A High-Performance Oblivious RAM Controller on the Convey HC-2ex Heterogeneous Computing Platform

Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari[‡],
Elaine Shi[†], Krste Asanović, John Kubiatowicz, Dawn Song

University of California, Berkeley [‡] University of Texas, Austin [†] University of Maryland, College Park

ABSTRACT

In recent work published at *ACM CCS 2013* [5], we introduced PHANTOM, a new secure processor that obfuscates its memory access trace. To an adversary who can observe the processor's output pins, all memory access traces are computationally indistinguishable (a property known as obliviousness). We achieve obliviousness through a cryptographic construct known as Oblivious RAM or ORAM.

Existing ORAM algorithms introduce a fundamental overhead by having to access significantly more data per memory access (i.e. cache miss): this overhead is 100-200× or more, making ORAM inefficient for real-world workloads. In PHANTOM, we exploit the Convey HC-2ex heterogeneous computing platform – a system consisting of an off-the-shelf x86 CPU paired with 4 high-end FPGAs and a highly parallel memory system – to reduce ORAM access latency.

We present a novel ORAM controller that aggressively exploits the HC-2ex's high DRAM bank parallelism to reduce ORAM access latency and scales well to a large number of memory channels. PHANTOM is efficient in both area and performance: accessing 4KB of data from a 1GB ORAM takes 26.2us (13.5us until the data is available), a 32× slowdown over accessing 4KB from regular memory, while SQLite queries on a population database see 1.2-6× slowdown.

1. INTRODUCTION

Confidentiality of data is a major concern for enterprises and individuals who wish to offload computation to the cloud. In particular, cloud operators have physical access to machines and can observe sensitive information (data and code) as it moves between a CPU and physical memory [4, 12]. To protect against such attacks, prior work has proposed secure processors [9, 10] that automatically encrypt and integrity-check all data outside the processor, whether in DRAM or non-volatile storage (this model is now starting to appear in commercial products such as Intel's SGX extensions or IBM's cryptographic coprocessors).

Although secure processors encrypt memory contents, off-the-shelf DRAMs require that *memory addresses* be transmitted over the memory bus in cleartext. An attacker with physical access can install memory probes to snoop the address bus (e.g. through malicious NVDIMMs) to observe the locations of RAM accessed [4] and in turn learn sensitive data such as encryption keys or information about user-level programs [12] and guest VMs in a virtualized server.

Preventing such information leakage requires making memory address traces indistinguishable, or *oblivious*. At *CCS 2013*, we proposed a new hardware architecture for efficient oblivious computation that ensures both *data confidentiality* and *memory trace obliviousness*: an attacker snooping the memory bus and the DRAM contents cannot learn anything about the secret program memory – not even the physical memory locations accessed. We introduced PHANTOM [5], a new secure processor that achieves this goal. PHANTOM is prototyped on the Convey HC-2ex heterogeneous computing platform, a system pairing an off-the-shelf x86 CPU with 4 FPGAs and a highly parallel memory system (Figure 1).

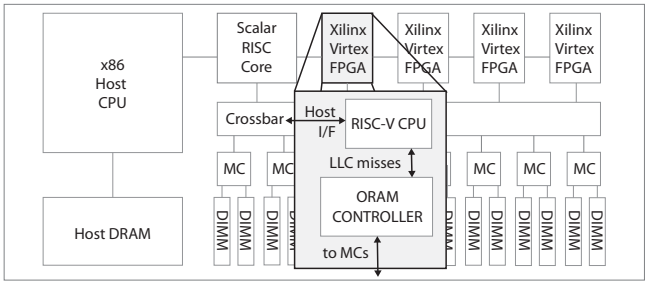

Figure 1: The Convey HC-2ex with our PHANTOM *prototype. The prototype comprises a CPU, non-volatile memory, and an ORAM controller implemented on one of the FPGAs. All* digital *signals outside the FPGA are assumed to be visible to the adversary.*

Oblivious RAM: We use an algorithmic construct called Oblivious RAM (ORAM). Intuitively, ORAM techniques obfuscate memory access patterns through random permutation, reshuffling, and re-encryption of memory contents, and require varying amounts of *trusted memory* that the adversary cannot observe. To develop a practical ORAM in hardware, we adopt Path ORAM [8] – a simple algorithm with a high degree of memory access parallelism.

Other recent work has also used Path ORAM to propose a secure processor (ASCEND [2, 6]); this work focussed on optimizing the Path ORAM *algorithm* while we focus on the *microarchitecture* of a practical oblivious system. As such, ASCEND is complementary to our work on PHANTOM.

Challenges: Path ORAM imposes a significant memory bandwidth overhead – more than 100× over a non-secure access. Furthermore, Path ORAM's irregular, data-driven nature makes it difficult to simply add more memory channels and build a deterministic yet efficient ORAM controller. Finally, we do not propose a custom chip but rely on an off-the-shelf FPGA platform that, on the one hand, can provide high memory bandwidth but on the other hand restricts us to use a slow FPGA for the ORAM controller logic (the ratio of slow logic to high memory bandwidth makes the problem of scaling to more memory channels even harder).

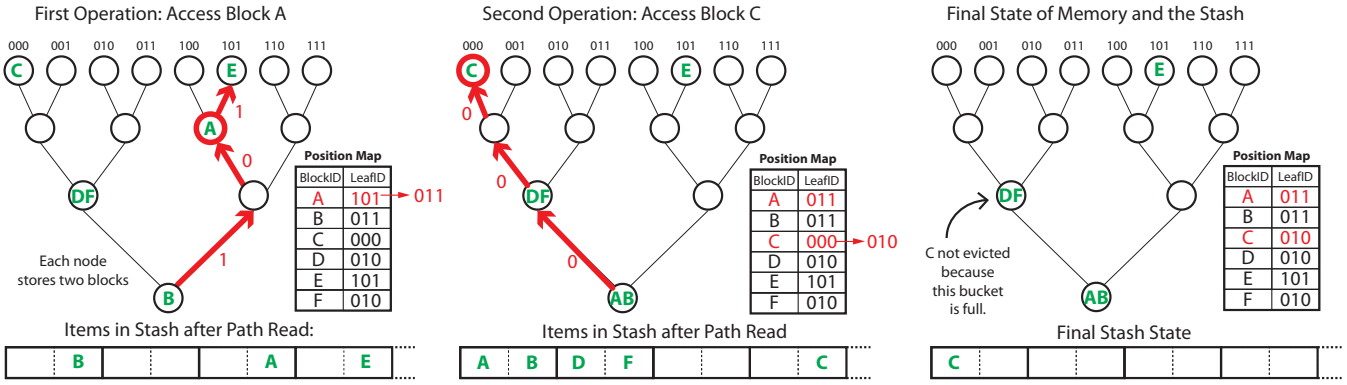

Figure 2: The Path ORAM Algorithm, demonstrated on two subsequent accesses. The first is a read of Block A, which the position map shows must be located somewhere on the path to leaf 101. Reading this path results in blocks B and E also being read into the stash. Block A is then randomly reassigned to leaf 011, and is therefore moved to the root of the path as it is being written back, since this is as far as it can now be inserted on the path to 101. Next, block C is read. Since the position map indicates that it is on the path to leaf bucket 000, that path is read, bringing blocks A, B, D, and F into the Stash as well. C is reassigned to leaf 010 and the bucket containing D and F is already full, so it can only be in the root of the path being written back. However, A and B must also be in the root as they cannot be moved any deeper, so C cannot be inserted. It therefore remains in the stash beyond the ORAM access.

Contributions: 1) We introduced an ORAM controller architecture that effectively utilizes high-bandwidth DRAM, even when implemented on slow FPGA logic. 2) We proposed critical improvements (and characterization) of the Path ORAM algorithm, and a deeply pipelined microarchitecture that utilizes 93% of the maximum DRAM bandwidth from 8 parallel memory controllers. 3) We built and evaluated PHANTOM on a Convey HC-2ex (running SQLite workloads), and simulated its performance for different cache sizes. The prototype sustains 38,191 full 4KB ORAM accesses per second to a 1GB ORAM, which translates to 1.2× to 6× slowdown for SQLite queries.

To the best of our knowledge, this is the first practical implementation of an oblivious processor.

2. USAGE & ATTACK MODEL

We consider a scenario where a cloud provider offers a secure processor, produced by a trusted hardware manufacturer, and remote clients can establish an authenticated connection with a *loader* program on the processor and transfer an encrypted ORAM image with sensitive data and code to the processor's memory. The loader then executes the code obliviously and stores the results back into ORAM. The remote client can collect the encrypted results once the time allocated for the computation has elapsed.

We aim to protect against attackers with physical access to the machine (such as malicious data center employees, intruders or government-mandated surveillance). Such attackers can snoop DRAM traffic by e.g. installing malicious DIMMs or probing the memory bus. While existing secure processors and solutions such as Intel SGX prevent explicit data leakage with encryption, we prevent *implicit* information leakage through the address bus. Specifically, we provide the property that "for any two data request sequences of the same length, their access patterns are computationally indistinguishable by anyone but the client" (from [7]).

Note that the *total* execution time (a termination channel) is out of scope for ORAM. Information leaks through this channel can be countered by normalizing to the worst case execution time for a program. Further, the timing of *individual* ORAM accesses does not leak information if PHANTOM is deployed such that a non-stop stream of DRAM traffic is maintained. Cache hits inside the CPU or the ORAM controller would not alter the pattern of DRAM accesses observable and only reduce the execution time (i.e. timing channels are reduced to a termination channel).

Our current implementation does not ensure *integrity* of data, but the Path ORAM tree can be treated as a Merkle tree to efficiently provide integrity with freshness [8]. We do not consider software-level digital attacks, or analog attacks that exploit the physical side-effects of computation (such as temperature, EM radiation, or even power draw).

3. THE PATH ORAM ALGORITHM

Path ORAM [8] prevents the information leakage through memory addresses by reshuffling contents of untrusted memory after each access, such that accesses to the same location cannot be linked (while also re-encrypting the accessed content with a different nonce at every access). We assume the secure processor has a small amount of trusted (in our case, on-chip) memory, which the ORAM controller can access without revealing any information to the attacker. Path ORAM ensures that all that is visible to an attacker is a series of random-looking accesses to untrusted memory.

Data is read and written in units called *blocks*. All data stored by an ORAM instance is arranged in untrusted memory as a binary tree, where each node contains space to store a few blocks (usually four). When a request is made to the ORAM for a particular block, the controller looks up the block in a table in trusted memory called the *position map*. In the position map, each block is assigned to a particular *leaf node* of the tree, and Path ORAM guarantees the invariant that each block is resident in one of the nodes along the path from the root to the block's designated leaf. Reading this entire path into the *stash* – a data structure that stores data blocks in trusted memory – will thus retrieve the desired block along with other blocks on the same path.

After the requested block is found and its data returned to the requester (e.g. a CPU), the ORAM controller writes the *same path* back to memory. All slots of the tree that do not contain a block are filled with *dummies*, which contain no actual data (and are never put into the stash) but are encrypted in the same way as blocks so that their ciphertext is indistinguishable from that of a block.

At every access, the requested block is reassigned to a random leaf node and hence may belong to a different path from that on which it was read (and has to be put into the upper levels of the tree to not violate the invariant). If no additional steps were taken, the upper levels of the tree would thus quickly become full. Path ORAM therefore has to move blocks in the stash as deep as possible towards the leaf of the current path – this is called *reordering*.

Even with reordering, there can be cases where not all blocks in the stash can be written back (Figure 2). This is addressed by making the stash larger than a path worth of blocks. Blocks that cannot be written back remain in the stash and are carried over into the next ORAM access and handled the same as if they had been read then.

The obliviousness of Path ORAM stems from the fact that blocks are reassigned to random leaf nodes every time they are accessed: repeated accesses to the same block will appear as accesses to a random sequence of paths through the tree (each of which consists of a full read followed by a full write of the same path). This is also the reason the writeback has to be to the same path: if the path was dependent on the *new* leaf of the block, accesses to the block could be linked.

Stash overflows: The stash may *overflow* (no more blocks fit into the stash). Path ORAM can recover from overflows by reading and writing random paths and try to evict blocks from the stash during those path reads and writes. While this does not leak information (the random path accesses are indistinguishable from regular ORAM accesses), it increases execution time and may hence cause execution to not finish in the allotted time. It is therefore desirable to size the stash in such a way that these accesses occur rarely. In [5] we present an empirical analysis to determine stash sizes that makes these overflows extremely unlikely.

Access timing: To avoid information leakage through memory access timing, Path ORAM can perform a non-stop sequence of path reads and writes, accessing a random path if there is no outstanding ORAM request from the CPU. Stash hits can be hidden by performing a fake path access as well, and multiple stash hits can be hidden behind the same access. This is orthogonal to our work (and hence not implemented) but would be required in a real deployment.

Fundamental Overheads: Path ORAM's obliviousness has both space and bandwidth costs. Only 50% of the physical memory is available as oblivious memory. Furthermore, as each block access results in an entire path read and write, Path ORAM's bandwidth overheads range from 104× for a 13-level ORAM tree (64MB capacity with 4KB blocks) to 192× for a 24-level tree (4GB capacity with 128B blocks). Hence, the task of building a low-latency ORAM system is challenging: such a system will have to read two orders of magnitude more data per memory access.

4. PHANTOM'S ARCHITECTURE

We chose to approach the fundamental overheads of Path ORAM by targeting platforms that have a very wide memory system (such as the HC-2ex). By employing a 1,024b-wide memory system, we can achieve a potential speed-up of 8× over an implementation using a conventional 128b-wide system. However, exploiting the higher memory bandwidth is non-trivial: it is necessary to co-optimize the hardware implementation and the Path ORAM algorithm itself.

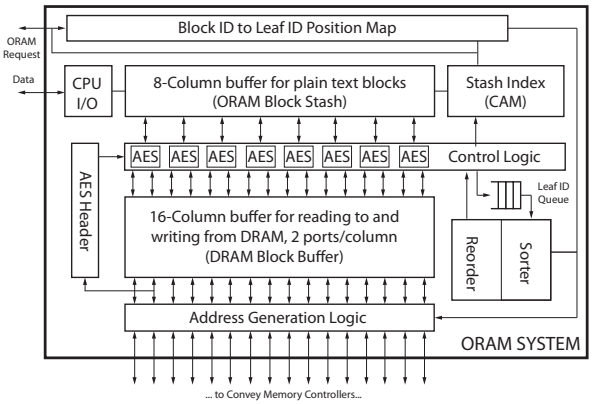

Figure 3: Overview of the ORAM system.

4.1 Achieving High Performance

Memory layout to improve utilization: Simply using a wide memory bus does not yield maximum DRAM performance: bank conflicts lead to stalls and decrease DRAM bandwidth utilization. To resolve them, we lay out the Path ORAM tree in DRAM such that data is striped across memory banks, ensuring that *all* DRAM controllers can return a value almost every cycle following an initial setup phase. Fully using eight memory controllers now makes the ORAM controller logic on the FPGA the main bottleneck.

Picking Blocks for Writeback: Bringing 1,024b per cycle into PHANTOM raises performance problems: the operation of the Path ORAM algorithm now has to complete much faster than it did before, to keep up with memory. While encryption can be parallelized by using multiple AES units in counter mode, the Path ORAM algorithm still has to manage its stash and decide which blocks from the stash to write back (the reordering step from Section 3).

The latter is of particular importance: The ORAM controller has to find the block that can be placed deepest into the current path, and do so while processing 1,024b per cycle. One approach would be to scan the entire stash and pick a block for every position on the path, in parallel with writing to memory. However, with Convey's high memory bandwidth, scanning through the stash takes longer than writing out an ORAM block, causing this approach to achieve less than half the potential performance for a stash size of 128 blocks. Also, to keep the stash small, it is crucial to select each block from the *entire* stash [5] – an approach that only considers e.g. the top blocks of the stash does not suffice.

We hence propose an approach that splits the task of picking the next block into two phases: a *sorting phase* that sorts blocks by how far they can be moved down the current path (XORing their leaf with the leaf ID of the current path), and a *selection stage* that (during bottom-up writeback) looks at the last block in the sorted list and checks in one cycle whether it can be written into the current position – if not, no other block can, and we have to write a dummy.

We further improve on this approach by replacing the sorting and selection stages by a min-heap (sorted by the current path's leaf ID). This replaces an $O(C \log C)$ operation by multiple $O(\log C)$ operations (each of which completely overlaps either with a block arriving from or being written to memory), where C is the size of the stash. This makes it now possible to overlap sorting completely with the path read and selecting with the path write phase.

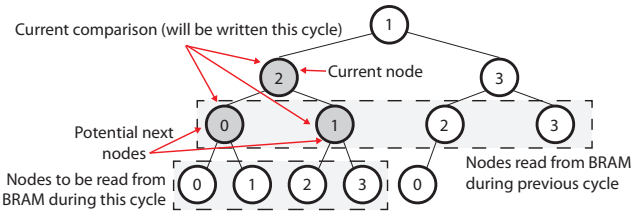

Figure 4: Min-heap implementation. The numbers represent the BRAM each node is stored in. The four second-degree children of a node are put into four BRAMs to access them in parallel.

Stash Management & Treetop Caching: The stash is a store for ORAM blocks, but it can also be used to improve performance by securely caching blocks on-chip. We can cache the top levels of the ORAM tree in the stash (we call this *treetop caching*) which avoids fetching these parts of the path from DRAM. Since the number of nodes is low at levels close to the root, this improves performance significantly while using only modest amounts of trusted memory.

We designed PHANTOM's stash management to support treetop caching with minimal effort (as well as other methods, such as LRU caching) by using a content-addressable memory (Xilinx XAPP 1151) that serves as lookup-table for entries in the stash, but is also used as directory for caching and as free-list to find empty slots in the stash. Reusing the stash for caching eliminates copying overheads that we saw in an early prototype that placed the cache separately.

4.2 Preserving Security

Design principles for obliviousness: We use two simple design principles to ensure that PHANTOM's design does not break Path ORAM's obliviousness guarantees. Any operation – checking the position map, reordering, caching etc. – that depends on ORAM data is either a) statically fixed to take the worst-case time or b) is overlapped with another operation that takes strictly longer. PHANTOM's decrypt operation could, for example, be optimized by not decrypting dummy ORAM blocks – but this leaks information since it would cause an operation to finish earlier depending on whether the last block was a dummy or not. Instead, PHANTOM pushes dummy blocks through the decryption units just the same as actual data blocks. These two design principles yield a completely deterministic PHANTOM pipeline.

Terminating timing channels at the periphery: The DRAM interface requires further attention to ensure security. PHANTOM sends path addresses to all DRAM controllers in parallel, but these controllers do not always return values in sync with each other. Although DRAM stalls do not compromise obliviousness (DRAM activity is not confidential), propagating these timing variations into PHANTOM's design can make timing analysis complicated. We therefore introduce buffers at the DRAM interface to isolate the rest of PHANTOM's ORAM controller from timing variations in the memory system. At the same time, all inputs to the DRAM interface and their timing are public (a leaf ID and 1,024b of encrypted data per cycle during writeback), so that no information can be leaked out of PHANTOM.

4.3 Implementation on the Convey HC-2ex

We prototyped PHANTOM on a Convey HC-2ex. Creating a full RTL-implementation (rather than a high-level simulation) was necessary to learn about the details that need

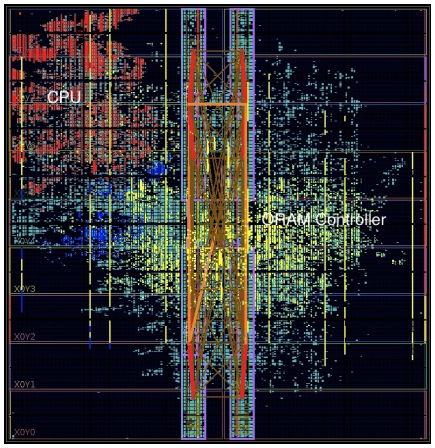

Figure 5: Synthesized design on a Virtex-6 LX760 FPGA.

to be considered in a real Path ORAM controller. Many of them are not apparent from the high-level description of the algorithm, such as the importance of how to reorder blocks, how to organize the stash, how to arrange data in (real) DRAM, what meta-data needs to be stored with each block and when to update it (e.g. we found that blocks should store their own leaf ID to decrease pressure on the position map, and that it needs to be updated during the read, not the write phase). Hence, without building a full hardware-implementation, it is not clear what to simulate (while RTL simulation took more than 30s per ORAM access).

PHANTOM is implemented on one of the HC-2ex's four Xilinx Virtex-6 LX760 (Figure 5), but some design points use up to two more FPGAs to store the position map. It consists of our custom designed ORAM controller and a single-core in-order RISC-V [11] CPU developed in our group. The entire project was implemented in Chisel [1], a new hardware description language developed at UC Berkeley.

Due to constraints in the Convey system, the ORAM controller had to run at 150 Mhz. Since this frequency was too high for the CPU (which was originally designed for ASICs), we put it into a second 75 Mhz clock domain available on the Convey architecture, and use asynchronous FIFOs to connect it to the ORAM controller. Nonetheless, without rearchitecting part of the uncore, we are only able to run with cache sizes of 4KB (IC), 4KB (DC) and 8KB (LLC), while still overclocking the circuit slightly (synthesis results differ between design points, but a representative example has 73/142 Mhz for the respective clock domains). We hence simulated larger cache sizes to get more realistic numbers.

Running the ORAM controller at this frequency posed a number of challenges. For example, the position map took up a large percentage of the FPGAs' BRAMs, which added a significant amount of routing pressure (and required us to pipeline accesses to the position map by 4 levels, as well as using design partitioning). Many other challenges were on the microarchitectural level and include the following:

Heap Implementation: For some heap operations, a node must be compared to both of its children (while also being written in the same cycle). This potentially doubles the access latency to the heap, since each of the FPGA's BRAMs has only one read and one write port. It would be possible to split the heap into two memories (one for even nodes, the other for odd nodes) so that the two children of a node are always in different BRAMs and can be accessed in the

same cycle – this is no more difficult than implementing a dual-ported memory by duplicating BRAMs, but does not double the required memory. However, the resulting circuit still contains paths from one BRAM through multiple levels of logic to another BRAM, leading to a long critical path. We therefore split the memory even further: our final implementation uses 4 different BRAMs (Figure 4). At every cycle, we prefetch the four grandchildren of a node so that the two children we are interested in will be available in buffer registers at the point when they are required for comparison, whichever way the previous comparison goes.

Stash Implementation: Each BRAM is limited to 2 ports. However, BRAMs in Phantom that constitute the stash have to be multiplexed among four functional units (encryption, decryption and reads/writes by the secure CPU). We designed Phantom such that all the units that read from or write to the stash are carefully scheduled such that only a single read port and a single write port on the BRAM is in use at any particular clock cycle.

4.4 Experiences with the Infrastructure

Convey HC-2ex: Convey provides a development kit for the HC-2ex which includes infrastructure IP and a set of scripts for the Xilinx design tools that synthesize and package FPGA images into a *personality* (a package that is automatically loaded by the Convey runtime to reprogram the FPGAs for a particular purpose). Personality-specific logic is implemented in a Verilog module that interfaces with the rest of the system (such as the memory controllers) and the host CPU communicates with personalities through a *management processor* (MP) that is connected to the FPGAs and runs code supplied by the host. It can access shared registers that are part of each personality and has special instructions that are directly dispatched to the personality. The Convey compiler bundles MP code (written in assembly) with the host binary, and host applications can perform *copcalls* to run a function on the MP.

Phantom is implemented as a personality and uses these mechanisms to establish a two-way channel between the host CPU and the RISC-V core on the FPGA: we provide copcalls that access a set of shared registers, which we connect to the host-target interface of the RISC-V *uncore*. This allows us to exchange commands with the RISC-V system (such as accessing control registers or writing to memory). Furthermore, the uncore's memory interface is connected to our ORAM controller, which uses the Convey memory system. The RISC-V core itself executes a minimal OS kernel on top of which we run programs such as SQLite. On the host, we run a server that is responsible for loading programs and executing the core's syscalls for debugging purposes.

We found the HC-2ex to be well-suited for ORAM research due to the simple interfacing of host and FPGA. Further, the Convey memory system offers many opportunities for experiments, e.g. with parallel requests or recursive ORAM. One useful feature would have been to handle cache misses from the host CPU on the FPGA, since this would enable to run the program on the host CPU and send encrypted memory requests to an ORAM controller on the FPGA (but there appears to be no fundamental reason this could not be provided by the hardware). Another interesting (future) platform for ORAM research may be a hybrid-memory cube FPGA – this would enable larger position maps.

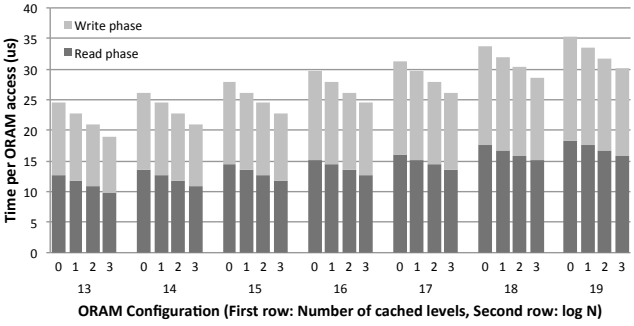

Figure 6: Average time per ORAM access for different configurations on hardware. 1M accesses each, block size 4KB.

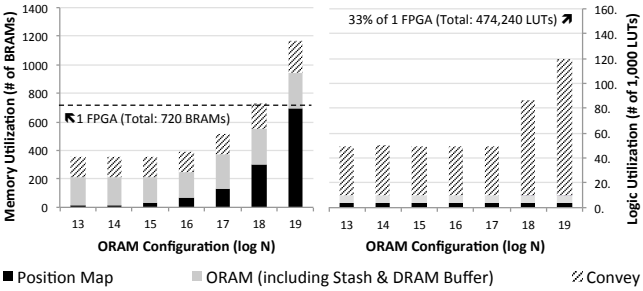

Figure 7: FPGA resource utilization. For sizes larger than 17 levels (1GB), the position map is stored on other FPGAs.

Chisel: Chisel is a hardware description language embedded in Scala. While it describes synthesizable circuits directly (similar to Verilog), it makes the full Scala programming language available for circuit generation (e.g. functional or recursive descriptions of circuits) and has additional features such as width inference and a type system for wires, support for structs, high-level descriptions of state machines and bulk wiring. It compiles to Verilog, which made it easy to interface with the Convey infrastructure.

Chisel facilitated the implementation of Phantom since it made it easier to e.g. generate parts of the design automatically based on design parameters and deal with complex control logic (in particular due to the interlocking in different parts of the pipeline). For example, to implement the heap (Section 4.3), we defined functions that transparently prefetch the right data from the different memories (hiding this complexity from the heap implementation).

5. EVALUATION

We experimentally determined the cost of obliviousness on Phantom and its performance impact on real applications.

ORAM Latency and Bandwidth: We synthesized a set of different configurations of Phantom, with effective ORAM storage capacity ranging from 64MB to 4GB (13-19 tree levels with a 4KB block size). Each ORAM configuration includes a stash of 128 elements, for up to 3 levels of treetop caching. For 18 and 19-level ORAMs, Phantom's position map is stored on one and two adjacent FPGAs (requests to them need to be encrypted and padded).

Figure 6 shows the total time per ORAM access for these configurations. We report the average times per access and how long it takes until the data is available (for reads), as the CPU can continue execution as soon as the data is returned from ORAM. For writes, it can continue immediately.

We measured that Phantom's latency until ORAM data is available ranges from 10us for a 13-level (64MB) ORAM to

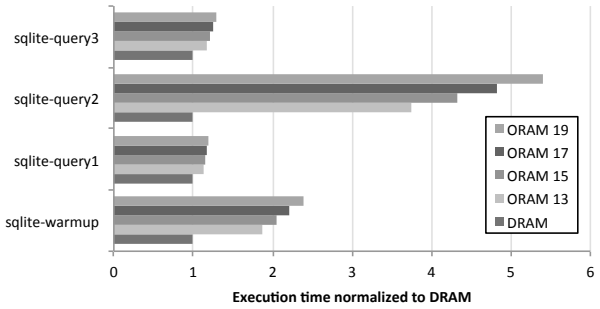

Figure 8: Simulated performance of sqlite on a timing model of our processor. We assume a 1MB L2 cache, 16KB icache, 32KB dcache and buffering 8 ORAM blocks (32KB).

16 us for a 19-level (4GB) ORAM. Compared to sequentially reading a 4kB (812ns) or a 128B (592ns) block of data[1] using all Convey memory controllers, this represents 12× to 27× overhead. An ORAM access that hits the stash takes 84 cycles (560ns). When considering full ORAM accesses, latencies range from 19us to 30us. Much of PHANTOM's real-world performance is therefore determined by how much of the write-back can be overlapped with computation, but is bounded above by 23× to 50× overhead.

We measured that PHANTOM utilizes 93% of the theoretical DRAM bandwidth (i.e. if each memory controller returned the maximum number of bits every cycle without any delays). As PHANTOM accesses no additional data beyond what is required by Path ORAM, this shows that we come very close to our goal of fully utilizing bandwidth.

FPGA Resource Usage: Figure 7 reports PHANTOM's hardware resource consumption through the number of LUTs used by the different configurations, as well as the number of BRAMs used on the FPGA. The design itself uses 30-52% of memory and about 2% of logic – the other resources are used by Convey's interface logic that interacts with the memory and the x86 core. These results do not include the resources that would be consumed by real AES crypto hardware. There exist optimized AES processor designs [3] that meet our bandwidth and frequency requirements while consuming about 22K LUTs and no BRAM – our nine required units would therefore fit onto our FPGA (as the ORAM controller leaves almost 90% of the FPGA's LUTs available).

Application Performance: We are interested in how the ORAM latency translates into application slowdowns. As an end-to-end example, we used our real RISC-V processor to run several SQLite queries on a 7.5MB census database on our real hardware. Due to the processor's extremely small cache sizes, a very large percentage of memory accesses are cache misses (we ran the same workloads on an ISA simulator with caching models and found 7.7% dcache misses, and 55.5% LLC miss rate, i.e. *LLC misses/LLC accesses* – we hypothesize that many of SQLite's data structures are too large to fit into an 8KB cache). As a result, we observed slow-downs of 6.5× to 14.7× for a set of different SQLite queries. This experiment shows how crucial caching is to achieve good performance in the presence of ORAM.

To investigate the effect on applications in the presence of realistic cache sizes (namely a 16KB icache, 32KB dcache and 1MB LLC), we ran the same applications on a timing model derived from our real processor to simulate how our system would perform with realistic cache sizes. The model assumes an in-order pipelined RISC-V processor with 1 cycle per instruction, 2 cycles for branch misses, 2 cycles to access the data cache, 14 cycles to access the LLC, and 89 cycles to access a 128B cache line from DRAM (using our measurement for the full Convey memory system) and our access latencies for different ORAM configurations.

We see that a real-world workload, such as SQLite, can absorb a fair amount of memory access overheads and make use of our relatively large block size of 4KB. Figure 8 shows our results for applying our timing model to several queries on the census database (details can be found in [5]). This shows that the overheads exposed by PHANTOM are indeed reasonable for real-world workloads, especially given the fundamental cost of obliviousness is a bandwidth overhead of greater than 100×. Note that these numbers depend on the application's LLC miss-rate and overheads are small because much of the application's working set fits in the cache.

[1]Since the granularity of ORAM accesses is much larger than a cache line, we compare to 128B reads to estimate the worst-case overhead where only a fraction of data is used.

6. CONCLUSION

We presented PHANTOM, a processor with a practical oblivious memory system that achieves high performance by exploiting memory parallelism on a Convey HC-2ex. As such, it enables obliviousness on hardware available today.

Acknowledgements: *For acknowledgements and funding information, please consult the CCS'13 paper [5].*

References

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing Hardware in a Scala Embedded Language," in *DAC*, 2012.

[2] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A Secure Processor Architecture for Encrypted Computation on Untrusted Programs," in *STC*, 2012.

[3] A. Hodjat and I. Verbauwhede, "A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA," in *FCCM*, 2004.

[4] A. Huang, "Keeping Secrets in Hardware: The Microsoft Xbox Case Study," in *CHES*, 2002.

[5] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanović, J. Kubiatowicz, and D. Song, "PHANTOM: Practical Oblivious Computation in a Secure Processor," in *CCS*, 2013.

[6] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas, "Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors," in *ISCA*, 2013.

[7] E. Stefanov, E. Shi, and D. Song, "Towards Practical Oblivious RAM," in *NDSS*, 2012.

[8] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path O-RAM: An Extremely Simple Oblivious RAM Protocol," in *CCS*, 2013.

[9] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing," in *ICS*, 2003.

[10] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *SIGOPS Oper. Syst. Rev.*, vol. 34, no. 5, pp. 168–177, 2000.

[11] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA," EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62, May 2011.

[12] X. Zhuang, T. Zhang, and S. Pande, "HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus," in *ASPLOS*, 2004.