VMT: Virtualized Multi-Threading for Accelerating Graph Workloads on Commodity Processors

Josué Feliu, Ajeya Naithani, Julio Sahuquillo, *Member, IEEE,* Salvador Petit, *Member, IEEE,* Moinuddin Qureshi, *Member, IEEE,* and Lieven Eeckhout, *Fellow, IEEE*

Abstract—Modern-day graph workloads operate on huge graphs through pointer chasing which leads to high last-level cache (LLC) miss rates and limited memory-level parallelism (MLP). Simultaneous Multi-Threading (SMT) effectively hides the memory access latencies for multi-threaded graph workloads provided that sufficient threads are supported in hardware. Unfortunately, providing a sufficiently large number of physical threads incurs an unjustifiably high hardware cost for commodity SMT processors which typically implement only two physical hardware threads. Ideally, we would like to achieve aggressive-SMT performance when running graph workloads on modest commodity processors.

In this paper, we propose Virtualized Multi-Threading (VMT), a low-overhead multi-threading paradigm for accelerating graph workloads on commodity processors. Unlike prior multi-threading paradigms, VMT virtualizes both the physical hardware threads and the architecture state: VMT maps a large number of logical software threads to a small number of physical hardware threads, while maintaining the architecture state of the logical threads in the processor's cache hierarchy. Implemented on top of a quad-core 2-way SMT processor, VMT achieves an average speedup of 1.74× for a set of representative graph workloads, while incurring minimal hardware cost (195 bytes per core to support up to 32 logical threads). VMT's low hardware cost paves the way for implementation in commodity processors.

Index Terms—Multi-Threading, Architecture State, Virtualization, Graph Workloads.

1 Introduction

Graph workloads have gained major interest from both industry and academy, primarily due to the increasing importance of social networks and other big data workloads [1], [2], [3], [4], [5], [6], [7], [8]. In addition, graph algorithms have found their way to solve scientific problems and to represent and understand unstructured data [9], [10], [11], [12]. Intrinsic graph characteristics make graph algorithms behave irregularly, which results in poor memory locality. This results in poor performance when running graph workloads on commodity superscalar processors. However, there is abundant thread-level parallelism (TLP) to be exploited in graph workloads. In this paper, we propose a novel and low-overhead multi-threading paradigm to significantly speed up graph workloads on commodity processors.

Multi-threading paradigms have been widely used to improve processor performance by exploiting TLP. Early computers deployed software multi-threading (or time-sharing) to hide I/O and storage (e.g., disk) latencies, i.e., these latencies were large enough to be hidden by software context switches. Unfortunately, software multi-threading is unable to hide idle times in the processor due to pipeline bubbles and cache/memory accesses. Hardware multi-threading hides these idle times by doing useful work from another thread while experiencing a latency-causing event. Coarse-grain multi-threaded (switch-on-event) processors [13], [14] execute one thread at a time to hide long latencies (such as memory accesses). Fine-grain multi-threaded processors [15] also execute one thread at a time while context switching every cycle to hide even short latencies. Simultaneous multi-threading (SMT) [16], the most widely deployed paradigm, can execute instructions from different threads in the same cycle to fully exploit the available superscalar issue bandwidth and further improve processor performance.

Unfortunately, all previous multi-threading paradigms incur significant hardware overhead to maintain the *architecture state*[1] of the concurrently executing threads [17]. Increasing the number of supported threads is challenging because the core needs to store the architecture state of all the threads that can run simultaneously. And the largest part of this state is in the register file, which must be accessible to the pipeline and its execution resources. Accessing such a bigger register file incurs a cost in complexity and can easily affect the processor cycle time. Consequently, high degrees of multi-threading are only supported in premium SMT processors for the high-end server markets, e.g., the

- *J. Feliu is with the Department of Computer Engineering, Universidad de Murcia, Spain. E-mail: josue.f.p@um.es*
- *A. Naithani and L. Eeckhout are with the Department of Electronics and Information Systems, Ghent University, Belgium.*
- *J. Sahuquillo and S. Petit are with the Department of Computer Engineering, Universitat Politécnica de Valéncia, Spain.*
- *M. Qureshi is with the Department of Computer Science, Georgia Tech, USA.*

Manuscript received MM XX, 2018; revised MM XX, 20YY.

1. The architecture state considered in this work is defined in Section 2.1.

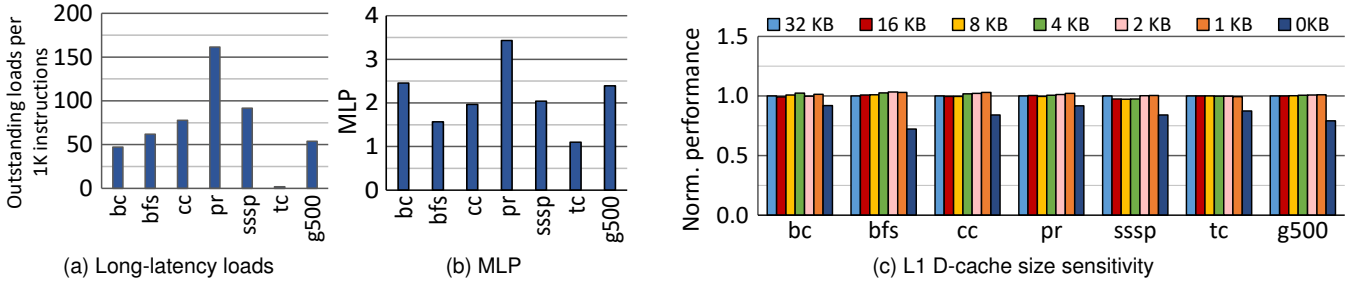

Fig. 2. Characterizing graph workloads in terms of (a) number of long-latency loads per 1K instructions, (b) MLP per thread, and (c) sensitivity to the L1 D-cache size. *Graph workloads feature a high number of long-latency loads and limited MLP, and almost no sensitivity to the L1 D-cache size.*

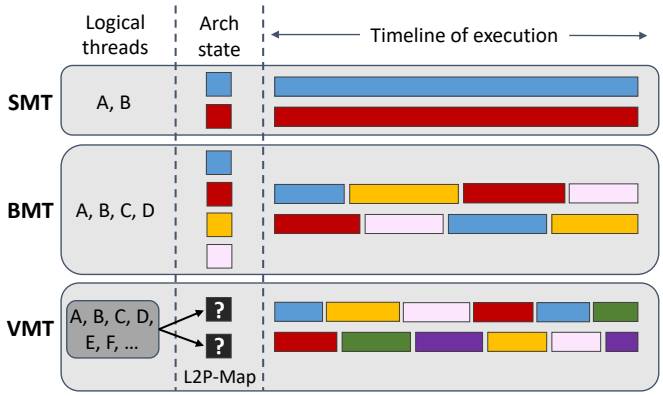

Fig. 1. Contrasting Virtualized Multi-Threading against SMT and Balanced Multi-Threading (BMT). *VMT virtualizes the physical threads and the architecture state, whereas BMT only virtualizes the physical threads, and SMT virtualizes neither the physical threads nor the architecture state.*

IBM POWER8 [18] and POWER9 support 8-way SMT. Commodity processors by Intel and AMD on the other hand feature a much lower degree of SMT, e.g., typically two SMT threads. The reason is the prohibitive hardware cost: in particular, IBM's POWER8 extends the register file with a second level of so-called Software Architected Registers to keep track of the entire architecture state when concurrently executing eight threads.

Supporting high degrees of multi-threading exacerbates the architecture state problem or requires modifications to software. In particular, Balanced Multithreading (BMT) [19] virtualizes the physical (hardware) threads. BMT requires a dedicated hardware structure to store the architecture state for all logical threads. This so-called 'inactive' register file incurs significant hardware overhead: 20.75 KB storage for 32 logical threads on the x86_64 instruction-set architecture; supporting recent vector extensions (e.g., AVX-512) increases the hardware cost to 68.75 KB per core, which is prohibitive for commodity processors. Informing loads [20] and co-routines [21] are software solutions to support high degrees of multi-threading; unfortunately, they require significant modifications to the source code and complex compiler optimizations to minimize switching latency.

This paper proposes *Virtualized Multi-Threading (VMT)*, a novel hardware multi-threading paradigm that virtualizes the architecture state by storing the architecture state of swapped-out logical threads in the (conventional) cache hierarchy, while requiring no changes to software. Because no dedicated structures are needed to maintain architecture state, VMT's hardware overhead is limited to 195 bytes per core while supporting up to 32 logical threads. Figure 1 illustrates how VMT virtualizes both the physical threads and the architecture state (i.e., the registers), which allows VMT to run a large number of threads without involving the OS. In contrast to VMT, SMT does not virtualize the physical threads, nor does it virtualize the architectural state, which leads either to support a very small number of threads or to incur a significant hardware cost. BMT virtualizes the physical threads but not the architecture state, thus requiring a (large) dedicated hardware structure to store the architecture state for all logical threads.

Virtualizing the architecture state in the cache hierarchy only really makes sense if it does not compromise the workload's memory behavior. We find this to be the case for graph workloads. Graph workloads align unfavorably with superscalar out-of-order processors: they suffer from high last-level cache (LLC) miss rates *and* limited memory-level parallelism (MLP) because of pointer chasing through huge graph structures, see Figures 2a and 2b, respectively. (Section 3 provides details about our experimental setup.) At the same time, there is abundant TLP to be exploited in graph workloads [22]. While these characteristics are well-known, we make the new observation that graph workloads are insensitive to L1 D-cache performance, see Figure 2c: reducing the L1 D-cache size from 32 KB to 1 KB does not degrade performance.[2] In other words, graph workloads do not benefit from a processor's L1 D-cache.

We exploit this key observation in the VMT proposal by virtualizing the architecture state in the processor's cache hierarchy. We find that VMT fits the characteristics of graph workloads particularly well, i.e., virtualizing the logical threads' architecture state in the cache hierarchy does not significantly interfere with the graph workload itself because of its inherently poor cache locality. Experimental results show that VMT significantly improves graph workload performance. For a quad-core 2-way SMT processor resembling a current commodity processor, VMT with 16 logical threads per core improves performance by 1.74× on average (and up to 3.17×) for a set of representative graph workloads including `graph500` [23] and the GAP benchmark suite [7].

In summary, this paper makes the following contributions:

- We propose virtualized multi-threading (VMT), a novel multi-threading paradigm to support high degrees of multi-threading (up to 32 threads) in a commodity core

2. Eliminating the L1 D-cache leads to a 16% average performance degradation because of stack accesses.

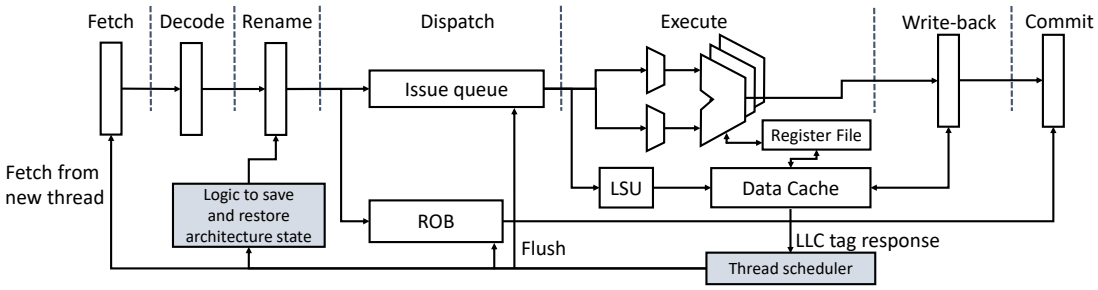

Fig. 3. VMT architecture overview. *A commodity core architecture needs to be extended with a thread scheduler and a small dedicated unit containing logic to insert load and store instructions to save and restore the logical threads' architecture state.*

at minimal hardware cost (195 bytes) by virtualizing the architecture state in the processor's cache hierarchy.
- We demonstrate that graph workloads fit VMT's architecture particularly well with average performance improvements by $1.74\times$, and up to $3.17\times$. Saving the architecture state in the cache hierarchy does not significantly affect the graph workloads' overall cache performance.
- We demonstrate that VMT's performance benefit comes from increased MLP. We further comprehensively evaluate VMT's mechanism, its performance overheads, and its robustness across input graphs. Finally, we demonstrate VMT's ability to speed up other parallel workloads, and we compare against a state-of-the-art indirect memory access prefetcher for graph workloads.

2 Virtualized Multi-Threading

Figure 3 provides an overview of the virtualized multi-threading (VMT) architecture. The figure shows the different stages and some of the main structures of the pipeline of a commodity core. VMT requires few extensions: a thread scheduler to orchestrate thread swapping plus logic to save and restore the architecture state of the logical threads. When a thread triggers an LLC miss, the thread scheduler initiates thread swapping: it flushes the long-latency load and subsequent instructions, saves the thread's architecture state, restores the architecture state of the incoming thread, and starts fetching instructions for this thread. VMT is particularly appealing for commodity processors with either single-threaded or SMT cores with limited degree of multi-threading, e.g., two-way SMT. VMT is enabled only when the workload benefits.

2.1 Virtualizing Architecture State

VMT's key feature is to save the architecture state of the logical threads in the processor's cache hierarchy, making the hardware cost to virtualize architecture state 'virtually' free. This is in sharp contrast to BMT [19] which requires a dedicated hardware structure. The architecture state per logical thread in current x86_64 architectures consists of 35 registers in total: 16 64-bits general-purpose registers, 16 256-bit floating-point/vector registers, and 3 64-bit special-purpose registers (i.e., program counter, flags register, and FPU status register). This amounts to 20.75 KB assuming 32 logical threads. For the recent AVX-512 extension, the number of floating-point/vector registers increases to 32 of 512 bits each. Overall, at most 2.15 KB is required to store the architectural state per thread.[3] Assuming 32 logical threads, this amounts to a total of 68.75 KB architecture state per core.

Note that we do not need to save all registers upon each thread swap, i.e., we only save registers that have been written since the last thread swap — this reduces the amount of cache space occupied and reduces the time overhead of the thread swap. Moreover, storing architecture state in the cache hierarchy works synergistically with graph workloads.

2.2 Thread Swapping

Quickly swapping threads after a long-latency memory request is key to achieve high performance. To this end, we extend current commodity (SMT) cores with a switch-on-event mechanism to swap out a thread that experiences a long-latency load instruction and swap in a thread whose memory request has already been completed. Implementing the thread swap operation in hardware without the intervention of the operating system (OS) enables fast migration of threads, thereby hiding most of the memory access latency.

Figure 4 illustrates how thread swapping affects reorder buffer (ROB) state. VMT exploits a key characteristic of graph workloads and seeks to initiate the thread swap as early as possible upon an LLC miss of the outgoing thread. The reason is that there is little MLP to be exploited within a single thread of execution. Starting the thread swap as early as possible advances the execution of the incoming thread and allows it to get to the next memory request faster, improving MLP.

VMT identifies a long-latency load miss by receiving an early miss reply upon an LLC tag lookup, which is propagated from the LLC to the core. This is illustrated in Figure 4a. The core reacts by flushing the load miss and all younger instructions in the ROB, while not canceling the in-flight memory request. This action can be carried out using the hardware that commodity cores implement to deal with misspeculated loads or mispredicted branches. In particular, the core rebuilds the Register Alias Table (RAT) as is done for mispredicted branches [24]. (See Section 3 for further details.)

3. This assumes that the logical threads of the same process share the extended thread state, which includes control registers (CRs) and Model-Specific Registers (MSRs). If needed, (part of) this extended thread state could be replicated per logical thread, and could be saved and restored by the thread swap routine.

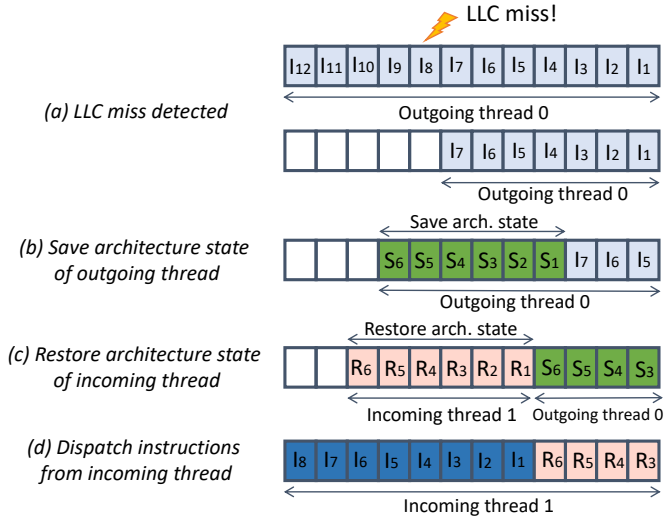

Fig. 4. Thread swapping. *Instruction 8 triggers a memory request. The core flushes instruction 8 and all younger instructions, and dispatches* save *instructions to store the architectural state of the outgoing thread. Next, the core dispatches* restore *instructions of the incoming thread. The core then continues dispatching and executing instructions from the incoming thread.*

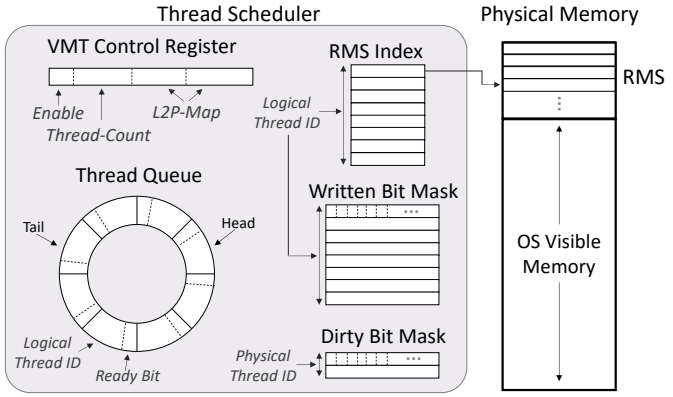

Fig. 5. VMT thread scheduler. *VMT requires a Control Register, Thread Queue, RMS Index, and Written/Dirty Bit Masks.*

Next, the core starts executing hardware-injected store instructions to save the architecture state of the outgoing thread in the L1 cache, see Figure 4b. The store instructions are renamed as ordinary instructions and the issue logic solves their dependencies with the instructions producing the register values they are saving. Meanwhile, the instructions that are older than the long-latency load that triggered the thread swap are allowed to execute and drain the pipeline as during normal operation.

Once the store instructions are dispatched into the ROB and store queues, the core starts retrieving the architectural state of the incoming thread, see Figure 4c. These hardware-injected load instructions are renamed and may possibly issue before the store instructions of the outgoing thread, reducing the overhead introduced by a thread swap.

Finally, after dispatching all restore instructions, the core starts dispatching instructions from the incoming thread, see Figure 4d. The execution of these instructions may overlap with the thread restore instructions and even with the save instructions of the outgoing thread. In other words, VMT does not need to have the entire architecture state of a thread restored before restarting execution. As the restore instructions get executed, ready instructions can be issued (out-of-order).

Note that when a thread swap is initiated along a mispredicted path, the thread swap is canceled (i.e., in-flight save and restore instructions are flushed) and the thread state is rolled back to the correct state after the mispredicted branch. We find that this scenario is rare and has a negligible impact on VMT performance.

2.3 Hardware Support

VMT requires a small microprogrammed routine to insert save and restore instructions in the pipeline, a mechanism for early miss notification from the LLC, a reserved memory space (RMS), and a thread scheduler that includes (i) a control register, (ii) a RMS index, (iii) written and dirty register masks, and (iv) the VMT thread queue, see Figure 5.

VMT Control Register. VMT's operation is controlled through a special-purpose 16-bit control register. The register is broken down into three fields. The first field consists of a single *Enable* bit and is used to indicate whether VMT execution is enabled for the current process. As we will discuss, applications can initiate VMT execution upon request if deemed beneficial. The second field, *Thread-Count*, denotes the number of logical threads VMT should virtualize. By default, VMT virtualizes 16 logical threads per core. However, a user or system operator, for example through profiling-based analysis, may suggest a different number of threads to virtualize; the Thread-Count field provides a mechanism to do so. A 5-bit field is enough to virtualize up to 32 threads. The last field, *Logical-to-Physical Thread Map (L2P-Map)*, keeps track of which logical thread is mapped to which physical thread. For VMT implemented on top of a single-threaded core, the L2P-Map indicates which logical thread is currently mapped to the physical thread; all other logical threads are swapped out. When implemented on top of a two-way SMT core, two logical threads can be mapped to the two physical threads available. The L2P-Map is required to support context switching, as we will discuss in Section 2.4. Assuming two physical threads and up to 32 logical threads, 10 bits is enough for the L2P-Map.

Reserved Memory Space. To save the architectural state of swapped-out threads in the processor's memory hierarchy, we need to save its architectural registers in a dedicated memory region. We reserve a small portion of the processor's physical address space that cannot be accessed by the OS and is otherwise unused, which we call the *Reserved Memory Space (RMS)*. Saving the architecture state in the RMS instead of the OS's memory structure for context switches allows VMT to perform thread swaps in hardware, without any intervention by the OS. The architecture state of a thread in x86_64 is at most 2.15 KB as mentioned before. For simplicity, we reserve 1 MB of the address space (for example, in the highest address range), which is large enough to hold up to 256 threads (conservatively) assuming 4 KB of architecture state per thread. Each thread is assigned a 4 KB chunk in the RMS. The architecture state of each thread starts aligned to a 4KB address (no cache block contains architecture state of different threads) and is private. Therefore, it should not be evicted by other cores.

For each logical thread, we keep track of the 8-bit chunk index in the so-called *RMS Index*.

Microprogrammed Routine and Register Masks. To virtualize the architectural state of the logical threads, the hardware needs to insert instructions to save and restore the architectural state of the threads involved in a thread swap. This functionality is provided by a small microprogrammed routine (off the critical path) that iterates over the architectural registers and generates load and store instructions to save and restore the architecture state to the RMS. These instructions are directly inserted in the pipeline after the decode stage.

To reduce thread swap overhead, VMT minimizes register saves and restores, as in [19]. When saving the architecture state of an outgoing thread, the microcode checks a *dirty* bit per architecture register which identifies whether the architectural register has been modified during the last execution epoch of the logical thread. Registers that are not dirty do not need to be saved. VMT keeps a *Dirty Bit Mask* per physical thread. The core sets the corresponding bit when a thread writes a register and the entire mask gets cleared when a thread is swapped out. A mask of *written* bits per logical thread is used to determine the architectural registers that a logical thread has ever written. A thread needs to restore a register if it has ever written the register or the outgoing thread wrote the register during its last epoch (i.e., it is dirty). The *Written Bit Mask* is updated when a thread is swapped out doing a logical OR operation between the thread Dirty Bit Mask and Written Bit Mask, and is cleared when VMT mode is initiated.

Because the save and restore operations operate on physical addresses, they bypass the TLB. Bypassing the TLB is done using a multiplexer that selects between the memory address provided by the instruction (for VMT save and restore instructions) versus the translated address provided by the TLB (for conventional load and store instructions).

Thread Queue. VMT thread scheduling requires a circular queue, the *Thread Queue (TQ)*, to keep track of the concurrently executing logical threads. Each TQ entry contains a logical thread identifier plus a *ready* bit that identifies if the memory request that triggered the thread swap has already completed. A thread swap is started by a logical thread that is being executed and triggers a memory request. Once the architecture state of this thread is saved, the logical thread is added at the TQ tail.

While the store instructions in charge of saving the architecture state of the outgoing thread are dispatched, an incoming thread is selected. The thread at the TQ head is selected for execution if its ready bit is set (i.e., its memory request has completed). Otherwise, it is moved to the TQ tail. We find that this situation is infrequent and has negligible impact on performance. If the number of logical threads is low, the thread will be re-selected soon; if, on the other hand, the number of logical threads is high, the probability that its memory request is still pending is low.

Note that the thread selection logic is out of the processor's critical path and does not need to select the next thread in a single cycle as the outgoing thread requires several cycles to insert all of its save instructions. Once this operation is completed, the incoming thread starts inserting instructions in the pipeline to restore its architecture state.

Hardware Cost. We assume 32 logical threads and 2 physical threads per core. The VMT control register requires 2 bytes of storage, as mentioned before. The TQ incurs 24 bytes of storage as it keeps track of up to 32 logical thread IDs and a *ready* bit per thread in a circular queue. The RMS Index requires 32 bytes of storage: one byte per logical thread. We need a 32-bit Written Bit Mask for each of the 32 logical threads, and we need a 32-bit Dirty Bit Mask for the 2 physical threads.[4] The microcode state machine for generating the save and restore instructions requires a counter to iterate over the architectural registers (6 bits). Put together, implementing VMT in a 2-way SMT core requires 195 bytes of storage.

2.4 Operating System Support

An application requests VMT support from the OS, e.g., when the application reaches a parallel section. To grant VMT support, the OS sets the Enable bit in the VMT control register of the core. If the application requests a particular number of logical threads to be virtualized, the Thread-Count field is set accordingly in the control register. Otherwise, the default number of logical threads per core is assumed (i.e., 16). A RMS Index is set for the logical threads. Finally, the hardware initializes the TQ with the logical thread IDs and the ready bits are set. The Written/Dirty Bit Masks are cleared. Once this is done, VMT is operational and the OS returns to the application, which now spawns as many threads as the underlying machine exposes logical threads.

During VMT operation, logical threads are swapped in and out without intervention of the OS. The L2P-Map keeps track of the current logical-to-physical thread mapping. From the OS' perspective, all the application threads mapped onto the core are running simultaneously as logical threads, even though only a limited number of threads are intermittently executing as physical threads. VMT works with any thread scheduling policy but VMT performance is maximized when multiple threads from the same application are co-scheduled to maximize the exploitable MLP across threads. Thus, we assume gang scheduling, which co-schedules threads from the same process on the same core. Any event that interrupts the execution of one thread (e.g., a page fault) will context-switch all threads on that core; the threads are re-scheduled onto the core again once the exception or interrupt returns.

The application disables VMT, e.g., once the parallel section terminates, by clearing the Enable bit in the VMT control register. Once VMT is disabled, the core only exposes the physical threads to the OS.

Context Switching. The VMT control register, the TQ, the RMS Index registers as well as the Written/Dirty Bit Masks are stored as part of the process control block (PCB) upon a context switch. To support context switching under VMT, the OS makes a distinction between the logical threads that are swapped-in versus the ones that are swapped-out. The OS does not need to save the architecture state

4. We need the masks for the 32 general-purpose registers only; the 3 special-purpose registers are always saved and restored.

of the swapped-out threads, because the architecture state of those threads has already been saved in their respective RMS chunks. On the other hand, for the swapped-in threads, the OS needs to save their architecture state using the conventional context switch routine. When the OS re-schedules the gang onto the core, the OS re-installs the VMT control register, the TQ, the RMS Index registers, and the Written/Dirty Bit Masks. This re-enables VMT (i.e., the enable bit is set), re-sets the number of logical threads (i.e., through the Thread-Count field), and re-stores the latest logical-to-physical thread mapping (i.e., through the L2P-Map).

Security Concerns. Security vulnerabilities are mitigated with two actions. First, VMT operates at the hardware level and only VMT save and restore instructions are allowed to access the RMS which cannot be accessed by the OS nor application software. Second, the OS manages the RMS indices and, in case multiple applications co-run on different cores in a multicore processor, the OS would need to assign a different RMS base address to each application that requests VMT execution to ensure that co-running applications cannot read or write each other's chunk in the RMS. When an application completes its VMT execution, the threads' state in the cache hierarchy is flushed by the hardware. Only then can the OS assign a RMS region to another application.

VMT does not increase vulnerability to recent microarchitectural side channel attacks such as Meltdown [25] and Spectre [26] because it does not increase the amount of speculative work performed. Moreover, hardware countermeasures are applicable to VMT-enabled processors. Other attacks exploit *lazy* saving and restoring of floating-point architectural registers during a context switch to obtain their values [27]. VMT does not increase the vulnerability to these attacks either since it performs lazy saving and restoring of registers only when swapping threads within a given application.

2.5 Setting the Number of VMT Threads

VMT performance scalability with logical thread count is limited by at least three factors: (i) by the amount of thread-level parallelism in the application (i.e., the application by itself should scale well with thread count); (ii) by the application's memory intensity, or in other words, by how quickly threads can get to the next LLC miss and thus swap threads; and (iii) by how fast VMT can swap threads, which depends on the number of dirty registers that need to be saved and the number of written registers that need to be restored. All three factors depend on the workload and its interaction with the underlying architecture. VMT performance benefits saturate with increasing logical thread count when there is always a swapped-out thread with its memory request completed (and thus ready to be swapped in again) when a currently running thread triggers a memory request. Increasing the number of logical threads beyond this point does not further improve performance. On the contrary, performance may even degrade if, for example, saving and restoring the architectural state of all threads affects cache performance, or if the amount TLP is limited because of synchronization overheads.

For the graph workloads and the processor architecture considered in this work, we find that 16 logical threads per core is optimal on average, as we will quantify in the evaluation section. We therefore employ 16 VMT threads per core by default. However, we provide hardware support for up to 32 VMT threads per core, as some workloads benefit from enabling more than 16 VMT threads per core. VMT provides support to set the number of VMT threads per core on a per-application basis. A profile-driven VMT approach could be used to determine the optimum number of VMT threads for each workload: VMT performance is evaluated as a function of VMT thread count using a training input, based on which the optimum VMT thread count is determined. For a previously unseen production input, the VMT thread count is then set to this optimum VMT thread count. Note that profiling incurs a one-time cost and is paid off across multiple runs of the same application.

TABLE 1
Simulated multicore processor configuration.

Cores and frequency	4 cores at 3.66 GHz
SMT threads	1 versus 2
Issue queue / ROB	97 / 224 entries
Load / store queue	72 / 66 entries
Processor width	dispatch: 4; issue: 6; commit: 4
Pipeline depth	8 (front-end)
Register file	168 64-bit int, 168 128-bit fp
Branch predictor	hybrid bimodal, gshare, loop
L1 I-cache	32 KB, 4-way, 2 cyc
L1 D-cache	32 KB, 8-way, 4 cyc, tag lookup: 1 cyc
Private L2 cache	256 KB, 8-way, 8 cyc, tag lookup: 3 cyc
Shared LLC	8 MB, 16-way, 30 cyc, tag lookup: 10 cyc
LLC prefetcher	stride prefetching, 16 streams per core
L1 TLBs	DTLB: 64-entry, 4-way ITLB: 128-entry, 4-way
L2 TLB	shared TLB: 512-entry, 4-way
MSHR	60 entries
Memory	DDR4, 51.2 GB/s, 45 ns

3 Experimental Setup

We evaluate VMT using the most accurate cycle-level core model in Sniper [28] — a parallel, fast and hardware-validated multicore simulator — which was extended to faithfully model a state-of-the-art multi-core SMT processor. We consider two baseline quad-core processor configurations with single-threaded and 2-way SMT cores (i.e., one and two physical threads per core), respectively. A total number of 128 and 256 threads are enabled on the chip, respectively, assuming VMT with 32 logical threads per physical thread, as opposed to 4 and 8 threads in the baseline configurations. We expect that VMT provides significant performance benefit for processors with several tens of cores since there is abundant TLP to be exploited in graph workloads [22] and memory bandwidth would only be saturated if there are lots of independent, overlapping memory accesses, which is not the case for graph workloads because of dependent misses. The simulated processor configuration is summarized in Table 1. Our baseline core configuration closely resembles a commodity processor like Intel's Skylake microarchitecture [29] with a three-level cache hierarchy and a stride-based LLC hardware prefetcher with 16 streams per core. Given that graph workload performance is dominated by irregular memory accesses with limited temporal and spatial locality, VMT performance is insensitive to cache size and replacement policy. We faithfully model VMT. The

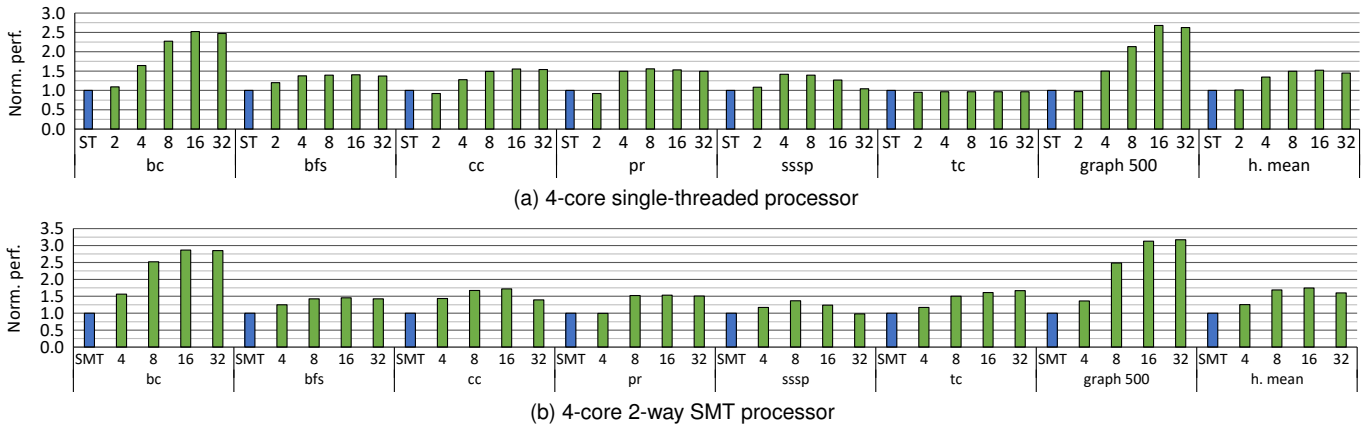

Fig. 6. Speedup through VMT with up to 32 logical threads per core for a single-threaded baseline core (a) and a 2-way SMT baseline core (b). *VMT yields substantial performance benefits over a conventional commodity processor.*

VMT thread scheduler is notified of an LLC miss 17 cycles after the load was issued by the core: this includes 14 cycles of cumulative tag lookups in the L1, L2 and LLC, plus 3 additional cycles to account for the time it takes for the core to be notified of the LLC miss through a dedicated channel.[5] We also account for the time the core requires to rebuild the RAT once it flushes the LLC-missing load and all younger instructions: the core restores the last valid checkpoint and traverses the ROB to update the RAT accordingly. We assume that a checkpoint is available at the youngest branch from which the core traverses the ROB updating the RAT at a pace of 4 instructions per cycle until reaching the youngest instruction in the ROB. This assumes a RAM-based RAT [30]; a CAM-based RAT [31] would incur even lower latency overhead since the RAT could be checkpointed on every load instruction. Our simulation experiments report that rebuilding the RAT takes 1.8 cycles on average (and at most 2.9 cycles, for `graph500`). The fact that the LLC-missing load is frequently the oldest instruction in the ROB when the miss is detected greatly contributes to the low RAT recovery latency.

It is worth noting that since all logical threads mapped to the same core share the same virtual address space, the entries in the data and instruction TLBs do not need to be flushed upon a thread swap. Similarly, the branch predictor tables are also shared — we observe minimal impact on branch prediction accuracy when sharing the branch predictor across threads for most of the benchmarks.[6]

We consider `graph500` v2.1.4 [23] and the six applications from the GAP benchmark suite [7]: Betweenness Centrality (`bc`), Breadth-First Search (`bfs`), Connected Components (`cc`), PageRank (`pr`), Single-Source Shortest Path (`sssp`) and Triangle Count (`tc`). We skip the initialization of the graphs as well as the preprocessing steps. We run each workload twice: we use the first execution to warm up the caches, and we simulate and report timing for the second execution. As input, we use graphs generated with the built-in graph generator with size 21 (except for `tc`). These graphs are formed by 2^{21} vertices following the Kronecker distribution, complying with the `graph500` specifications. The overall number of instructions simulated in detail ranges from 200 million for `bfs` to 2.7 billion for `pr`. `tc` is less memory-intensive than the other workloads and its instruction count grows much faster with graph size. To keep simulation time reasonable, we simulate `tc` with graphs of size 19, which results in 24.5 billion instructions. Section 4.4 analyzes VMT performance when running real-world graphs.

5. An alternative implement would be to notify the core via a message sent across the NoC. In any case, we find that VMT is rather insensitive to the LLC notification latency. VMT's performance benefit is only marginally affected by an increased LLC notification latency: 1.74× improvement (for 3-cycle latency) versus 1.70× improvement (for 10-cycle latency).

6. Branch misprediction rate decreases from 6.7% to 3.5% for `cc` and increases from 5.5% to 5.8% for `sssp`; the other benchmarks are unaffected.

4 Evaluation

4.1 Overall Performance

Figure 6 reports performance normalized to the baseline commodity quad-core processor with single-threaded (Figure 6a) and 2-way SMT (Figure 6b) cores, respectively, for different configurations of the proposed VMT architecture: (a) single-threaded cores enhanced with VMT support and 2 to 32 logical threads per core, and (b) SMT cores enhanced with VMT and 4 to 32 logical threads per core. Thread swap overhead is accounted for in the results. VMT provides significant performance benefits. Focusing on the 2-way SMT configuration first, VMT improves performance by 3.17× for `graph500`, 2.86× for `bc`, 1.72× for `cc`, and 1.53× for `pr`. The highest performance is typically achieved for 16 logical threads. With this large number of threads, the number of cycles the core has all threads stalled waiting for the memory requests to be completed is limited. Consequently, increasing the number of logical threads beyond 16 provides only marginal performance benefits. For some workloads, performance even degrades, for the reasons alluded to before. VMT also provides a significant performance boost for the single-threaded cores, improving performance by 2.68× for `graph500`, 2.52× for `bc`, 1.56× for `pr`, and 1.55× for `cc`.

We observe somewhat different behavior for `tc` compared to the other graph workloads, for two reasons: (1) `tc` is less memory-intensive, and (2) it is highly imbalanced as the first thread receives a much higher load [8]. VMT does not improve performance for the single-threaded core because VMT essentially serializes the execution of various

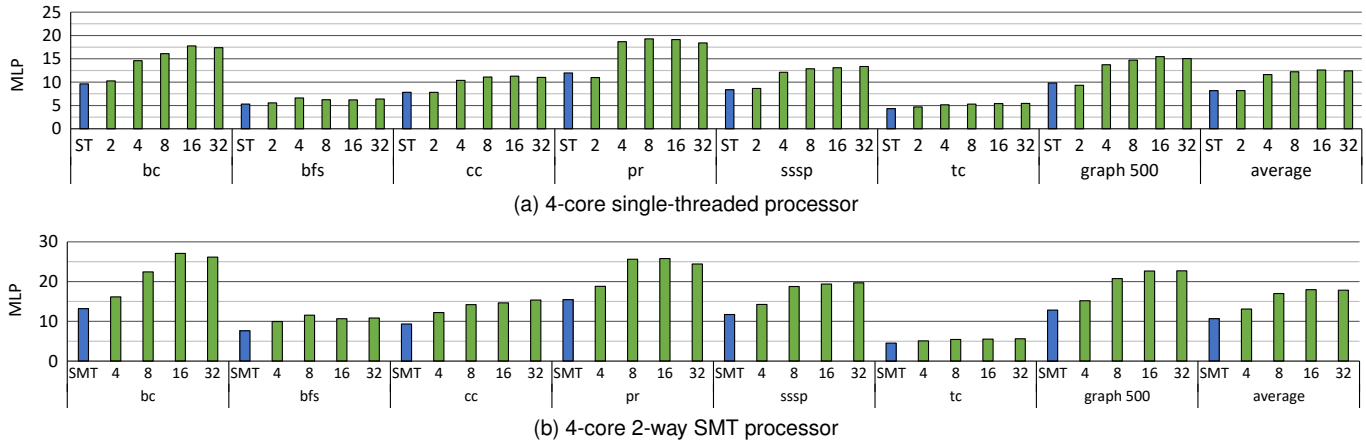

Fig. 7. Quantifying MLP benefits under VMT. *Increased MLP strongly correlates with VMT's performance benefits.*

logical threads without improving MLP (as we show next). The performance improvement observed for the 2-way SMT core is a result of overlapping the execution of multiple lightly loaded threads with the heavy-loaded thread.

Memory-Level Parallelism. We find that memory-level parallelism (MLP) is the key contributor to improved performance, see Figure 7 which reports MLP for the different processor configurations: improvements in MLP strongly correlate with the VMT performance gains. MLP improves with an increasing number of logical threads, and for most benchmarks, MLP tends to saturate around 8 or 16 logical threads. The degree of MLP exposed through VMT is a result of the workload's memory intensity, thread swap frequency, and thread swap overhead (i.e., number of save and restore instructions). `graph500` and `bc` reach the highest MLP and performance gains because of their relatively high thread swap frequency and few save and restore instructions per thread swap, as we will quantify in the next section. The other workloads swap threads at a lower pace and the MLP improvement over 8 logical threads is limited. Increasing the number of logical threads further does not improve MLP because the core is unable to swap logical threads quickly enough to expose more parallel memory requests. `tc` is the least memory-intensive benchmark of the workloads considered in this study, see Figure 2a, hence the amount of extracted MLP is limited.

VMT Default Configuration. As noted before, the optimum logical thread count varies across benchmarks, however, we find that, on average, optimum VMT performance is achieved for 16 logical threads for both the single-threaded and SMT cores. This configuration leads to an average speedup of $1.52\times$ (and up to $2.68\times$) for VMT on top of the baseline 4-core processor with single-threaded cores, and $1.74\times$ (and up to $3.17\times$) for VMT on top of the baseline 4-core processor with 2-way SMT cores. In the remainder of this paper, we report VMT results assuming 16 logical threads and 2 physical threads per core, unless stated otherwise.

4.2 Saving and Restoring Architecture State

We now quantify VMT's thread swap operation, i.e., thread swap frequency, the number of save and restore instructions per thread swap, and the hit rate of the restore instructions in the cache hierarchy.

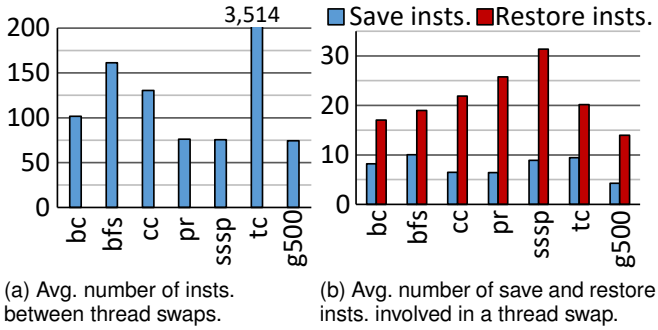

Fig. 8. Average number of instructions between VMT thread swaps and average number of save and restore instructions involved in a VMT thread swap. *A smaller number of instructions between thread swaps allows for triggering them faster. A smaller number of save and restore instructions reduces the swapping overhead.*

Thread swap frequency. A high thread swap frequency is needed to engage VMT: the smaller the number of instructions between thread swaps, the higher the opportunity to expose MLP. Figure 8a quantifies the number of instructions between two thread swaps which is a function of the workload's memory intensity. The number of instructions between thread swaps varies from 74 (`graph500`) to 150 instructions (`bfs`) on average; `tc` is the outlier with 3,514 instructions between thread swaps because of its relatively low memory intensity.

Number of save and restore instructions. It is important that the overhead per thread swap is as small as possible. A first-order metric for thread swap overhead is the number of save and restore instructions per thread swap, see Figure 8b. The number of save instructions is smaller than the number of restore instructions because the save instructions only need to store the dirty architecture registers that were written in the last execution epoch; the restore instructions need to load all the architecture registers that the thread has ever written.

Hit rate of restore instructions. It is critical that the restore instructions find the architecture state as close to the core as possible, and preferably in the L1 data cache. Figure 9 reports the levels in the memory hierarchy at which the

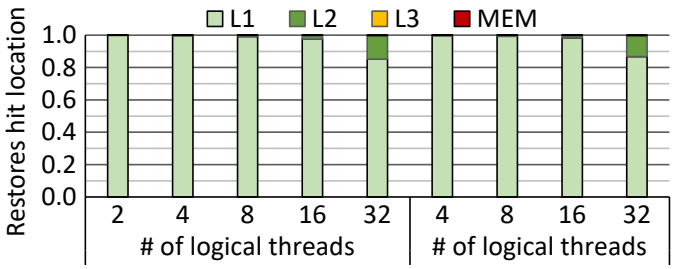

Fig. 9. Percentage of restore instructions that hit in L1, L2, L3 and main memory for `sssp`. *VMT restore instructions most frequently hit in L1.*

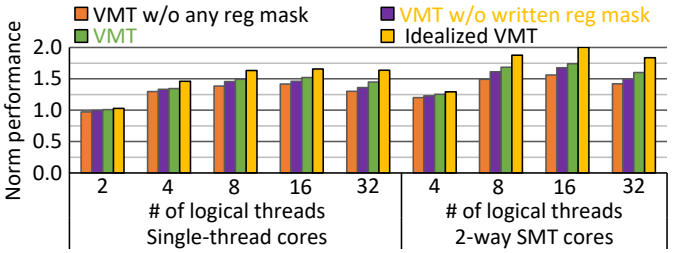

Fig. 10. Comparing VMT variants against idealized VMT. *Register masks significantly reduce VMT overhead.*

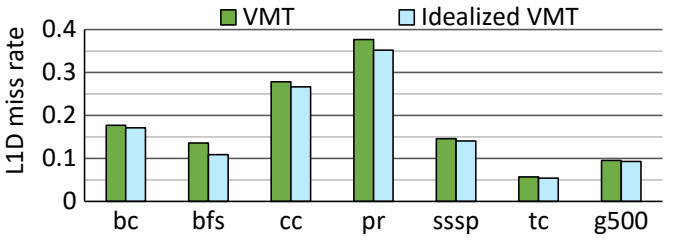

Fig. 11. L1 D-cache miss rate for VMT versus idealized VMT. *VMT marginally impacts the L1 D-cache miss rate by saving architecture state in the cache hierarchy.*

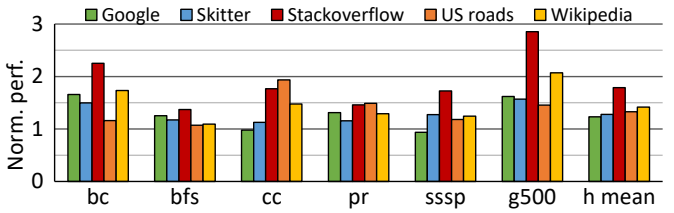

Fig. 12. Normalized performance for VMT compared to our baseline. *VMT offers significant performance improvements across workloads and input graphs.*

restore instructions hit when running the `sssp` benchmark. (`sssp` is the benchmark with the highest number of restore instructions missing in L1.) These results show that VMT most often restores the architectural state of the threads from the L1 cache. The fact that graph workloads present poor memory locality and the high pace at which threads are swapped in and out allows for a high L1 hit rate when restoring architecture state. Even for `sssp`, the hit rate in L1 only reduces notably for 32 logical threads, in which case the L1 hit rate of the restore instructions reduces to 85%; this increases thread swapping latency which in turn dampens VMT's performance benefits. Note that the architectural state never gets evicted to main memory and very infrequently to the LLC.

4.3 VMT Performance Analysis

It is instructive to analyze VMT's performance contributors and its maximum potential. Figure 10 evaluates four VMT variants: (i) VMT without any register masks; (ii) VMT with the dirty register masks (but no written register mask), (iii) VM with both the dirty and written register masks (i.e., the proposed VMT solution), and (iv) an idealized version of VMT with an unrealistically large register file to hold the architectural state for all logical threads. The latter does not trigger any overhead for saving architecture state.

VMT without register masks outperforms the baseline 2-way SMT processor by 1.56×. Enabling the dirty register mask increases performance to 1.68×. Enabling both the dirty and the written register masks increases performance by 1.74×. The idealized VMT yields a 2.00× speedup. We conclude that the register masks are an important component to reduce VMT overhead. We also find that the gap with an idealized version of VMT is considerable.

We further find that the impact of storing a thread's architecture state in the cache hierarchy on cache performance is limited, see Figure 11 which quantifies the impact of VMT on L1 data miss rate: the L1 D-cache miss rate is not significantly affected.[7] We thus conclude that VMT's performance overhead is mainly caused by the time required to save and restore a thread's architecture state.

4.4 Input-Graph Sensitivity

We now evaluate VMT performance across input graphs. We evaluate different real-world graphs taking the profile-driven VMT approach. Based on a profiling phase, i.e., performance evaluation carried out using the Kronecker-based graph, we set the optimum number of threads for each workload as follows: 32 threads for `graph500`, 16 threads for `bc`, `bfs`, `cc`, and `pr`, and 8 threads for `sssp`.[8] Figure 12 reports VMT performance normalized to the baseline processor across the evaluated input graphs. This includes five real-world graphs [32]: `Google`, `Skitter`, `Stackoverflow`, `US-roads` and `Wikipedia`.

The key conclusion is that VMT provides significant performance benefits across the broader set of input graphs, even though the achieved benefits vary across input graphs (and workloads). The highest performance improvement (1.79×) is reported for the `Stackoverflow` graph. The `Google` and `Skitter` graphs result in smaller working sets, which in turn results in lower LLC MPKIs and consequently somewhat lower VMT speedups. On average, across all the real-world graphs, the performance benefit of VMT compared to the baseline SMT architecture amounts to 1.41×.

4.5 Other Parallel Workloads

As extensively argued, VMT fits the characteristics of graph workloads particularly well. However, VMT can also significantly improve performance for other memory-intensive

7. We note that the graph workloads experience an L1 I-cache miss rate of less than 1%. In addition, we find that VMT affects TLB performance only marginally. Not shown here because of space constraints.

8. `tc` is not evaluated here since it requires undirected graphs whereas the real-world graphs evaluated are directed.

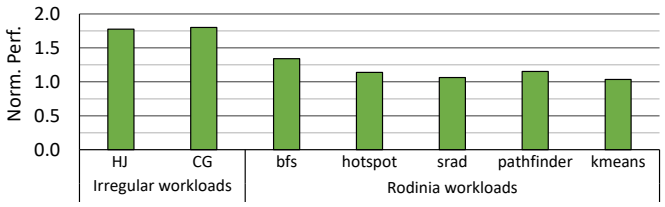

Fig. 13. Normalized performance for VMT for the Rodinia benchmarks. *VMT improves performance for parallel applications other than graph workloads. The performance improvements depend on the workload's TLP, memory intensity, and exploitable MLP within and across threads.*

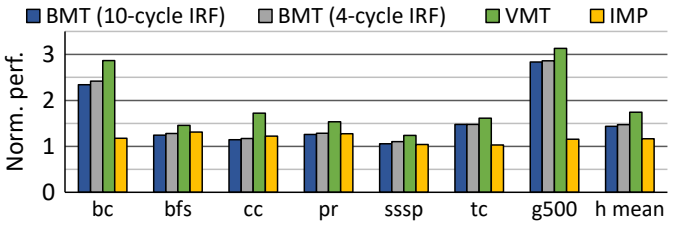

Fig. 14. Normalized performance for VMT, BMT with 10- and 4-cycle IRF access latency, and IMP compared to our baseline. *VMT outperforms the original BMT proposal by 22% on average while incurring much less hardware overhead (21 KB per core for BMT versus 195 bytes for VMT). VMT outperforms IMP by 49%: IMP's indirect pattern detection approach is less effective for aggressive out-of-order cores.*

parallel workloads. The achieved benefits depend on a workload's memory-intensity, its TLP, and its MLP. We evaluate VMT for two workloads that perform indirect memory accesses: Conjugate Gradient (CG) from the NAS Parallel Suite, which performs irregular memory accesses on a large, sparse, and unstructured matrix; and Hash Join (HJ) [33], a kernel that mimics database systems and performs irregular memory accesses through different hash tables. In addition, we also consider a set of Rodinia v3.1 benchmarks with an LLC MPKI above 1. Figure 13 reports normalized performance for VMT compared to the baseline processor; benchmarks are sorted from highest to lowest LLC MPKI. We report high speedups through VMT for the two irregular workloads: $1.80\times$ for CG and $1.78\times$ for HJ. Regarding the Rodinia benchmarks, VMT outperforms SMT by up to $1.34\times$ (bfs); speedup is more moderate for the other, less memory-intensive, benchmarks.

4.6 Comparison to BMT

In addition to VMT being substantially more hardware-efficient than BMT, as extensively argued before, there are three other differences in the underlying mechanisms that impact performance. (1) BMT waits for a fixed number of cycles before triggering a thread swap: the LLC hit latency plus some additional cycles to account for cache contention. (2) BMT waits until the LLC-missing load is the oldest instruction in the reorder buffer (ROB) to trigger the thread swap while VMT triggers the thread swap as soon as the miss notification is received. (3) BMT incurs a 10-cycle access latency to access the inactive register file (IRF).[9]

In our implementation of BMT we trigger a thread swap if a load does not complete its execution in 60 cycles (53 average LLC hit latency plus 7 additional cycles to account for cache contention) *and* the load is the oldest instruction in the ROB. In contrast, VMT triggers a thread swap as soon as an LLC tag miss is detected (i.e., 14 cycles L1/L2/LLC tag lookup plus 3 cycles to notify the core). This is motivated by the observation that there is limited per-thread MLP for graph workloads. In addition, we compare against BMT with a 10-cycle IRF access latency and BMT with a more aggressive IRF access latency of 4 cycles.

Figure 14 compares VMT against BMT with 10- and 4-cycle IRF access latency. VMT achieves higher performance than BMT. On average, VMT outperforms BMT, assuming a latency of 10 and 4 cycles to the IRF, by 22% and 19%, respectively. These results show that for graph workloads it is important to switch threads early, i.e., there is more MLP to be exploited across threads than within a thread.

4.7 Comparison to IMP

The Indirect Memory Prefetcher (IMP) [2] is a state-of-the-art hardware prefetcher for graph workloads and indirect memory accesses in the general form of A[B[i]], where arrays A and B refer to the data and index arrays, respectively. The index array (B[i]) is typically stored consecutively in memory and accessed sequentially. Accesses to the data array (A[B[i]), however, depend on the value of B[i], and tend to touch non-consecutive memory locations. The IMP approach is to first detect a streaming pattern to the index array and then identify an indirect pattern. IMP therefore relies on a mechanism that needs to first observe an access to the index array (B[i]), then an access to the data array (A[B[i]]), before the next element in the index array (B[i+1]) is accessed. While this condition is always met for in-order cores, which is the baseline configuration assumed in the IMP work, we observe that speculation in aggressive out-of-order cores frequently disturbs this access pattern (even when index array accesses hit in the L1), which complicates identifying indirect memory access patterns (and gaining confidence on the detected ones).

Figure 14 compares the performance benefits achieved by VMT against IMP. IMP improves performance compared to our baseline (which includes a stride prefetcher) by 16% but falls far from the performance benefit provided by VMT ($1.74\times$). The highest benefit is achieved for `bfs` and `pr` for which IMP improves performance by 31% and 28%, respectively.

5 Related Work

Multi-Threading Paradigms. Most of the body of research in hardware multi-threading relates to the three main paradigms: coarse-grain, fine-grain and simultaneous multi-threading. The fundamental innovation by VMT compared to prior work is that it virtualizes the logical threads' architecture state.

Coarse-Grain Multi-Threading (CGMT) processors [13], [14] switch logical threads upon miss events, e.g., a long-latency memory access, e.g., Intel Montecito and Sun Rock.

9. We assume the same access latency as in the BMT proposal even though we consider a larger number of logical threads (12 in the original BMT proposal versus up to 32 in our evaluation) and a bigger architectural state per thread (496 bytes in the original BMT proposal versus 664 bytes in our evaluation), which results in a $3.6\times$ bigger IRF. The 10-cycle access latency is motivated in the BMT paper by its distance from the core being similar to a 2MB L2 cache.

Fine-Grain Multi-Threading (FGMT) processors switch between logical threads on a per-cycle basis, e.g., Cray ThreadStorm [15]. Simultaneous Multi-Threading (SMT) [16] is the only multi-threading paradigm that allows a core to issue instructions from different threads in the same cycle. SMT is implemented in many commodity and server processors manufactured by Intel [29], AMD, and IBM [18].

Finally, Balanced Multi-Threading (BMT) [19] virtualizes physical threads which exacerbates the problem of maintaining the architecture state for all logical threads. In follow-on work, Brown et al. [34] propose to share BMT's inactive register file across cores, allowing threads to swiftly migrate across cores.

Memory-Backed Register File. Prior work has devised solutions to overcome the constraints imposed by the limited register file, particularly for multi-threading architectures. Soundararajan et al. [35], Nuth et al. [36], Kogge et al. [37], Oehmke et al. [38] and Li et al. [39] present hardware mechanisms to virtualize the logical registers by treating the physical register file as a cache of the much larger memory-mapped logical register space. These proposals require a deep redesign of the rename stage and/or register file, which opposes to our goal of requiring minimal hardware modifications on top of a commodity processor. Moreover, this prior work does not describe how virtualizing the architecture state enables a novel multi-threading paradigm that fits graph workloads well.

Tackling the same problem in a different scenario, Huguet et al. [40] propose a combination of architectural and compilation support to reduce the register saves and restores across function calls. Similar support could be added to VMT to reduce register saves and restores involved in a thread swap.

Improving Graph Workload Performance. A selection of prior work proposed hardware and software techniques to improve graph workload performance. Yu et al. [2] devise the indirect memory prefetcher (IMP), which is able to prefetch specific irregular patterns but relies on an indirect memory pattern detection mechanism that turns out to be less effective on aggressive out-of-order cores compared to in-order cores. We have shown that VMT outperforms IMP by a significant margin. Kiriansky el al. [3] extend the compiler to transform annotated loops with indirect memory references into batches of sequential DRAM accesses. Ainsworth et al. [4] propose an event-triggered programmable prefetcher that generates prefetches from annotated source code. Mukkara et al. [41] propose HATS, a hardware-accelerated traversal scheduler to improve graphs locality without expensive preprocessing. HATS requires changes in the graph processing framework and its performance benefits are related with the community structure of the graphs. Finally, Faldu et al. [42] propose GRASP, domain-specific LLC management for graph workloads to protect hot vertices against cache thrashing. GRASP requires graphs to be reordered to induce spatial locality; further, the set of hot vertices it can protect is limited by the LLC size. VMT can be applied to unmodified graphs.

Accelerators have also been proposed for graph workloads. Ahn et al. [5] propose a programmable processing-in-memory accelerator to provide memory-capacity-proportional performance in large-scale graph processing. Ozdal et al. [6] propose a customizable architecture optimized for different access patterns and graph workloads. Ahn et al. [5] and Zhuo et al. [43] propose Processing-In-Memory (PIM) architectures to maximize and optimize memory bandwidth usage.

Software Techniques to Improve MLP. Software techniques require modifications to source code or binaries, unlike VMT. Horowitz et al. [20] propose a new class of memory operations, called 'informing memory operations', to let software know if a memory reference suffers a cache miss. They briefly discuss how this mechanism allows for implementing a software-based approach in which a miss handler switches threads upon cache misses. More recent software approaches focus on improving MLP in pointer-chasing codes. In particular, Chen et al. [44] hide the cache miss latency associated with join operations in memory-sized hash tables by overlapping misses with computation. Kocberber et al. [45] propose asynchronous memory access chaining (AMAC) for exploiting inter-lookup parallelism to hide the memory access latency in in-memory databases. Psaropoulos et al. [21] propose coroutines to interleave among different instruction streams upon cache misses. The compiler encodes the different stages of a lookup within the coroutine and separates them with suspension/resumption points on potentially LLC-missing loads marked by the user. Unlike VMT, coroutines require rewriting the code and cannot be used in existing binaries. Nevertheless, comparing VMT against this software approach could be an interesting avenue for future work.

6 Conclusion

Graph workloads highly benefit from thread-level parallelism as different threads can issue independent memory requests, exposing MLP and improving performance. Multi-threading architectures are therefore an excellent fit for emerging graph workloads. Unfortunately, all existing multi-threading designs require dedicated hardware structures for storing the architecture state of all logical threads. This leads to an unjustifiably high hardware cost for commodity processors.

This paper proposes *Virtualized Multi-Threading*, a novel hardware multi-threading paradigm for commodity SMT processors to accelerate graph workloads at minimal hardware cost. VMT virtualizes an SMT's physical thread context among a large number of logical threads that are swapped in and out upon long-latency memory requests. VMT's key innovation is to virtualize the logical threads' architecture state by saving the architecture state in the processor's cache hierarchy, to minimize hardware cost. We further find that graph workloads are insensitive to L1 D-cache performance, in addition to having TLP, high LLC miss rate and low MLP, hence they are a particularly good fit for VMT. Our experimental results report that VMT achieves an average speedup of $1.74\times$ (and up to $3.17\times$) for a set of representative graph workloads, while incurring a minimal hardware cost of 195 bytes per core and limited additional logic.

The overall conclusion is that VMT is a promising paradigm to dramatically speed up graph workload performance. Its extremely low hardware cost paves the way

for adoption in current commodity processors. At the meta level, this work demonstrates a promising direction to innovate general-purpose processors through small hardware modifications with substantial performance improvements for important and emerging classes of workloads.

REFERENCES

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at Facebook-scale," *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, vol. 8, no. 12, pp. 1804–1815, Sep. 2015.

[2] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2015, pp. 178–190.

[3] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," in *Proceedings of the 25th International Conference on Parallel Architectures and Compilation (PACT)*, Sep. 2016, pp. 299–312.

[4] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2018, pp. 578–592.

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, Jun. 2015, pp. 105–117.

[6] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, Jun. 2016, pp. 166–177.

[7] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *CoRR*, vol. abs/1508.03619, 2015, [Online]. Available: http://arxiv.org/abs/1508.03619.

[8] S. Eyerman, W. Heirman, K. D. Bois, J. B. Fryman, and I. Hur, "Many-core graph workload analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, Nov. 2018, pp. 22:1–22:11.

[9] R. F. Mihalcea and D. R. Radev, *Graph-based natural language processing and information retrieval*. Cambridge University Press, 2011.

[10] M. J. Wainwright and M. I. Jordan, "Graphical models, exponential families, and variational inference," *Foundations and Trends in Machine Learning*, vol. 1, no. 1–2, pp. 1–305, Jan. 2008.

[11] T. Aittokallio and B. Schwikowski, "Graph-based methods for analysing networks in cell biology," *Briefings in Bioinformatics*, vol. 7, no. 3, pp. 243–255, Sep. 2006.

[12] T. Erős, D. Schmera, and R. S. Schick, "Network thinking in riverscape conservation – a graph-based approach," *Biological Conservation*, vol. 144, no. 1, pp. 184 – 192, 2011.

[13] W.-D. Weber and A. Gupta, "Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results," in *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 1989, pp. 273–280.

[14] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife machine: architecture and performance," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, Jun. 1995, pp. 2–13.

[15] A. Kopser and D. Vollrath, "Overview of the next generation Cray XMT," in *Cray User Group Proceedings*, 2011, pp. 1–10.

[16] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, Jun. 1995, pp. 392–403.

[17] M. Nemirovsky and D. M. Tullsen, *Multithreading architecture*, ser. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2013.

[18] B. Sinharoy, J. A. V. Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler, "IBM POWER8 processor core microarchitecture," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 2:1–2:21, Jan 2015.

[19] E. Tune, R. Kumar, D. Tullsen, and B. Calder, "Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy," in *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO)*, Dec. 2004, pp. 183–194.

[20] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith, "Informing memory operations: Providing memory performance feedback in modern processors," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, May 1996, pp. 260–270.

[21] G. Psaropoulos, T. Legler, N. May, and A. Ailamaki, "Interleaving with coroutines: A practical approach for robust index joins," *Proceedings of the VLDB Endowment*, vol. 11, no. 2, pp. 230–242, Oct. 2017.

[22] F. Checconi and F. Petrini, "Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines," in *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, may 2014, pp. 425–434.

[23] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[24] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2003, pp. 423–434.

[25] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX)*, 2018.

[26] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 1–19.

[27] J. Stecklina and T. Prescher, "Lazyfp: Leaking fpu register state using microarchitectural side-channels," *arXiv preprint arXiv:1806.07480*, 2018.

[28] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, p. 28, Aug. 2014.

[29] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation Intel Core: New microarchitecture code-named Skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

[30] E. Safi, P. Akl, A. Moshovos, A. Veneris, and A. Arapoyianni, "On the latency, energy and area of checkpointed, superscalar register alias tables," in *Proceedings of the 12th International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2007, pp. 379–382.

[31] E. Safi, A. Moshovos, and A. Veneris, "A physical level study and optimization of cam-based checkpointed register alias table," in *Proceeding of the 13th International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2008, pp. 233–236.

[32] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[33] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *Proceedings of the IEEE 29th International Conference on Data Engineering (ICDE)*, Apr. 2013, pp. 362–373.

[34] J. A. Brown and D. M. Tullsen, "The shared-thread multiprocessor," in *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS)*, Oct. 2008, pp. 73–82.

[35] V. Soundararajan and A. Agarwal, "Dribbling registers: A mechanism for reducing context switch latency in large-scale multiprocessors," pp. 1–21, 1994, mIT Technical Report.

[36] P. R. Nuth and W. J. Dally, "The named-state register file: Implementation and performance," in *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, Jan. 1995, pp. 4–13.

[37] P. M. Kogge, J. B. Brockman, D. T. Harper III, B. Smith, and I. C. D. Callahan, "Computer systems with lightweight multi-threaded architectures," 2009, uS Patent 7,584,332.

[38] D. W. Oehmke, N. L. Binkert, T. Mudge, and S. K. Reinhardt, "How to fake 1000 registers," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Nov. 2005, pp. 7–18.

[39] S. Li, S. Kuntz, J. Brockman, and P. Kogge, "Lightweight chip multi-threading (lcmt): Maximizing fine-grained parallelism on-chip," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 22, no. 7, pp. 1178–1191, 2011.
[40] M. Huguet and T. Lang, "Architectural support for reduced register saving/restoring in single-window register files," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 66–97, 1991.
[41] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 1–14.
[42] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, feb 2020, pp. 234–248.
[43] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, oct 2019, pp. 712–725.
[44] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry, "Improving hash join performance through prefetching," *ACM Transactions on Database Systems*, vol. 32, no. 3, 2007.
[45] O. Kocberber, B. Falsafi, and B. Grot, "Asynchronous memory access chaining," *Proceedings of the VLDB Endowment*, vol. 9, no. 4, pp. 252–263, Dec. 2015.

Josué Feliu received his MSc and PhD degrees in computer engineering from the UPV, Spain, in 2012 and 2017, respectively. He is currently working as a postdoctoral researcher at the Department of Computer Engineering of the same university. His research interests include scheduling strategies and performance modeling for multicore and multi-threaded processors. He was awarded the "IEEE TCSC Outstanding Ph.D Dissertation Award" in 2017.

Ajeya Naithani received the PhD degree in computer science and engineering from Ghent University in 2019, and the MS degree in computer science from the University of Arizona in 2011. Currently, he is working as a postdoctoral researcher at Ghent University. His research interests include the area of computer architecture with an emphasis on designing novel techniques to improve performance, energy-efficiency, and reliability of modern processors.

Julio Sahuquillo received the BS, MS, and PhD degrees from the UPV, Spain, all in computer engineering. He is a Full Professor with the Department of Computer Engineering at the UPV. He has authored over 150 refereed conference and journal papers. His current research interests include multi- and manycore processors, memory hierarchy design, cache coherence, GPU architecture, and architecture-aware scheduling. Dr. Sahuquillo is a member of the IEEE Computer Society.

Salvador Petit received the PhD degree in computer engineering for the UPV, Spain. Since 2009, he has been an Associate Professor with the Computer Engineering Department, UPV. He has authored over 100 refereed conference and journal papers. His current research interests include multithreaded and multicore processors, memory hierarchy design, GPU architecture, and resource management. Dr. Petit is a member of the IEEE Computer Society. In 2013, he received the Intel Early Career Faculty Honor Program Award.

Moinuddin Qureshi received his Ph.D. (2007) and M.S. (2003) from the University of Texas at Austin. He currently is a Professor of Computer Science at Georgia Tech. His research interests include computer architecture, memory systems, hardware security, and quantum computing. He is a member of the Hall of Fame for ISCA, MICRO, and HPCA. His research has been recognized with the best paper award at MICRO 2018 and HiPC 2014, and two awards at IEEE MICRO Top Picks. His ISCA 2009 paper on Phase Change Memory was awarded the 2019 Persistent Impact Prize. He was the Program Chair of MICRO 2015 and Selection Committee Co-Chair of Top Picks 2017.

Lieven Eeckhout received the PhD degree in computer science and engineering from Ghent University, in 2002. He currently is a Full Professor at Ghent University, Belgium. His research interests are in the area of computer architecture, with a specific interest in performance analysis, evaluation and modeling, as well as dynamic resource management. He is the recipient of the 2017 ACM SIGARCH Maurice Wilkes Award, the 2017 ACM SIGPLAN OOPSLA Most Influential Paper Award, two IEEE Micro Top Picks selections, and was elevated to IEEE Fellow in 2018. He served as the Editor-in-Chief of IEEE Micro (2015–2018), and as Program Chair of ISPASS 2009, CGO 2013, HPCA 2015 and ISCA 2020.