Prototyping a Hybrid Main Memory Using a Virtual Machine Monitor

Dong Ye[†], Aravind Pavuluri[‡], Carl A. Waldspurger[‡]
Brian Tsang[§], Bohuslav Rychlik[§], Steven Woo[§]
[†]*Northeastern University,* [‡]*VMware, Inc.,* [§]*Rambus, Inc.*
[†]*dye@ece.neu.edu,* [‡]*{apavuluri,carl}@vmware.com,* [§]*{btsang,brychlik,swoo}@rambus.com*

***Abstract* — We use a novel virtualization-based approach for computer architecture performance analysis. We present a case study analyzing a hypothetical hybrid main memory, which consists of a first-level DRAM augmented by a 10-100x slower second-level memory. This architecture is motivated by the recent emergence of lower-cost, higher-density, and lower-power alternative memory technologies. To model such a system, we customize a virtual machine monitor (VMM) with delay-simulation and instrumentation code. Benchmarks representing server, technical computing, and desktop productivity workloads are evaluated in virtual machines (VMs). Relative to baseline all-DRAM systems, these workloads experience widely varying performance degradation when run on hybrid main memory systems which have significant amounts of second-level memory.**

I. Introduction

Current uses of PC-based virtualization have included server consolidation, enhanced security, and desktop centralization [20]. In this work we explore a new role for virtualization: performance modeling and analysis of new computer architectures. This has conventionally been done using software simulators, which typically offer great modeling flexibility at the cost of simulation speed [21].

A virtual machine monitor (VMM) is similar to a software simulator by providing an execution environment to run software. A VMM achieves higher performance by interposing only when necessary to manage privileged machine state and typically requires that the guest operating system and host machine share the same instruction set. This restricts the breadth of computer architectures that can be analyzed by a VMM-based tool. Nevertheless, for some scenarios, the combination of evaluation speed, freedom in target workload, and complete execution is attractive. We identify one such scenario and perform a case study.

We target a hybrid main memory system featuring an additional level of memory inserted into the traditional memory hierarchy between DRAM and disk. This choice is motivated by two factors. First, recent memory technology developments offer some lower-cost, higher-density, and lower-power alternatives/complements to DRAM. Second, the software-based memory virtualization implemented in VMMs appears well-suited to model a hybrid main memory.

The rest of the paper is organized as follows. Section II reviews relevant trends in computing, memory technologies, and virtualization. Section III describes the architecture of a generic hybrid main memory and its performance model. Section IV describes our experimental method. Section V presents quantitative performance data, which is analyzed and discussed in Section VI. We summarize related work in Section VII and give our conclusions in Section VIII.

II. Background

A. Processor and Computing Trends

The power and thermal challenges faced when improving single-core processor performance are driving a shift to multi-core systems. These multi-core systems support more simultaneous applications and their memory working sets, thereby increasing pressure on system memory capacity. One way to cost-effectively address the need for capacity is to introduce a new layer in the memory hierarchy between DRAM and disk. Such a layer would bridge the five orders of magnitude performance gap between DRAM and disk, while offering a middle-ground cost structure.

B. Alternative Memory Technology Trends

Several current and emerging memory technologies (*e.g.*, NAND flash, NOR flash and Phase Change Memory) possess characteristics making them candidates for this new memory layer. As an example, we compare the advantages and disadvantages of NAND flash with DRAM. Table 1 shows the density of DRAM and NAND flash over the next few years as predicted in the ITRS 2007 roadmap [12]. Table 2 compares DRAM and NAND flash in cost, performance, power consumption, and endurance. Over the next 10 years, NAND flash will consume 10x less active power and 100x less standby power, command 10x more density, and cost 4-8x less than DRAM. However, NAND flash is 10-100x slower than DRAM in access latency and data transfer rate, and has asymmetric read and write speeds (writes are 10x slower than reads). Furthermore, NAND flash must be accessed at a page granularity of up to 16KB, and each page can only be written a limited number of times.

Although our study is guided in part by the salient characteristics of NAND flash, our method is not restricted to this technology. Rather, we allow a range of parameters for the hybrid memory, applicable to other technologies.

Table 1: ITRS 2007 roadmap for DRAM and NAND flash

Gbits/cm^2	2007	'08	'09	'10	'11	'12	'13	'14	2015
DRAM	2.31	2.91	3.66	4.62	5.82	7.33	9.23	11.63	14.65
Flash SLC	5.97	8.44	10.60	13.40	16.90	21.30	26.80	33.80	42.50
Flash MLC	11.90	16.90	21.30	26.80	33.80	42.50	53.60	67.50	85.10

Table 2: Cost, performance, power, and reliability comparison between DRAM and NAND flash

	Erase Time	Write Time	Read Time	Data Rate (Write)	$ Cost	Capacity/Chip	Active Power/Chip	Standby Power/Chip	Write Endurance
DRAM	N/A	<100ns	<100ns	800MB/s	~$2/Gb	1Gb	~500mW	~mW	∞
SLC	2ms	200µs	25µs	10+MB/s	$0.9/Gb	8Gb	80mW	50µW	10^5
MLC	2ms	800µs	50µs	10MB/s	$0.3/Gb	16Gb	80mW	50µW	10^4

C. ESX Server Software VMM

We implement the performance model of hybrid main memory by customizing the VMM component of a VMware ESX Server hypervisor [27]. Two existing features facilitate our implementation. First, an extra level of memory indirection exists in the VMM which translates guest physical addresses to host physical addresses used to access hardware [28]. The VMM manages the allocation and mapping of guest physical memory by maintaining metadata for each guest physical page. In simulating a hybrid main memory, we enhance this metadata to track an extra attribute for each guest physical page indicating whether it is of DRAM type or of alternative memory type. Second, the VMM implements a general-purpose memory protection mechanism called *tracing*. Upon accessing a trace-protected guest physical page, execution is directed to a designated handler in the VMM [1]. We employ a custom trace that triggers on accesses to alternative memory pages. The custom trace handler simulates the desired timing and collects statistics.

III. ARCHITECTURE

A. Hybrid Main Memory System

Figure 1 shows the conceptual organization of a hybrid main memory system similar to the multi-level memory system proposed in [7, 8]. Since our exploration is similar in spirit, we adopt the terminology used in [8]: a second-level memory *M2*, comprising the alternative memory devices, augments a first-level memory *M1* of conventional DRAM. We assume that an entire page is always transferred from M2 to M1 before any of its data is accessible to the CPU.

Many different mechanisms could be employed to manage the mapping and movement of memory pages between M1 and M2 including both hardware and software techniques. Some of the possibilities include specialized memory controllers, a new OS-level memory management component, or a new hypervisor-level memory virtualization component. Exploration in this direction is beyond the scope of this paper. We instead focus on characterizing first-order performance implications of such a system using a VMM-based performance evaluation method.

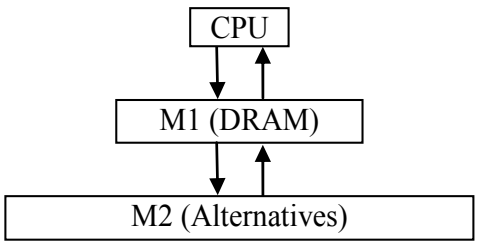

Figure 1: A computer system featuring hybrid memory

B. Performance Model

In establishing the performance model of hybrid main memory systems, we focus on the time penalty of M2 accesses. Our platform takes the following actions on each M2 page access: (1) The previously-installed custom trace handler is triggered. (2) A victim M1 page is chosen randomly across the entire physical memory space. (3) M2 write latency is optionally incurred to account for transferring the victim M1 page to M2. (4) M2 read latency is incurred. (5) Additional delay is optionally incurred to reflect read/write bandwidth limits. (6) The M1/M2 markings of the victimized and faulted pages are swapped and so are their trace associations. No actual exchange of content is performed. While more sophisticated victim selection policies such as LRU are likely to perform better, our choice of random replacement provides a conservative baseline.

We provide several parameters for configuring the hybrid memory system simulated by our performance model:

- *VM total memory size* and *M2 fraction* of total memory.
- *Read latency*: Time to access and transfer one page from M2 to M1. It is incurred upon an M2 access.
- *Write latency*: Time to transfer one page from M1 to M2. It is incurred upon an M2 access unless it is hidden by the write buffer or when the M1 victim page is unmodified under an inclusive caching organization.
- *Read bandwidth*: Read bandwidth limit is specified via two values, a time window (L_R) and a maximum number of read operations (M_R) allowed within the time window. Two state variables are maintained: counter (C_R) and timestamp (TS_R). C_R starts from *0*, increments upon each M2 access, and returns to 0 at M_R. TS_R records the time corresponding to the M2 access when $C_R = 0$. When $C_R = M_R$, we delay until the time elapsed since TS_R equals L_R, and reset C_R and TS_R.
- *Write bandwidth*: Same as above but on the write side.
- *Write buffer*: Modeled by a ring buffer. Each buffer slot keeps a timestamp, recording the time when this slot is used to hold the content of a victim M1 page. When a M1 page is victimized, if a write buffer slot is available, then no delay is incurred. Otherwise, we delay until the time elapsed since the oldest timestamp equals the write latency (*i.e.*, the oldest buffer slot becomes available.)
- *Caching*: Two organization alternatives are modeled. If M1 and M2 operate as an *exclusive* hierarchy, any M1 victim pages must be written to M2. In an *inclusive* hierarchy, each M1 page is backed up in M2, so only

modified M1 pages must be written to M2.

- *Sampling frequency*: M2 access statistics are logged and reset periodically. Two choices of this sampling frequency are supported: 1Hz and 10Hz.

IV. Experimental Method

A. Experimental Setup

We measure the performance of complete systems consisting of unmodified applications running on unmodified, commodity operating systems. The experimental process involves the following steps for each application and configuration: (1) Configure a VM for the performance model described in III.B. (2) Start the VM on the custom virtualization platform. (3) Run the application to its completion in the guest OS. (4) Shut down the VM making sure to record the relevant metrics and access statistics.

For each application under investigation, we measure an appropriate application-specific performance metric. We establish a *baseline VM* for each application: this is an M1-only system where the VM guest memory is specifically sized to the application. In most cases, this moderately exceeds the memory footprint. The memory footprint is the memory size of a VM that runs the application and guest OS without noticeable paging activity; while halving it would result in substantial disk paging.

For each hybrid main memory configuration, we report the normalized performance of applications relative to their performance on baseline VMs. Note that we measure wall-clock execution time using an external time source. This is necessary since under heavy load, the VMM may distort the precise timing of guest timer interrupt delivery [26], which is exacerbated by our introduction of substantial M2 delays.

To minimize the variations introduced by hypervisor activities as well as concurrently-executing VMs, we ensure that the host processor and memory are under-committed when carrying out our experiments. First, we run only one VM at a time. Second, the host is configured with ample memory (8GB). Third, all experimental VMs are configured with a single virtual CPU bound to a dedicated host processor core.

B. Benchmarks

We use benchmarks representative of server, technical computing, and desktop productivity workloads. Table 3 catalogs their memory footprints, baseline VM memory sizes, and the guest operating systems.

Table 3: The benchmarks

Name	Workload	Footprint	Baseline VM	Guest OS
DB	Oracle 10g /	800MB	1024MB	64-bit Linux
JBB	SPECjbb2005	400MB	512MB	32-bit Linux
KC	Kernel Compilation	300MB	512MB	32-bit Linux
Deal2	SPEC CFP2006	500MB	512MB	32-bit Linux
MCF	SPEC CINT2006	900MB	1536MB	32-bit Linux
WinB	Business Winstone	128MB	512MB*	32-bit Windows

We represent server workloads using a database (DB) and a Java-based business-logic (JBB) benchmark. We use Swingbench [10] as the database client and load generator to drive and test an Oracle 10g database server running inside a 64-bit Linux VM. The Swingbench client process runs on a native Windows XP system and sends requests over the local network to the Oracle 10g database server. We use the transactions-per-second metric reported by the Swingbench client. SPECjbb2005 represents the middle tier of a three-tier client/server system with emphasis on components such as the JVM [22]. It calculates a business-operations-per-second (bops) value. We use this as its performance metric, re-scaled appropriately to the wall-clock execution time. The benchmark runs inside a 32-bit Linux VM provisioned with a BEA Jrockit 1.6.0 JRE.

Kernel compilation (KC), Deal2, and MCF are technical computing workloads. Kernel compilation builds the Linux kernel from vanilla Linux 2.6.21 source code. Deal2 and MCF are from the SPEC CPU2006 suite [23], both featuring significant memory footprints and changing memory demand over time. All are run inside a 32-bit Linux VM and execution speed is reported (inverse of execution time).

Business Winstone 2002 (WinB) is a desktop productivity benchmark representing real-world office usage [9]. In a single run, a session of operations with Word, Excel, PowerPoint, Access, FrontPage (all from Microsoft Office), Netscape, Microsoft Project, Norton Anti-Virus, and Lotus Notes is performed. The scripted session is tailor-made to include effects such as user wait time. It runs inside a 32-bit Windows XP VM. Since this workload performs the same operations during each run, we use its execution speed instead of a suite-specific score as its performance metric.

V. Results

A. Performance Sensitivity to VM Memory Size

Figure 2 shows application performance with respect to the VM memory size. A key observation is that application performance is largely a make-or-break case with respect to the total memory size. The performance-sensitive range is

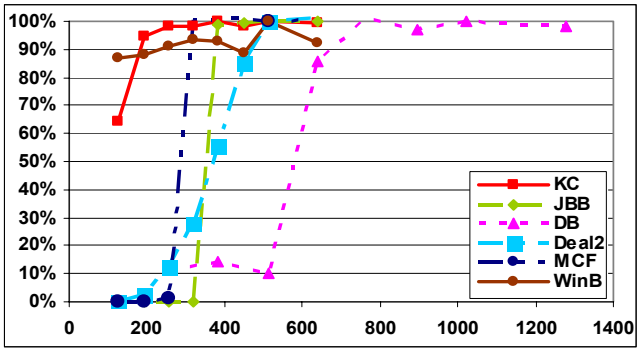

Figure 2: Performance sensitivity to memory size on a M1-only system. X-axis is VM memory size (MB). Y-axis is normalized performance (relative to baseline VMs).

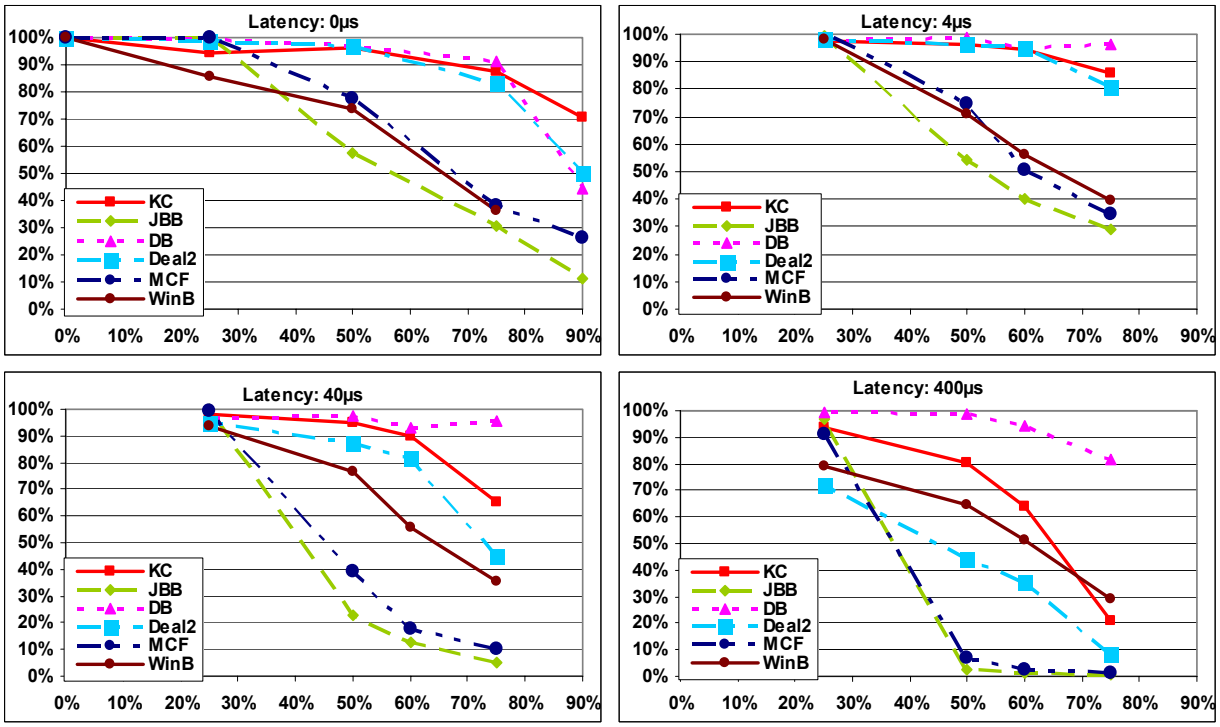

Figure 3: Performance impact of M2 read latency. Each subfigure evaluates a particular value of M2 read latency. X-axis is M2 fraction. Y-axis is normalized performance (relative to baseline VMs).

also quite narrow: Once the VM memory size exceeds the footprint of a workload, additional memory yields little performance benefit. These results guide our baseline VM memory sizing for all applications except WinB. Unlike other applications, WinB's footprint (128MB) is smaller than typical systems running Windows XP. To remain consistent with typical systems, we sized the baseline VM for WinB at 512MB.

B. Performance Impact of M2 Latency

In Figure 3, we plot the benchmarks' performance sensitivity to M2 fraction and M2 latency. The benchmarks exhibit widely-varying tolerance to these parameters. A hybrid memory of 25% M2 (40µs read latency) causes less than 5% performance degradation to all the benchmarks. DB exhibits a high level of tolerance to both large M2 fractions and long M2 read latencies—a 60% M2 with 400µs read latency yields only 7% performance degradation. KC and Deal2 show medium tolerance to M2 latency, while JBB and MCF suffer even under a modest M2 fraction. WinB shows similarities with both M2-sensitive and M2-insensitive applications depending on the M2 latency.

C. Performance Impact of M2 Bandwidth

In Figure 4, we plot the performance sensitivity to M2 bandwidth by varying the bandwidth limit (assuming zero M2 latencies). We also report the average and peak demand for M2 bandwidth (using 10Hz sampling) when the application is not constrained.

It is not surprising to see that demand for M2 bandwidth (*i.e.*, M2 access throughput) increases as the specified M2

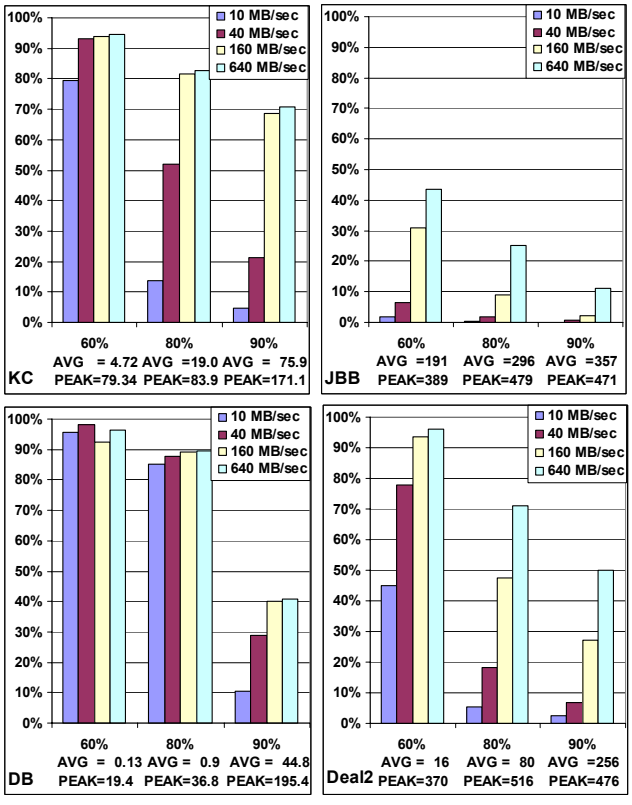

Figure 4: Performance impact of M2 bandwidth capacity. Each subfigure evaluates one benchmark. X-axis is M2 fraction (commented with application's demand for M2 bandwidth, both average and peak, in MB/s). Y-axis is normalized performance (relative to baseline VMs). It is noted that performance under 640MB/s is indistinguishable from that under unlimited bandwidth. MCF graph is omitted as it demonstrates a very similar pattern as JBB.

fraction increases. Yet, the extent to which an M2 bandwidth limitation can influence performance is benchmark-specific: JBB is much more sensitive to M2 bandwidth limitation than KC and DB.

As described in Section III.B, a bandwidth limit is enforced at a parameterized time window. All bandwidth values in Figure 4 are enforced using a 10ms window.

D. *Effectiveness of Write Buffers*

The long write latency of some candidate M2 memory technologies makes their viability questionable. A natural approach to accommodate slower writes is write buffering. Comparing the two subfigures of Figure 5, we can see that a modestly-sized write buffer of 64 pages effectively hides write latencies as long as 1ms. Hence, write latency is not necessarily a problem in the hybrid main memory system.

E. *Utility of Inclusive Caching*

Some potential M2 memories support only a finite number of writes over their lifetime. To reduce writes, we experimented with inclusive- as well as exclusive-caching policies. Under inclusive caching, only dirty M1 victim pages are written back to M2. However, we observe only modest write savings with this approach (1-20%) for 75% M2 fraction, suggesting limited utility of this technique.

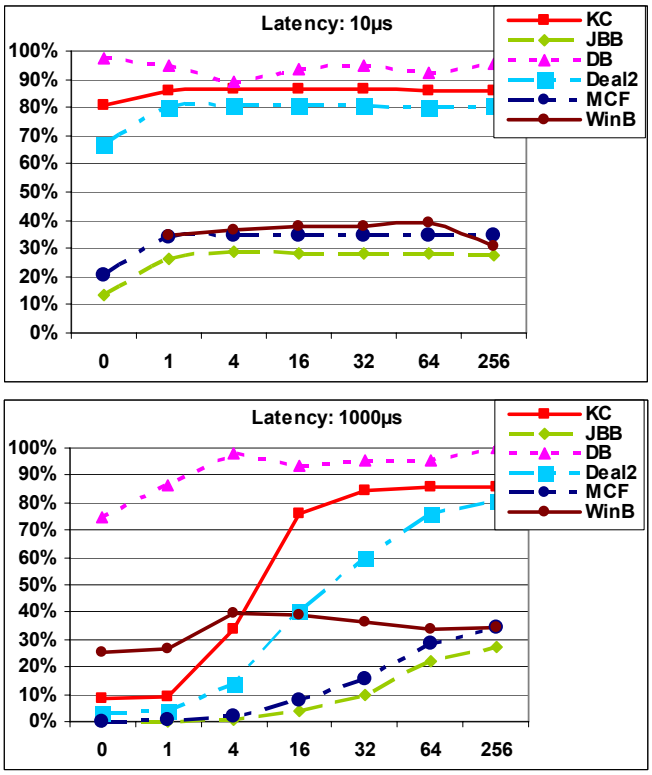

Figure 5: The effectiveness of write buffers to hide M2 write latency. Each subfigure evaluates one write latency value. X-axis is the write buffer size (number of M2 pages). Y-axis is normalized performance (relative to baseline VMs). All VM are configured with 4µs read latency and 75% M2 fraction.

VI. Discussion

A. *Measurement versus Simulation*

A prototyping tool is valuable in the exploratory stage of new system designs. Developing such a tool is often a balancing act, trading off details, speed, flexibility and workload support. For example, Ekman et al. [7] used a complex methodology employing simulation, direct measurement of native execution, and various extrapolations.

Virtualization offers an interesting alternative to address this problem. We have demonstrated the capability of a VMM-based tool to run a variety of full workloads to completion in real time. We were also able to explore a large evaluation space simply by changing VM settings. Thus, within the constraints of our chosen scenario, we achieved a blend of evaluation speed and flexibility. In the following sections, we validate the timing accuracy of our approach.

B. *Measurement Bias*

Since we measure wall-clock execution time, we must ensure that the impact of non-guest components included in the wall-clock execution time doesn't distort the comparisons between executions of the same guest code under different configurations. Non-guest components include synchronous VMM code (trace delivery cost), asynchronous hypervisor activities (virtualization, scheduling, etc.), as well as M2 access simulation/instrumentation code.

We expected minimal environmental disturbance (*e.g.*, cache pollution) due to the modest footprint of simulation code and data (<200 lines of C code and ~4KB of internal states). We also expected minimal disturbance from asynchronous hypervisor activities, given our tight control over the host and the experiment environment. The left part of Figure 6 exhibits a linear relation between the application slowdown and M2 access latency, validating the expectation of minimal environmental disturbance affecting guest execution and suggesting a fixed trace delivery overhead.

However, in the right part of Figure 6, we observe that as our target latencies drop below 10µs, the plots are no longer linear. We suspect this is because the actual induced latency exceeds the specified latency, due to trace delivery and instrumentation overhead. Hence, our platform tends to overestimate the slowdown for small M2 latencies of a few microseconds. We confirm this limitation of our platform by adding instrumentation to count *overtimes*. Overtimes occur when our inserted simulation and delay calculation code takes longer to execute than the specified latency. We confirmed that overtimes occur most frequently for latencies below 4µs, and almost never for latencies greater than 10µs. We also observe a trend of decreasing ability to simulate small latencies as M2 is accessed less frequently. This occurs when the M2 portion is small. We hypothesize that this is due to the decreasing likelihood that the delay simulation code and data remain resident in the processor caches. In all cases, however, the pessimistic bias of our platform decreases as the slowdown due to M2 accesses increases, whether through longer M2 latencies or through more frequent M2 accesses.

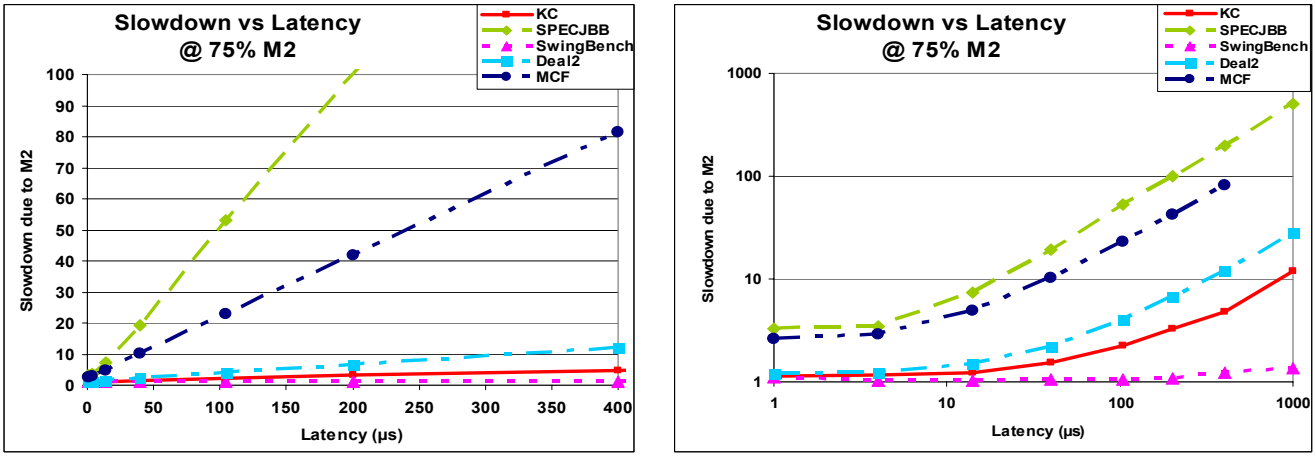

Figure 6: A linear relationship between normalized slowdown and M2 access latency is observed. Y-axis is the slowdown from baseline performance where there is no M2. X-axis is the read latency. The right subfigure plots the same data in logarithmic scale and shows that as latency drops below 10µs, the relationship is no longer linear.

Our platform is never optimistic.

C. *Observational Granularities*

When generating the peak bandwidths reported in section V.C, we record and reset aggregated statistics of M2 accesses for each sampling period. Since our implementation is limited to a maximum sampling rate of 10Hz, our observation granularity is limited. We want to estimate the likelihood that this limitation leads to information loss. Our approach is to see how much additional information we gain by sampling at 10Hz versus 1Hz. In Figure 7, we compare the page access histograms attained with 10Hz sampling to those we would be able to extrapolate from 1Hz sampling, assuming a uniform distribution for M2 access inside a sampling period. When these histograms generally match, as for MCF, we have confidence that we are not missing major trends with our 10Hz sampling. However, mismatches as observed in KC, suggest that even finer sampling granularity would be desirable to detect short-lived periods of high M2 throughput. In fact, bursty M2 accesses can explain the seemingly counterintuitive observation from Figure 4: Even when the bandwidth limit exceeds the peak 10Hz-sampled throughput, performance still lags behind the case in which bandwidth is unlimited.

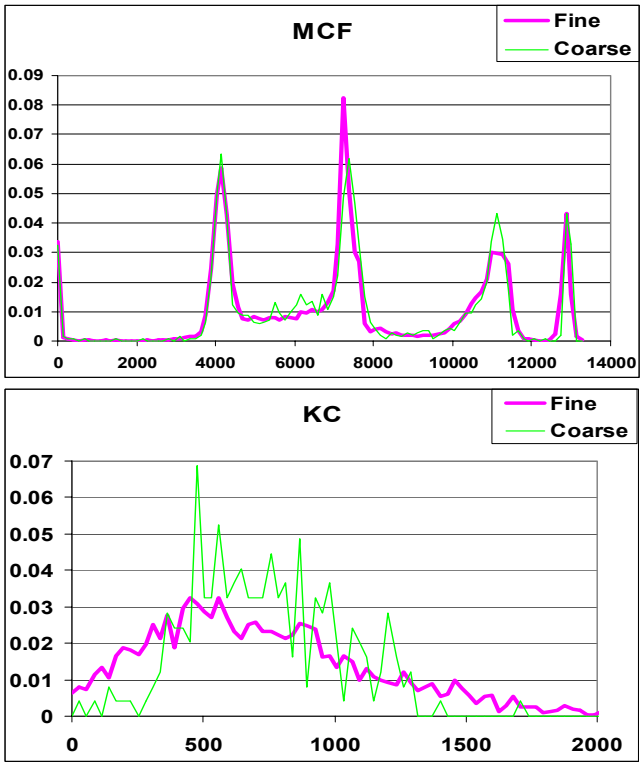

Figure 7: Histograms of M2 access throughputs. Each subfigure evaluates one benchmark where M2 comprises 75% of the total memory and has two curves—fine and coarse. The fine curve shows what is extracted and observed from a VMM 10Hz sampling. The coarse curve shows what is extracted from a VMM 1Hz sampling but uniformly distributed inside its 1-second period. X-axis is M2 page access counts during each 0.1-second interval. Y-axis reports the number of intervals during which certain M2 access counts are observed, normalized to the total number of intervals.

D. *Validation with Prior Results*

For validation of our platform, we compare our observed memory access profiles to those previously published. For example, the signature behavior of the working set size of deal2 (from SPEC CPU2006) reported in [11] roughly matches the M2 page access rate profile obtained from our experiment (Figure 8). First, it is apparent the memory access profile is dominated by the application itself instead of other software components. Second, it is consistent that M2 access count matches working set size, given the high M2 fraction (75%) and random M1 replacement policy. Finally, note that the last three major phases are prolonged in the bottom subfigure. These exactly correspond to situations where working set size exceeds the M1 capacity of a hybrid memory (512MB × 25% = 128MB), causing significant paging into M2, and a corresponding elongation of the execution time due to the limited M2 access rate of our platform.

VII. Related Work

An early virtualization-based performance evaluation was investigated on the IBM VM/370 system [3]. A key

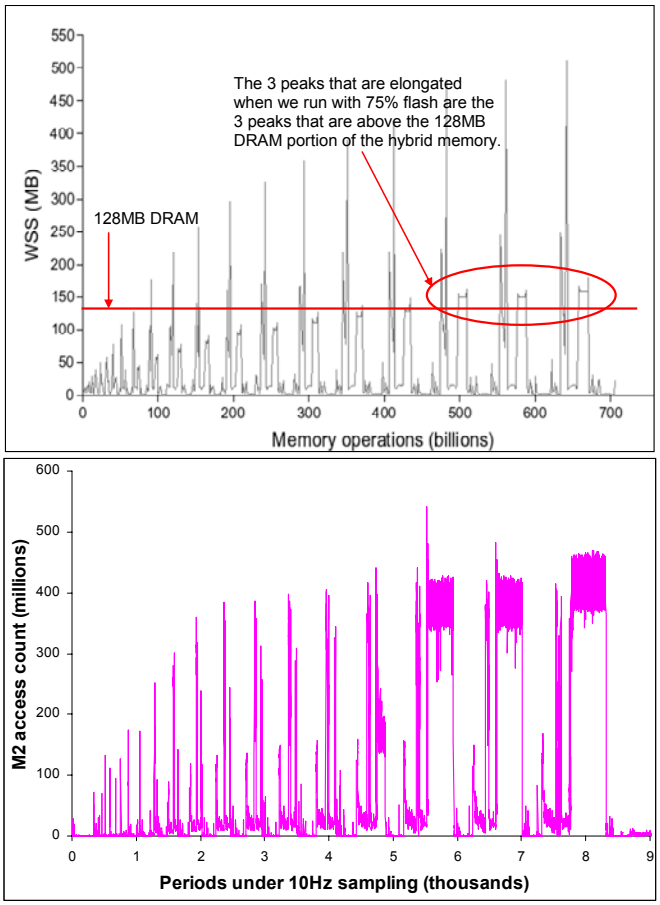

Figure 8: Working set size profile of deal2 reported in [11] superimposed with our comments (top) and obtained in our experiment (bottom). The experimental VM has 512MB of total memory, 75% of which is M2.

difference was the use of virtual time, independent of the real wall-clock time. The progress of the virtual time was controlled through a virtual processor speed parameter. Memory subsystem performance was lumped into the coarse-grained virtual processor speed. Hence, the performance of a workload reported in virtual time is strongly influenced by the accuracy of the virtual processor speed, which is itself workload-dependent. This circular dependence presents challenges when the virtual processor speed cannot be estimated reliably. Still, use of virtual time would broaden the scope of VMM-based performance evaluation.

The mechanism of our VMM-based simulation resembles trap-driven memory simulation [24]. Unlike conventional simulation where each memory access has to be evaluated, only misses in the simulated cache are evaluated in trap-driven simulation. The simulator ensures hardware traps are always installed on the memory locations associated with the simulated cache miss addresses. Simulation is done in a kernel trap service routine where it recalculates associations between memory locations and traps and records statistics. Special care must be taken to avoid setting traps on code and data of the simulator itself, as it would complicate the interpretation of statistics collected.

Analytical studies have previously suggested more levels in a main memory hierarchy to approach more cost-effective combinations [16, 19, 2, 13]. The two-level memory proposed in [8] encompasses the memory system studied here. The Simics full system simulator [17] was used to simulate the two-level main memory to obtain first-level miss counts. Execution time is estimated by adding the product of the miss counts and the second-level access latency to the baseline time. Complete runs of large workloads are nearly impractical due to simulation slowdown. The authors stated that it took several weeks to bring a simulated machine into the steady-state execution phase.

Multi-level main memories have been implemented in real systems [6, 5] and research prototypes [15]. In Multics, a strategy to treat core memory, drum, and disk as a three-level system was proposed [6]. The IBM 3090 system provided an expanded memory in addition to its main memory [5]. Similar to our conceptual architecture, data in the expanded memory must go through main memory to become accessible to the CPU. In [15], a research hardware system was built and evaluated. It featured a primary memory bus connecting to fast but low-capacity primary memory modules and a secondary memory bus connecting to high-capacity but 2x slower memory modules.

Prior research introducing alternative memories (NAND flash in particular) into the memory hierarchy concentrated on finding a narrower usage domain (*e.g.*, file buffer cache between DRAM and disk) to avoid expected performance degradation, while achieving other benefits such as power savings [18, 4, 14, 25]. In [18, 4], the designs were evaluated through off-line trace-driven simulation. In [14], the design was evaluated using full system simulation. In [25], the design was implemented in a custom Linux kernel. Performance and power characteristics were derived from measurement based on complete systems.

VIII. Conclusion and Future Work

In summary, we explore using a VMM for the performance evaluation of new computer designs. Using a customized VMM, we investigate some performance trends of hybrid main memories. We find that only certain workloads are amenable to hybrid main memory systems when total memory size is tightly provisioned. However, for all of our workloads, we show that a hybrid memory system with M2 latency of up to 40µs reduces performance by less than 10%, as long as DRAM is apportioned for at least 75% of the working set. This is a reasonable configuration, since today's real-world systems are likely to be over-provisioned. We also evaluate the effectiveness of some organization choices (such as write buffer and caching scheme) in mitigating the adverse impact implied by some suboptimal characteristics of potential M2 memories. Most importantly, we validate the VMM-based performance evaluation technique, finding that it offers both evaluation speed typical of hardware prototyping and evaluation flexibility typical of software simulation, with modest incremental development effort.

As future work, we plan to extend our study to SMP VMs with multiple virtual CPUs. We are also conducting research on algorithms which adapt to hybrid main memories and plan to investigate their effectiveness on our platform.

ACKNOWLEDGEMENTS

The simulation code was written when Dong Ye interned at VMware, and the data was collected by Brian Tsang at Rambus. We are grateful to David Kaeli for his generous support. Thanks to Alex Garthwaite and Rajesh Venkatasubramanian for their technical help and discussions. Thanks to Beng-Hong Lim, Mendel Rosenblum, Ken Barr, Peter Desnoyers, Irfan Ahmad and Rajit Kambo for their comments and suggestions. Thanks to Yiu Cho Lau for his automation scripts and the VMware performance group for their help. We appreciate Gary Bronner, Brent Haukness, William Ng, Eric Linstadt, Ian Shaeffer, and Mark Horowitz at Rambus for their expertise in memory technologies and insightful comments. Thanks to Yogesh Verma for helping set up the simulation systems.

REFERENCES

[1] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–13, 2006.

[2] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Marc Snir. A Model for Hierarchical Memory. In *STOC '87: Proceedings of the 19th Annual ACM Conference on Theory of Computing*, pp. 305–314.

[3] M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriquez-Rosell. A Virtual Machine Emulator for Performance Evaluation. In *SOSP '79: Proceedings of the 7th ACM Symposium on Operating Systems Principles*, 1979.

[4] Feng Chen, Song Jiang, and Xiaodong Zhang. SmartSaver: Turning Flash Drive into a Disk Energy Saver for Mobile Computers. In *ISLPED '06: Proceedings of the International Symposium on Low power Electronics and Design*, pp. 412–417, 2006.

[5] Edward I. Cohen, Gary M. King, and James T. Brady. Storage Hierarchies. *IBM Systems Journal*, 28(1):62–76, 1989.

[6] Fernando J. Corbató, Jerome H. Saltzer, and Chris T. Clingen. Multics – The First Seven Years. In *SJCC '72: Proceedings of the AFIPS Spring Joint Computer Conference*, 1972.

[7] Magnus Ekman and Per Stenstrom. A Case for Multi-Level Main Memory. In *WMPI '04: Proceedings of the 3rd Workshop on Memory Performance Issues*, pp. 1–8, 2004.

[8] Magnus Ekman and Per Stenstrom. A Cost-Effective Main Memory Organization for Future Servers. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*.

[9] eTesting Labs. Business Winstone 2002. Website, 2002. www.realworldtech.com/page.cfm?ArticleID=RWT101801000816.

[10] Dominic Giles. Swingbench Benchmark. Website, 2007. http://dominicgiles.com/swingbench.html.

[11] Darryl Gove. CPU2006 Working Set Size. *SIGARCH Computer Architecture News*, 35(1):90–96, 2007.

[12] ITRS. International Technology Roadmap for Semiconductors Report 2007 Edition. http://www.itrs.net/Links/2007ITRS/Home2007.htm.

[13] Bruce L. Jacob, Peter M. Chen, Seth R. Silverman, and Trevor N. Mudge. An Analytical Model for Designing Memory Hierarchies. *IEEE Transactions on Computers*, 45(10):1180–1194, 1996.

[14] Taeho Kgil, David Roberts, and Trevor Mudge. Improving NAND Flash Based Disk Caches. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pp. 327–338.

[15] Kai Li and Karin Peterson. Evaluation of Memory System Extensions. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 84–93, 1991.

[16] Yeong S. Lin and Richard L. Mattson. Cost-Performance Evaluation of Memory Hierarchies. *IEEE Transactions on Magnetics*, 8(3):390–392, 1972.

[17] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav H, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[18] B. Marsh, F. Douglis, and P. Krishnan. Flash Memory File Caching for Mobile Computers. In *HICSS-27: Proceedings of the 27th Hawaii Conference on Systems Science*, pp. 451–461, 1994.

[19] Satish L. Rege. Cost, Performance and Size Tradeoffs for Different Levels in a Memory Hierarchy. In *ISCA '76: Proceedings of the 3rd Annual Symposium on Computer Architecture*, pp. 64–67D, 1976.

[20] Mendel Rosenblum and Tal Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, 38(5):39–47, May 2005.

[21] Timothy Sherwood and Joshua J. Yi. Guest Editors' Introduction: Computer Architecture Simulation and Modeling. *IEEE Micro*, 26(4):5–7, 2006.

[22] SPEC. SPECjbb2005 Suite. Website. http://www.spec.org/jbb2005.

[23] SPEC. CPU2006 Suite. Website. http://www.spec.org/cpu2006.

[24] Richard Albert Uhlig. *Trap-Driven Memory Simulation*. PhD thesis, EECS Dept., University of Michigan, Ann Arbor, MI. 1995.

[25] Luis Useche, Jorge Guerra, Medha Bhadkamkar, Mauricio Alaron, and Raju Rangaswami. EXCES: EXternal Caching in Energy Saving Storage Systems. In *HPCA-13: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2008.

[26] VMware. Timekeeping in VMware Virtual Machines. Website, 2005. http://www.vmware.com/pdf/vmware_timekeeping.pdf.

[27] VMware. Introduction to VMware Infrastructure. Website, 2007. www.vmware.com/pdf/vi3_35/esx_3/r35/vi3_35_25_intro_vi.pdf.

[28] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.