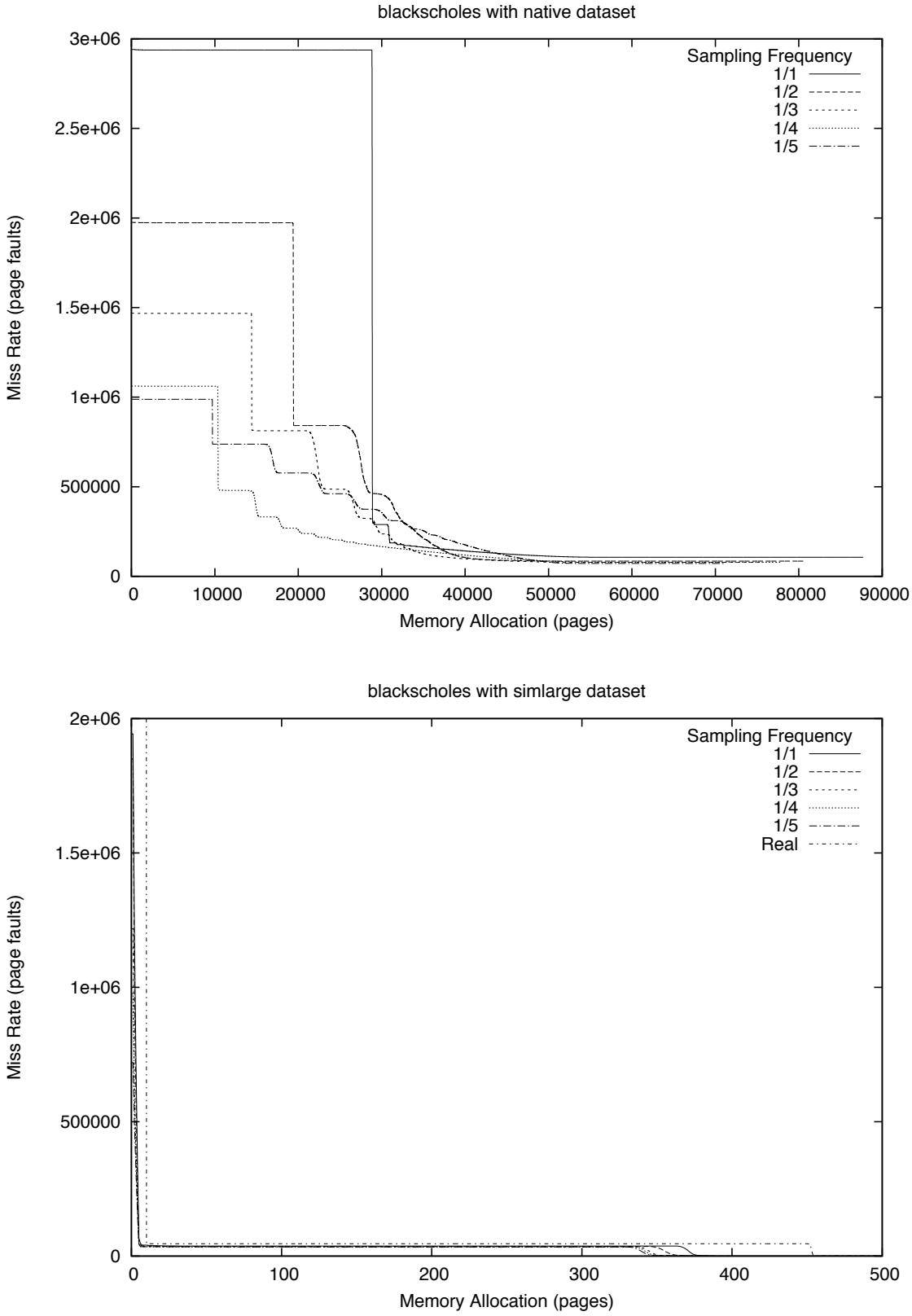


Figure 6.14: Blackscholes MRCs

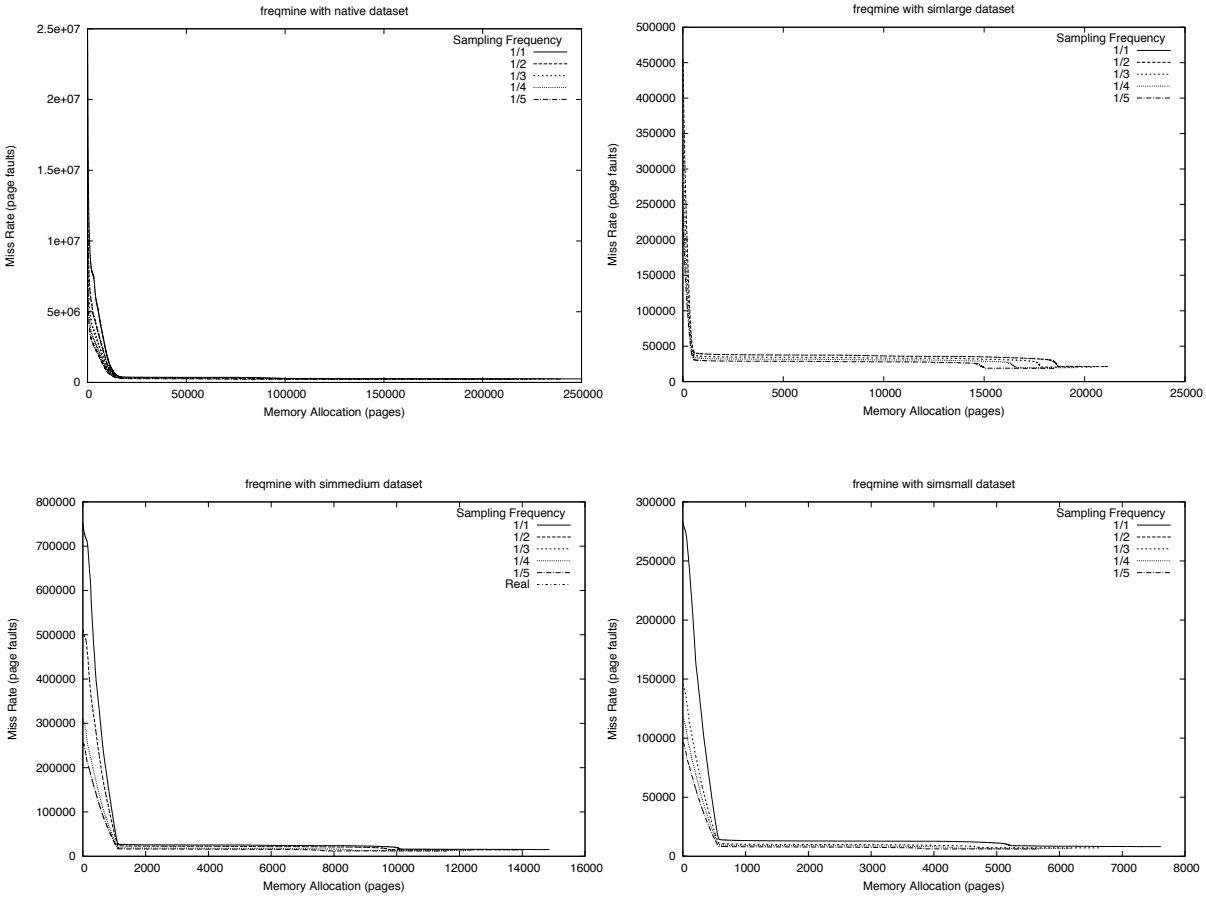


Figure 6.15: Freqmine MRCs

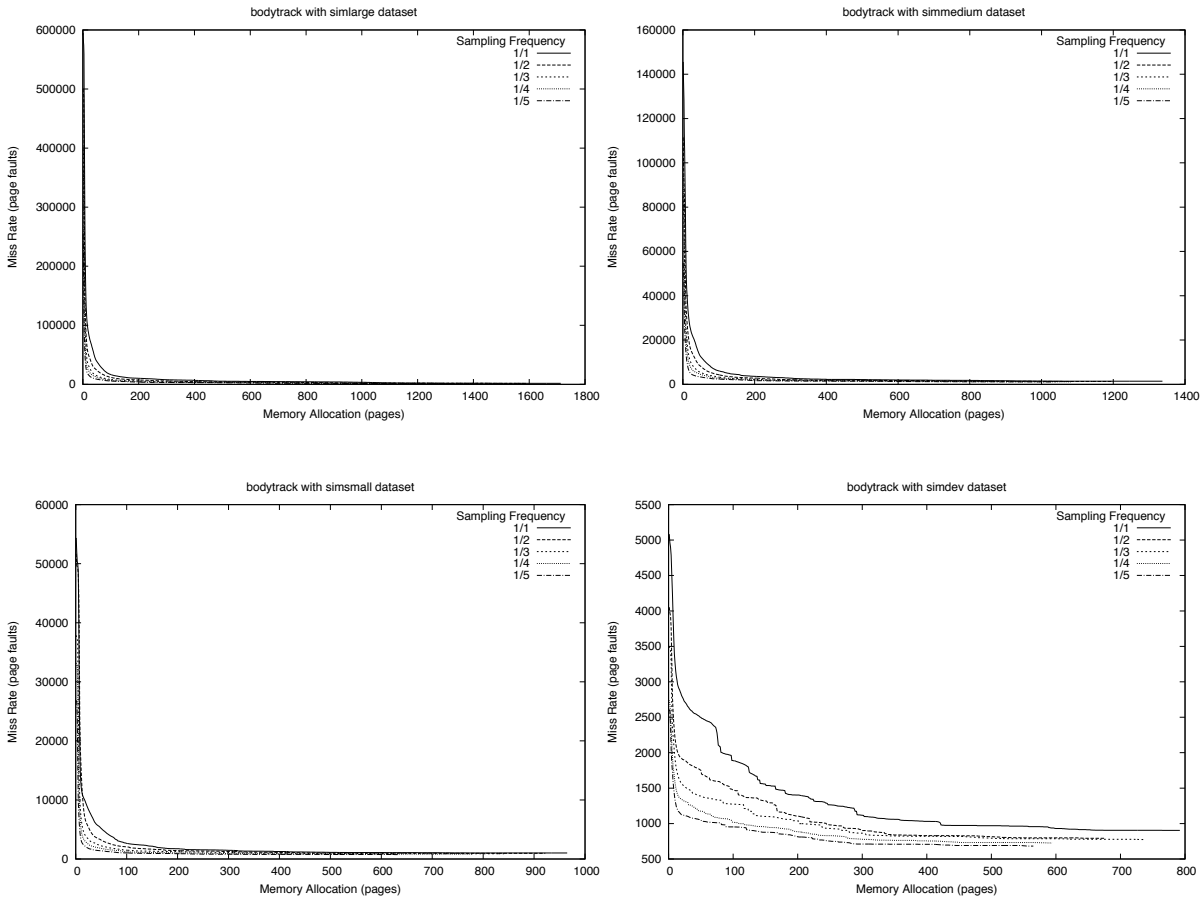
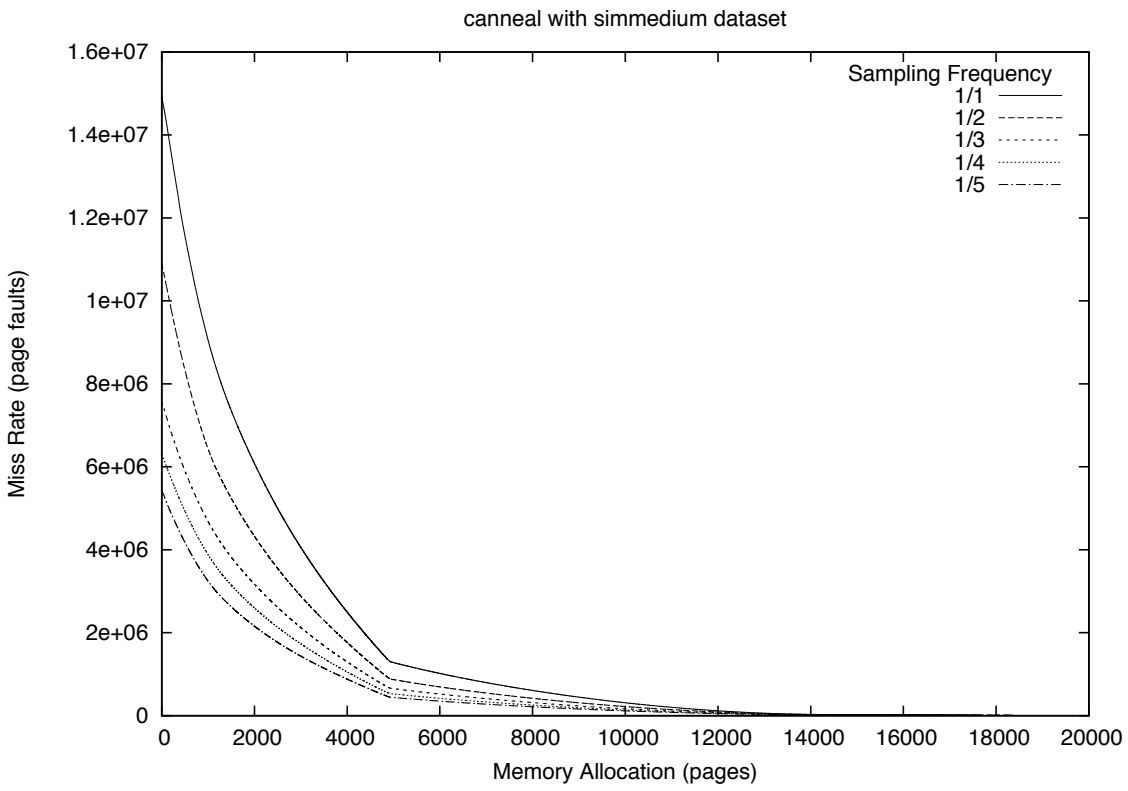


Figure 6.16: Bodytrack MRCs

ulation. Figure 6.17 on page 53 shows the canneal benchmark, which performs simulated annealing for chip design. Figure 6.18 on page 54 shows the x264 video encoding benchmark. They all show the sort of classical, convex, smoothly sloped MRC on which greedy memory allocators perform well.

The facesim facial movement simulation benchmark, shown in Figure 6.19 on page 55, and the fluidanimate fluid dynamics benchmark, shown in Figure 6.20 on page 56 both have very choppy MRCs. A greedy memory allocator risks finding a local optimum very far from the global optimum.

The final two benchmarks, the ferret content similarity search server in Figure 6.21 on page 57 and the streamcluster online clustering benchmark in Figure 6.22 on page 58 both have very close to linear MRCs. This is likely the result of a pseudo-random access



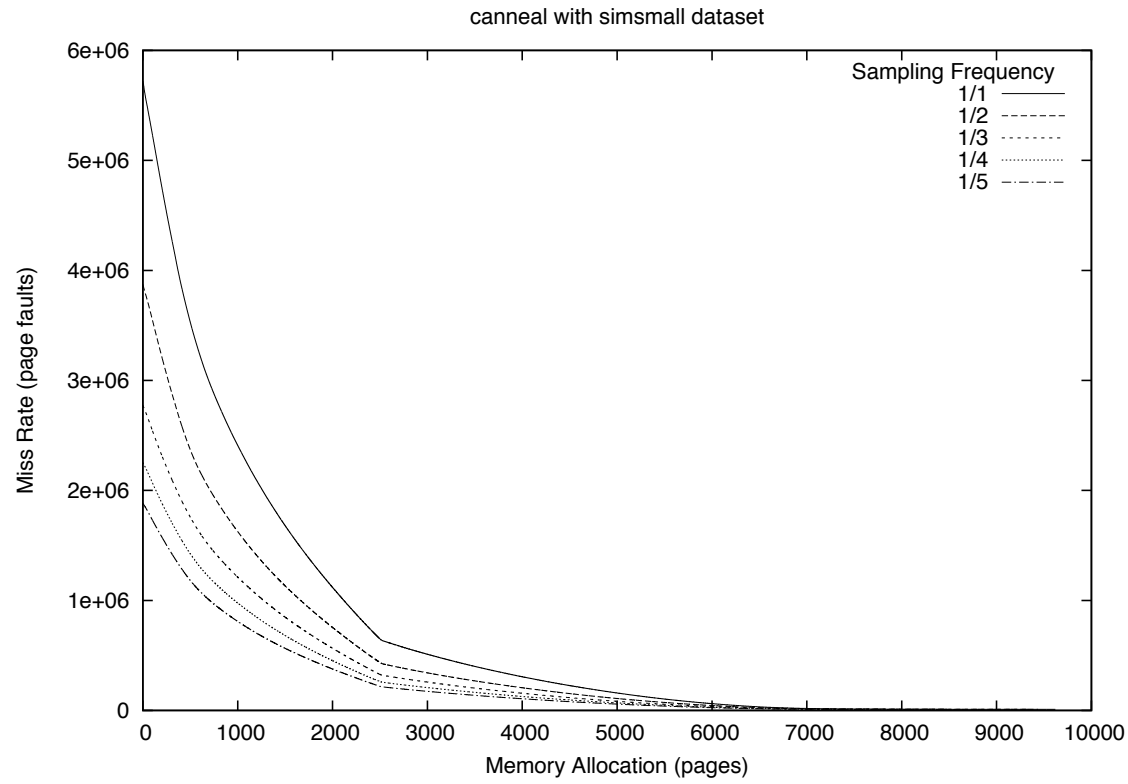


Figure 6.17: Canneal MRC

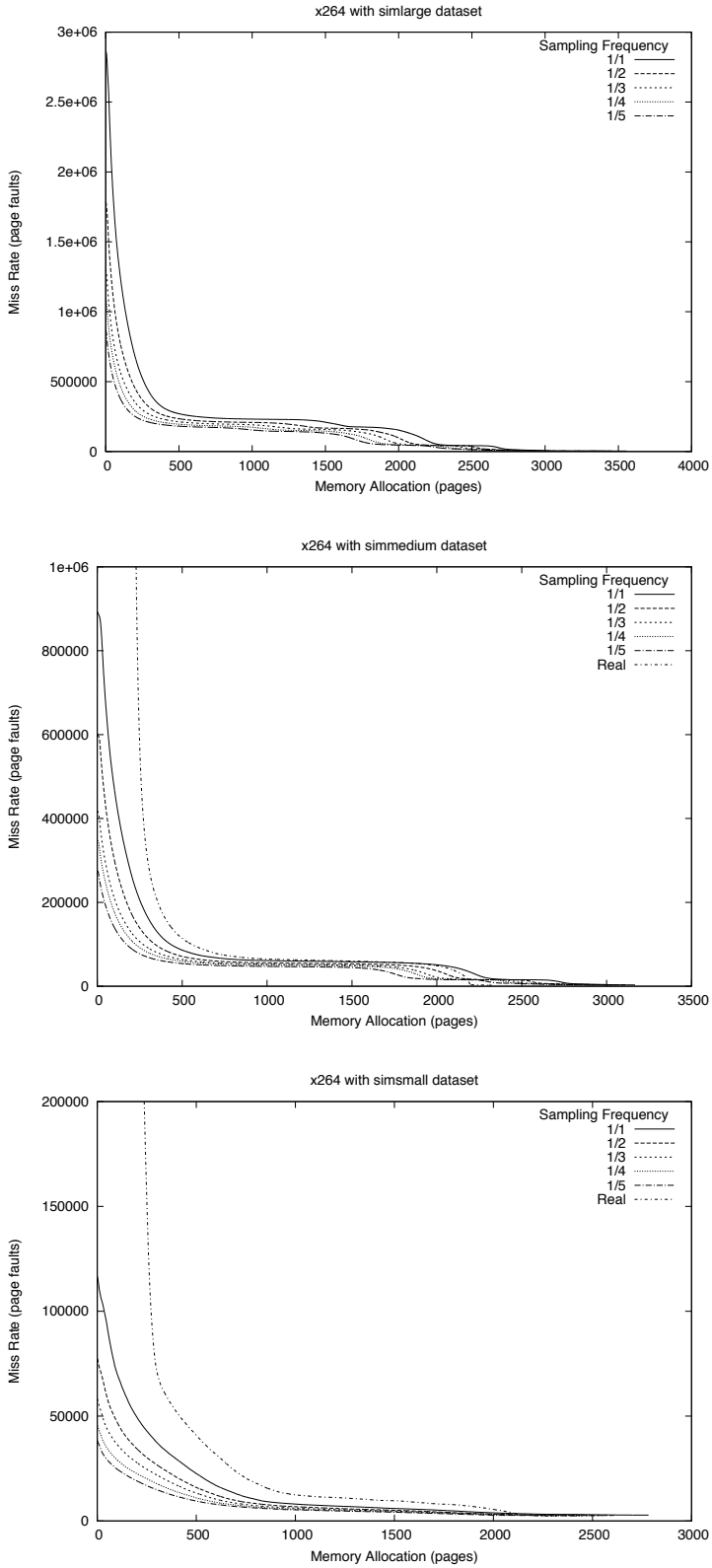
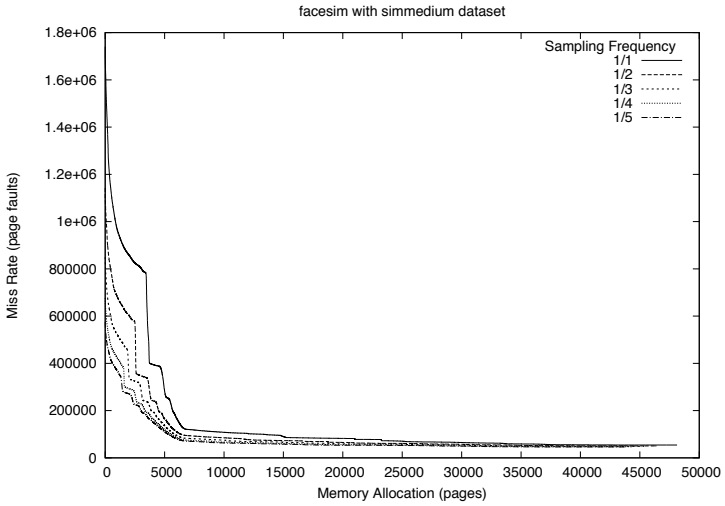
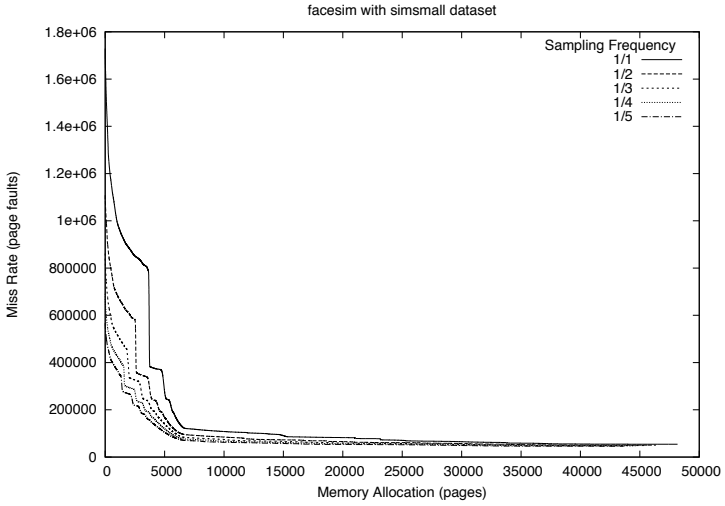


Figure 6.18: X264 MRCs





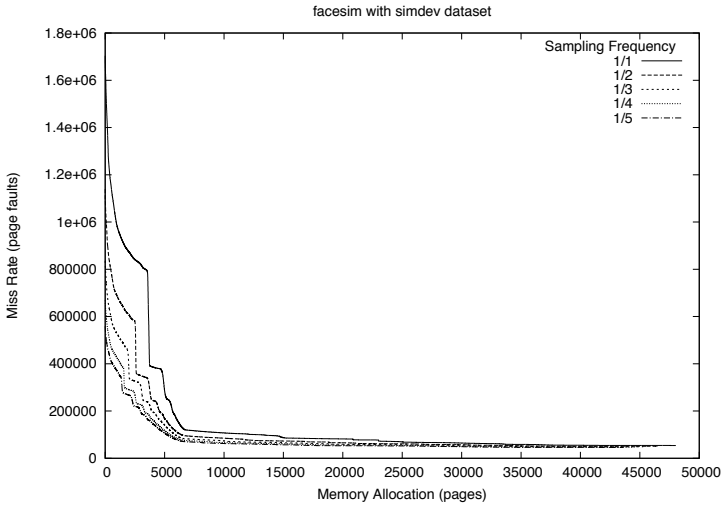


Figure 6.19: Facesim MRCs

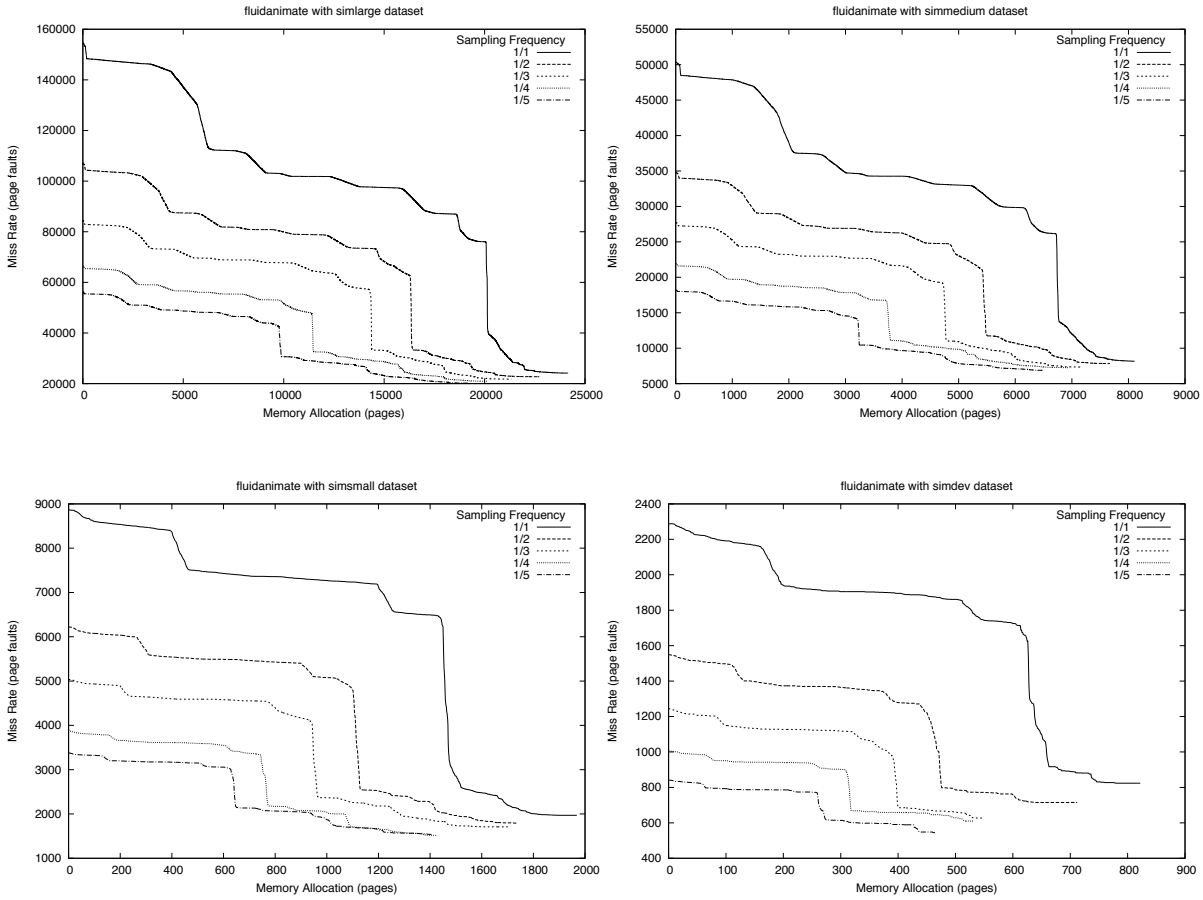


Figure 6.20: Fluidanimate MRCs

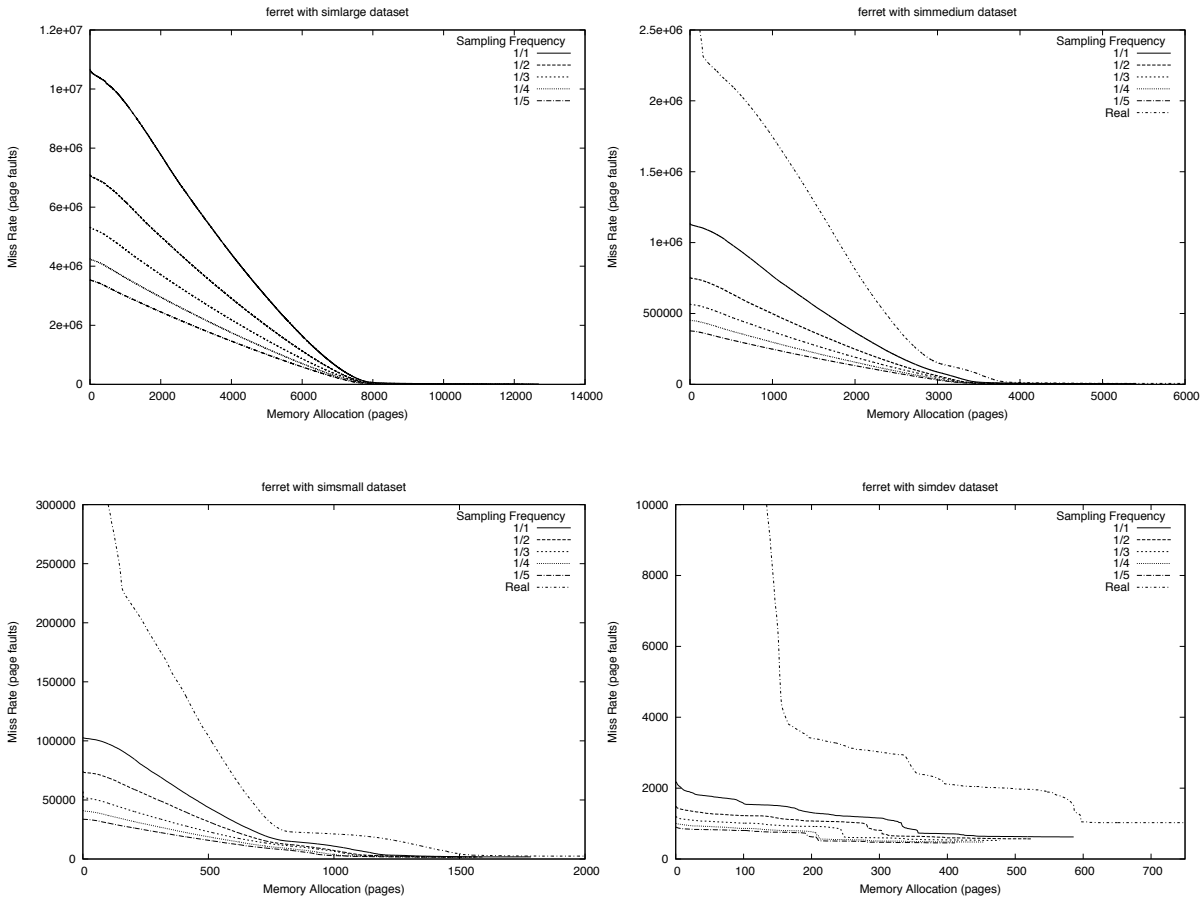


Figure 6.21: Ferret MRCs

pattern, as the probability of having a page in memory is proportionate to the amount of memory allocated.

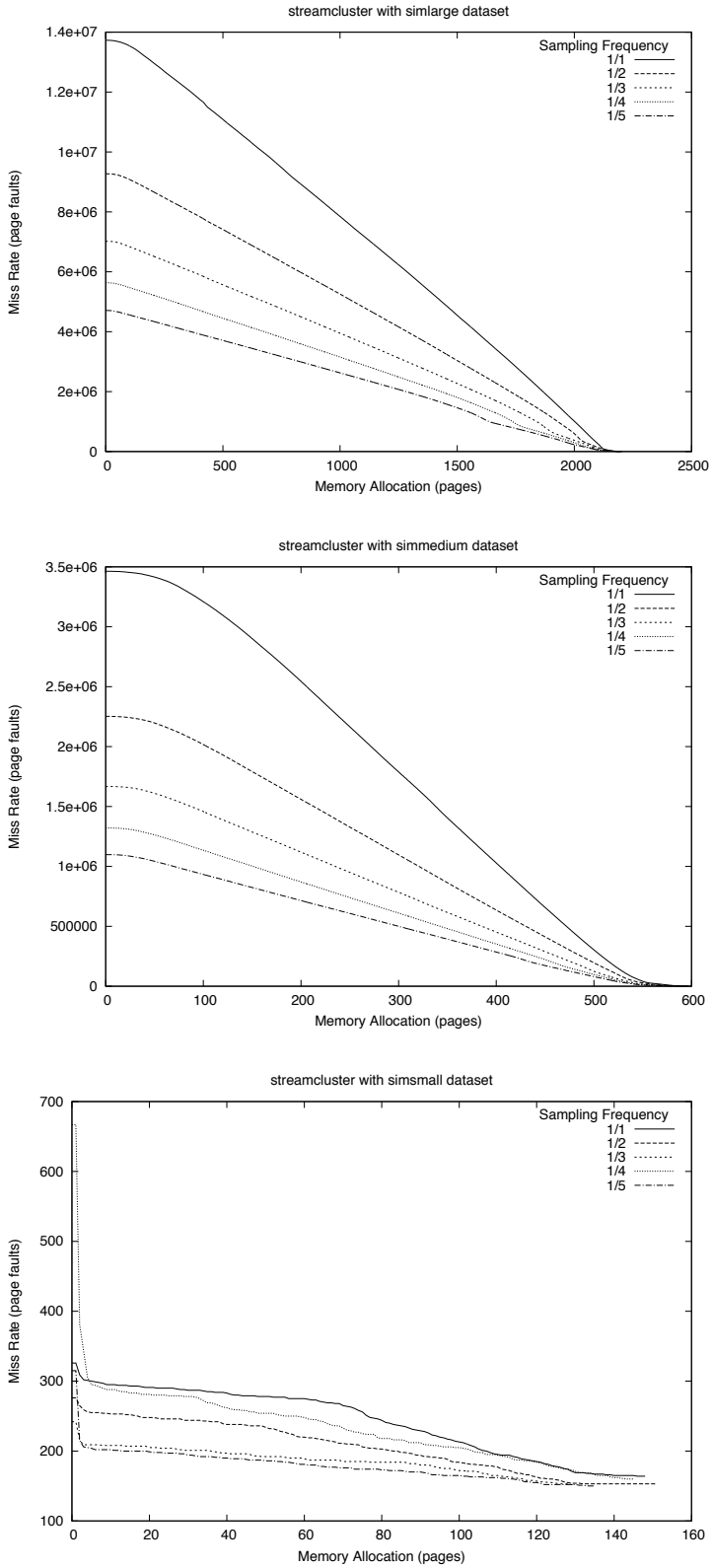


Figure 6.22: Streamcluster MRCs

Chapter 7

Discussion

The MRCs presented in Chapter 6 are similar in shape to those generated from a full trace of all memory accesses, but they are not perfect. In Section 7.1 we discuss the limitations of PEBS for MRC generation. From there we turn to a discussion of the future. Section 7.2 discusses possible hardware-based improvements. Section 7.3 looks at unanswered questions in MRC-related research and proposes their study, and it is hoped that this work may make this possible. Finally, Section 7.4 we conclude by looking at our results and what they mean going forward.

7.1 Applicability of PEBS to MRC Generation

The PEBS-generated MRCs were often similar in shape to those generated from a trace of all memory accesses, but not always. In these cases, it is an open question of which MRCs are more representative of real-world performance. While PEBS has its limitations, certain variations may be due to PEBS capturing real-world performance optimizations such as prefetching that are not captured through instrumentation. To answer this question, we must measure the actual page-fault rate of our benchmarks with varying memory allocations, which would be a very time-consuming experiment.

We do know, however, that PEBS suffers from data loss, and that while the shapes

of the curves are often correct, the magnitude is quite consistently off. It is unclear how much this would affect the performance of our MRCs in the applications described in Section 2.2. It is also possible that additional measurements, such as measurements of the existing memory allocation and page fault rate, might be used to adjust the scale of the generated MRCs.

To our knowledge, the accuracy of MRCs generated through software-based techniques has not been validated, so it is impossible to compare our accuracy to theirs. Our PEBS overheads of 5% are lower than the total overheads of software-based techniques, but omit the cost of Mattson's Stack Algorithm, so a direct comparison is not possible. Our overhead could likely be reduced further by only generating MRCs at the beginning of each phase as was done by RapidMRC for cache-level MRCs[29], which would reduce not only the PEBS collection overhead but also the cost of running Mattson's Stack Algorithm, for which high performance implementations already exist. This optimization could likely be applied to software-based techniques as well, however.

7.2 PMU Optimizations

While PEBS works for memory tracing, our experience led us to believe that with some improvements it could perform even better. Without expertise in microprocessor design, it is impossible to know how much complexity the following modifications would add to the performance monitoring unit; however, it seems plausible that they would be relatively minor.

The whole process of working with memory addresses could be significantly streamlined if the memory address was more readily available. Newer POWER CPUs from IBM expose a register to the programmer that contains the result of address calculations. However, these CPUs do not have a PEBS-like facility for automatic sampling, so an interrupt is required in order to read this register. If such a register were exposed

to the programmer on Intel CPUs and furthermore was included in the PEBS record format, memory addresses could be sampled with low complexity and low overhead.

While we have not confirmed this, it is believed that recording the register state is a significant source of both overhead and data loss when attempting to use PEBS with a high sampling frequency. For many applications, the complete register state is largely unnecessary. For instance, some applications require only the instruction pointer, while most memory-related work requires only the memory address. Providing alternate, smaller PEBS record formats only containing this information would likely reduce the time required to store PEBS entries and would certainly reduce the amount of memory required. It is interesting to note that Intel provides a number of bits for setting the PEBS record format, although currently there is only one valid setting.

7.3 Understanding MRCs

There is little published information about the general properties of MRCs and we believe this hinders research in MRC-based algorithms. It also may be the case that MRC-knowledge motivates or influences other memory management research, such as providing motivation for the use of local page replacement. Finally, better knowledge of MRC characteristics might allow us to improve our MRC generation techniques, either improving their accuracy or reducing their overhead. For instance, reliable relationships might be found between the shapes of imperfect and perfect MRCs and those relationships might be exploited to estimate the true MRC based upon one or more MRCs generated through imperfect sampling.

It is unknown to what extent the MRCs generated in existing MRC research are representative of the MRCs or real-world applications. In particular, we do not know the memory utilization characteristics of memory-intensive consumer software such as games and web-browsers.

It is still an open question whether LRU MRC shapes can be used to predict the shape of non-LRU MRCs and hence predict the performance of these replacement policies. If such insight were available, it might be possible to modify our replacement policies on the fly to improve performance.

It is hoped that the availability of low-overhead MRC generation techniques such as ours will facilitate greater study into MRCs and their properties.

7.4 Conclusions

We found that PEBS-based MRCs could produce MRCs with similar characteristics to MRCs based on complete memory access traces, but with significantly lower magnitudes and some discrepancies in shape. We believe the magnitude differences are reflective of PEBS' limitations, while it is an open question whether the shape discrepancies are more reflective of the limitations of PEBS or of real-world behaviour that is not captured in the perfect memory access traces. We do not have enough information about the other MRC generation techniques to compare our accuracy against theirs.

The overheads of PEBS-based data collection were below 5% on average, but not uniformly low, peaking at around 20%. These overheads, however, are not directly comparable against alternate MRC generation techniques. We believe it is possible to reduce these overheads further using techniques that have been applied to MRCs for L2 caches.

It is an open question whether PEBS is the best available method of generating MRCs. However, we feel that PEBS has been shown to be good enough to motivate the continuing use of general purpose hardware performance monitoring units for this main memory MRC generation. Perhaps it may also motivate hardware designers to develop features of their hardware performance monitoring hardware with MRC generation and other tasks requiring memory tracing in mind.

Bibliography

[1] G. Abandah. Configuration independent analysis for characterizing shared-memory applications. *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, pages 485–492, 1998.

[2] J. Appavoo, M. Auslander, M. Butrico, D. M. da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Experience with k42, an open-source, linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[3] Reza Azimi, Livio Soares, Michael Stumm, Thomas Walsh, and Angela Demke Brown. Path: page access tracking to improve memory management. *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 31–42, 2007.

[4] Reza Azimi, Michael Stumm, and Robert W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110, 2005.

[5] Jesse G. Beu. Pmpt - performance monitoring pebs tool. Master's thesis, North Carolina State University, January 2006.

[6] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based i/o prefetching for out-of-core applications. *ACM Trans. Comput. Syst.*, 19(2):111–170, 2001.

[7] Stéphane Eranian. Perfmon2: a flexible performance monitoring interface. *Proceedings of the Linux Symposium*, 1:269–288, July 2006.

[8] Intel® Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, February 2008.

[9] James K Archibald J. Kelly Flanagan, Brent E. Nelson and Knut Grimsrud. Bach: Byu address collection hardware, the collection of complete traces. *Proceedings of the 6th Int. Conf on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 128–137, 1992.

[10] Song Jiang and Xiaodong Zhang. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 31–42, 2002.

[11] Magnus Karlsson and Per Stenström. An analytical model of the working-set sizes in decision-support systems. *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 275–285, 2000.

[12] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 9–9, 2000.

[13] Yul H. Kim, Mark D. Hill, and David A. Wood. Implementing stack simulation for highly-associative memories. *SIGMETRICS '91: Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 212–213, 1991.

[14] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12, 2009.

[15] Pin Lu and Kai Shen. Virtual machine memory access tracing with hypervisor exclusive cache. *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–15, 2007.

[16] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[17] Sally A. McKee. Reflections on the memory wall. *CF '04: Proceedings of the 1st conference on Computing frontiers*, page 162, 2004.

[18] PARSEC Benchmark Suite. *parsec.cs.princeton.edu*.

[19] Philip Patchin, H. Andrés Lagar-Cavilla, Eyal de Lara, and Michael Brudno. Adding the easy button to the cloud with snowflock and mpi. *HPCVirt '09: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 1–8, 2009.

[20] R. H Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. Technical report, Pittsburgh, PA, USA, 1995.

[21] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, 2006.

[22] D. Slutz R. Mattson, J. Gecsei and I. Traiger. Evaluation techniques and storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[23] Madhusudan Raman. Trace-based optimization for precomputation and prefetching. Master's thesis, University of Toronto, 2006.

[24] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 14–26, 1993.

[25] E. Torrie J.P.Singh S. Woo, M. Ohara and A. Gupta. Methodological considerations and characterization of the splash-2 parallel application suite. *ISCA*, 23, May 1996.

[26] Florian T. Schneider, Mathias Payer, and Thomas R. Gross. Online optimizations driven by hardware performance monitoring. *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 373–382, 2007.

[27] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 41(9):1054–1068, 1992.

[28] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1):7–26, 2004.

[29] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 121–132, 2009.

[30] Vivek Thakkar. Dynamic page migration on ccnuma platforms guided by hardware tracing. Master's thesis, North Carolina State University, 2008.

[31] Myles G. Watson. Does the halting necessary for hardware trace collection inordinately perturb the results? Master's thesis, Brigham Young University, 2004.

[32] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Cramm: virtual memory support for garbage-collected applications. *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 103–116, 2006.

[33] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 177–188, 2004.

[34] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 91–104, 2001.