TriKon: A Hypervisor Aware Manycore Processor

Rohan Bhalla†, Prathmesh Kallurkar†, Nitin Gupta‡, Smruti R. Sarangi†
† Computer Science Department, Indian Institute of Technology, Hauz Khas, New Delhi, India
‡ Amazon Development Centre, Bangalore, India
E-mail: † {mcs122799, prathmesh.kallurkar, srsarangi}@cse.iitd.ac.in, ‡nitigupt@amazon.com

***Abstract*—Virtualization is increasingly being deployed to run applications in a cloud computing environment. Sadly, there are overheads associated with hypervisors that can prohibitively reduce application performance. A major source of the overheads is the destructive interference between the application, OS, and hypervisor in the memory system. We characterize such overheads in this paper, and propose the design of a novel *Triangle* cache that can effectively mitigate destructive interference across these three classes of workloads. We subsequently, proceed to design the *TriKon* manycore processor that consists of a set of heterogeneous cores with caches of different sizes, and *Triangle* caches. To maximize the throughput of the system as a whole, we propose a dynamic scheduling algorithm for scheduling a class of system and CPU intensive applications on the set of heterogeneous cores.**

The area of the *TriKon* processor is within 2% of a baseline processor, and with such a system, we could achieve a performance gain of 12% for a suite of benchmarks. Within this suite, the system intensive benchmarks show a performance gain of 20% while the performance of the compute intensive ones remains unaffected. Also, by allocating extra area for cores with sophisticated cache designs, we further improved the performance of the system intensive benchmarks to 30%.

***Keywords*-cloud; hypervisor; architecture support for virtualization;**

I. INTRODUCTION

As a direct consequence of Moore's law, the number of cores per chip is doubling roughly every two years. Consequently, in the near future we expect to have 32-64 core multicore processors. Researchers have already started designing processors with 80+ cores [1, 2]. Such processors are ideally suited for a cloud computing environment that caters to a large number of users with different needs. Particularly, in a cloud computing environment such as Amazon EC2, users are given a platform of their choice, and are allotted computing and storage resources that can grow or shrink depending upon the usage. A critical enabling technology in a cloud is the virtual machine monitor (also referred to as a hypervisor). The hypervisor is a software that provides the abstraction of real hardware to an operating system and applications running on it. It is possible to run multiple virtual machines on a server, and thus support multiple operating systems. Each group of users can be assigned to one operating system. They will be completely oblivious of other operating systems running on the same physical machine. In this manner it is possible to isolate sets of users from each other by creating a secure sandbox for each user. Moreover, this approach is also very flexible. Users are no more tied to a particular machine. If a machine is down, then the virtual machine can seamlessly be moved to another machine, and the services can be restarted. This process can be made completely transparent to the user. Lastly, virtualization allows a great degree of customizability. Users can install their own libraries, software, and run their own services without affecting the setup of other users.

It is important to note that there are two kinds of hypervisors – Type 1 or bare metal (run on the physical machine), and Type 2 (run on a guest operating system). Type 1 hypervisors such as Xen [3] and VMWare ESX [4] are more efficient, and are thus preferred in large cloud computing environments. Sadly, virtualization is not a panacea for all problems. Various studies [5, 6] have analyzed performance overheads of applications running on a VM. They have concluded that slowdowns can range from 25-34% as compared to applications running on a non-virtualized system. Our results also show a similar trend. This is because in the Xen VM, each system call becomes more complicated with the the guest operating system making a *hypercall* to the hypervisor for processing the system call. This is needed because guest operating systems are typically not allowed to access I/O devices directly. They need to bundle users' requests, and send them to the hypervisor such that it can take the desired action. Often it is necessary to make several hypercalls for processing a single system call. Similarly, interrupt processing routines also get more complicated. An interrupt needs to be routed from the hypervisor to the guest operating system, and needs processing at both the levels (guest OS and hypervisor).

The first category of solutions for mitigating overheads focus on smarter scheduling [7, 8, 9, 10] of applications within a VM, and VMs within a multicore processor. These approaches assume a large multicore processors with different types of cores. Some cores are suitable for running compute intensive benchmarks and some slower low-power cores are more suitable for running I/O intensive benchmarks. A set of applications are scheduled on these cores to either maximize throughput with power constraints, or minimize power with throughput constraints. The second

category of solutions [11, 12, 13] focus on modifying the hypervisor (with or without hardware support) to make it more efficient. Such schemes try to reduce the number of context switches between the applications and the guest OS, or between the guest OS and the hypervisor. Alternatively, they try to directly give access to the guest OS to some I/O devices (albeit with security guarantees).

We try a different approach in this paper. We start out by recognizing that the loss in performance due to virtualization has two reasons. The first is that the hypervisor steals actual CPU time, and the second is that hypervisor routines displace application and guest OS lines from the caches. This causes destructive interference in the caches and reduces performance. In this paper, we propose the *TriKon*[1] processor that has a novel memory system, which is designed to solve the problem of destructive interference. In specific, we make four contributions. (1) We characterize a set of system and CPU intensive benchmarks and study their memory behavior in terms of the interaction between the application, guest OS, and the hypervisor. (2) Based on the results of our characterization studies, we propose the design of a new kind of cache called a *triangle cache*. The triangle cache is actually a set of three small caches that can trade data between each other. The aim is to appropriately balance the memory requirements of the application, guest OS, and hypervisor. (3) We experimentally prove that a triangle cache is primarily required for instructions, and interference at the L1 cache is less of an issue. (4) We propose the *TriKon* architecture that has 32 cores, and different cores have different configurations for their L1 and I caches. Some have small caches, some have large caches, and some have triangle caches. We show that using a sampling based scheduling algorithm, it is possible to improve the performance of a suite of benchmarks by 12% over a homogeneous baseline system with 32 cores and a conventional memory system. In this case the area of the *TriKon* processor is within 2% of that of the baseline system. We further show that it is possible to get an improvement in performance of 16% and 18%, if we increase the area of the *TriKon* processor by 5% and 10% respectively.

We begin by studying the various characteristics of our benchmarks in Section III, and then based on our characterization, we design our *TriKon* processor and the triangle cache in Section IV. We evaluate the performance of our scheduling algorithm in Section V, and finally conclude in Section VI.

II. Related Work

The major focus of hypervisor research has been to mitigate or analyze the performance overheads in a hypervisor with everyone agreeing on the fact that virtualization has the potential to be used in large scale systems in the future and there is a need to look for solutions to bring its performance on par with a non-virtualized system. Xen is one such hypervisor which is being widely used and is our focus in this paper.

A. Analysis and Mitigation of Performance Overheads of Hypervisors

Menon et al. [5] and Cherkasova et al. [6] have analyzed the overheads of the open source Xen hypervisor. The authors of [5] have designed a tool, Xenoprof, which can be used to measure the performance overhead of applications running on Xen. They demonstrate a roughly 20% decrease in system throughput for networking applications. Cherkasova et. al. [6] designed a similar tool for measuring the overhead of CPU usage and found a 4-20% overhead. Our conclusions are similar.

SplitX [11] dedicates a core (or set of cores) for running hypervisor tasks. The communication between the guest and the hypervisor cores happens through an inter-processor interrupt based mechanism that does not require the application to switch its context. ELI [12] assigns some devices to the guest so that the guest does not need to invoke the hypervisor to process interrupts from the devices. It also proposes *exitless* mechanisms to pass messages from the guest operating system to the hypervisor without a context switch. Elvis [13] uses this exitless mechanism and also dedicates a set of cores for hypervisors. It proposes a novel shared memory based communication mechanism between the guest OS and the hypervisor. Jin et al. [14] have considered page coloring based approaches to partition the shared L2 cache among various virtual machines for eliminating the interference in the shared cache among the cores' data and instructions. We did not find a substantial benefit by dedicating a set of cores for the hypervisor tasks because it took a lot of time to transfer data between the cache of the core that invoked the hypervisor to the core that is running the hypervisor. Secondly, page coloring based approaches were also not very beneficial because the memory footprint of a hypervisor keeps changing with time, and additional flexibility is essential.

B. Scheduling of Hypervisor Tasks

Most of the related work in scheduling hypervisor tasks is in scheduling tasks on a large multicore processor with different types of cores (known as an asymmetric multicore processor). Kwon et al. [10] have designed an asymmetry aware scheduler for the Xen hypervisor where they make the default Xen scheduler aware of the underlying asymmetry between the cores and subsequently use various heuristics such as IPC and utilization for arriving at a scheduling decision. Fedorova et al. [15] propose to use fast cores for CPU intensive tasks and slower (low power) cores for I/O intensive tasks. Li et al. [8] have designed a heterogeneity aware scheduler, *AMPS*, which has advanced load balancing

[1] *TriKon* means a triangle in Sanskrit

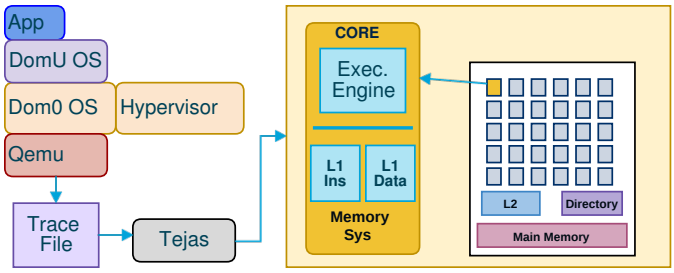

Figure 1: Simulation framework

features. For finding the suitability of an application to a core, proposals either use predictive models [8], or use a sampling based approach, where the application is run on a set of cores to find the best fit [7].

III. Characterization

A. Experimental Setup

Sr. No.	Name	Benchmark Suite	Type
1.	apache	[16]	Network server/client
2.	bodytrack	Parsec	Computer vision
3.	calculix	SPEC CPU 2006	Finite element analysis
4.	cpu_sb	Sysbench	Prime numbers
5.	fileio_sb	Sysbench	Random read/write ops
6.	fmm	Splash2	N-body simulation
7.	iozone	Iozone	File read/writes
8.	memory_sb	Sysbench	Sequential mem. access
9.	mummer	BioBench	Genome realignment
10.	mysql_sb	Sysbench	OLTP database workload
11.	netperf	[17]	Network performance
12.	pbzip2C	[18]	Parallel compression
13.	pbzip2D	[18]	Parallel decompression
14.	radiosity	Splash2	Diffuse calculations
15.	specjbb	SPECjbb 2005	Java app. server
16.	threads_sb	Sysbench	Multithreaded scheduler

Table I: List of benchmarks

1) Benchmarks: We show a list of all the benchmarks in Table I. We have chosen a suite of 16 benchmarks from the BioBench [19], Sysbench [20], Splash2 [21], Parsec [22], Specjbb2005, and SpecCPU2006 [23] suites. We have a mix of both CPU intensive and I/O intensive benchmarks.

2) Architectural Simulation: The simulation framework for our evaluations is described in Figure 1. We use the full system emulator, Qemu [24], to get a trace of executed instructions, branch outcomes, and memory addresses for the entire system consisting of the applications, guest OS, and the Xen hypervisor. We instrumented Qemu to write the traces to a file. We also modified the guest kernel (Debian/Squeeze 6.0.1) to generate specific interrupt signals on a CPL switch and DOM(domain/privilege level) switch. In order to emulate a multi-core system for a multi-threaded benchmark, we use used the smp (simultaneous multi-processor) mode in Qemu.

Parameter	**Value**	**Parameter**	**Value**
Cores	16	Technology	22nm
Frequency	3.2GHz	V_{dd}	1V
Pipeline			
Retire Width	4	Integer RF (phy)	160
ROB Size	168	Predictor	GaG
IW Size	54	Bmispred penalty	14 cycles
LSQ Size	64	Float RF (phy)	160
iTLB	128 entry	dTLB	128 entry
Integer ALU	4 units	Int ALU Latency	1 cycles
Integer Mul	1 unit	Int Mul Latency	2 cycles
Integer Div	1 unit	Int Div Latency	4 cycles
Float ALU	2 units	Int ALU Latency	2 cycles
Float Mul	1 unit	Int Mul Latency	4 cycles
Float Div	1 unit	Int Div Latency	8 cycles
L1 i-cache, d-cache			
Write-Mode	Write-Back	Block Size	64
Associativity	8	Size	32 kB
Latency	3 cycles		
Coherence	Directory based MOESI (fully mapped, 256 KB, 8-way)		
Shared L2			
Write-Mode	Write-Back	Block Size	64
Associativity	8	Size	4096 kB
Avg. Latency	28 cycles		
Main Memory and NOC			
Latency	200 cyc	Memory Controllers	2
NOC	2-D Mesh	Flit Size	16 bytes
Routing	XY	Router + Hop latency	3 cycles
Xen Version	4.0.1		
DomU OS	Debian GNU/Linux 6.0.1 squeeze		
Dom0 OS	Debian GNU/Linux 6.0.1 squeeze		

Table II: Details of the baseline system

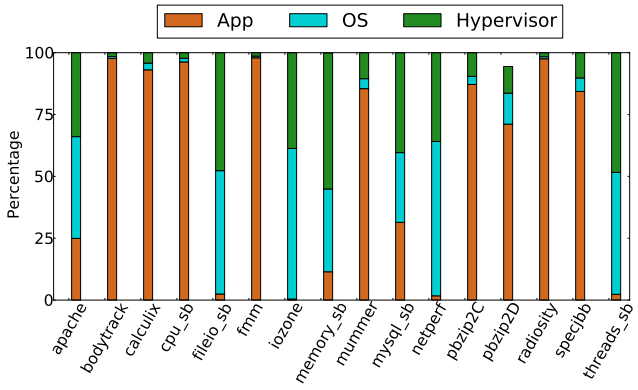

Figure 2: Instruction mix (app, OS, hypervisor)

We subsequently, feed the traces to the Tejas [25] simulator that is a detailed cycle accurate simulator for multicore processors. We use the sequential mode (for enhanced accuracy). The details of our simulated system are shown in Table II. We have a private L1 cache at each core and a shared L2 cache with a directory for coherence.

B. Instruction Mix

Figure 2 shows the breakup of instructions between the application, OS, and hypervisor. The OS instructions

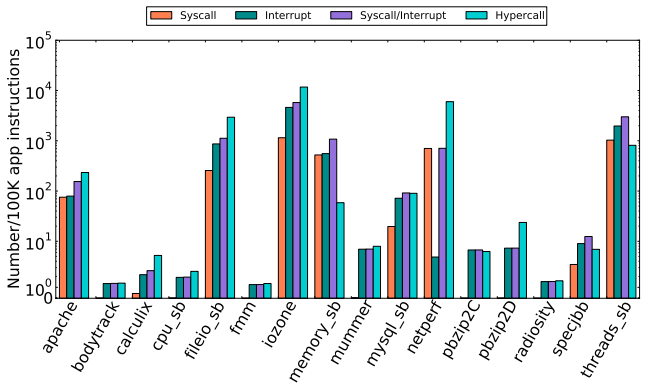

Figure 3: Number of Syscalls, Interrupts, Hypercalls

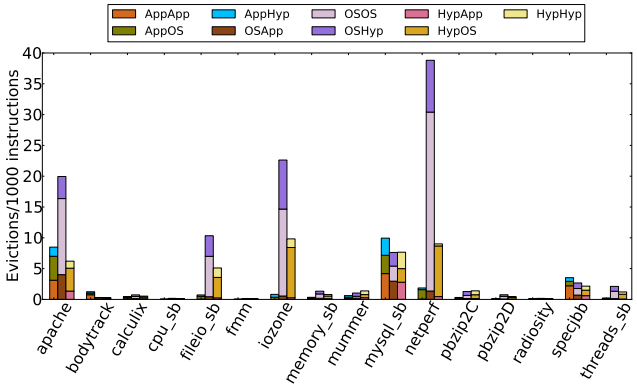

Figure 4: Evictions in the *Instruction Cache*

include both the Dom0 (part of the OS that runs with the hypervisor's privilege level) and DomU kernel instructions. It must be noted that all the benchmarks with a high fraction of OS instructions also have a significant amount of hypervisor activity. $iozone$, $fileio\_sb$, $netperf$, $memory\_sb$, and $threads\_sb$ have a very low fraction of application instructions ($< 10\%$). $apache$ and $mysql\_sb$ have 25-30% application instructions. Benchmarks such as $mummer$, $pbzip2C$, $pbzip2D$, and $specjbb$ have 75-80% application instructions. The rest of the applications: $bodytrack$, $calculix$, cpu_sb and $radiosity$, are fairly CPU intensive.

C. Characterization of OS/Hypervisor Events

Let us now try to understand the overhead. Figure 3 shows the number of system calls, interrupts, and hypercalls (calls made by the guest OS to the hypervisor) for a representative run of 100,000 instructions for each benchmark. It is important to note that the y-axis is in the log scale, and the third bar shows the cumulative sum of system calls and interrupts. The system intensive benchmarks have a very high number of system calls and interrupts with $iozone$ reporting as many as 10,000 system calls and interrupts. In contrast, the compute intensive applications have significantly lesser number of system calls and interrupts with most benchmarks registering less than 10 system calls.

D. Instruction Cache Evictions

There are two negative aspects of the hypercalls. The first is that they steal CPU time, and the second is that the hypervisor code displaces application/OS code and data from the caches [26]. This causes cache pollution, and results in slowdowns. Our main aim in this paper is to reduce this destructive interference.

Figure 4 shows the number of evictions per 1000 application instructions. We need to understand that we can have nine types of evictions as shown in Figure 4. For example, the $AppOS$ category refers to the OS lines that are evicted by the application. Similarly, the $HypOS$ category refers to the OS lines evicted by the hypervisor. We see that in system intensive benchmarks, most of the evictions (15-30/1000)are in the $OSOS$ category. The next two contributors are the $OSHyp$ and $HypOS$ categories (5-10/1000). For CPU intensive benchmarks the eviction rate is very low ($< 2/1000$). We further need to note that in benchmarks with moderate amount of system activity such as $apache$, $mysql\_sb$ and $specjbb$, $AppApp$ evictions account for less than 20% of the total evictions. We can thus conclude that it is necessary to design a sophisticated memory system that avoids collisions between the OS, hypervisor, and application instructions.

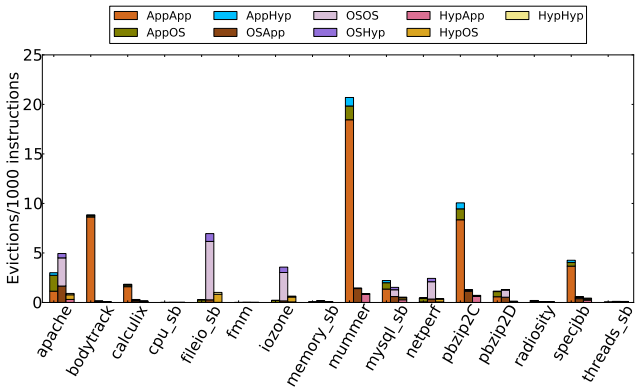

Figure 5: Evictions in the *Data Cache*

E. L1 Data Cache Evictions

Figure 5 shows the evictions in the L1 data cache. The scenario of cache line evictions in the data cache is quite dissimilar to the evictions in the instruction cache. Here, the application, and the operating system displace lines mostly belonging to their own context. There is less interference between the application, the OS, and the hypervisor.

Evictions due to the application have more than a 60% share in the more CPU intensive applications such as $bodytrack$, $pbzip2C$, and $specjbb$. For four benchmarks

(cpu_sb, fmm, $radiosity$, $threads\_sb$), the data cache misses in our representative run are very small given the high degree of temporal locality in these benchmarks. For 4 system intensive benchmarks namely $apache$, $fileio\_sb$, $iozone$, and $netperf$, the number of evictions due to the OS is the major source (3-7/1000) of cache line evictions.

The other important point to note here is that the absolute number of evictions in the data cache is roughly about half the number for that of the instruction cache (0-20 versus 3-45). This suggests that the the amount of destructive interference in the data cache due to guest OS and hypervisor data is minimal as compared to the instruction cache. Hence, we need to focus our efforts on optimizing the *instruction cache* in the *TriKon* processor.

IV. Implementation

As we observed in Section III, there is a significant amount of interference between the application, OS and hypervisor in the instruction cache. To prevent this, we would ideally want each context (application, OS, hypervisor) to have a separate cache. However, substituting a single instruction cache with three caches of the same size will have a prohibitive area overhead. Additionally, there might be phases in the application execution where the instruction footprint of a particular context exceeds the capacity of its designated cache due to which there is a high penalty to fetch the cache line from the lower level cache.

Thus, our aim is to design a memory system that (a) provides an abstraction of a separate cache to the application, OS, and hypervisor, (b) simultaneously minimizes the area overhead, and (c) allows each context to use the free space in other caches, if its instruction footprint exceeds the capacity of its designated cache.

A. Advanced OS Cache

Let us start by proposing to have two caches instead of a single instruction cache: an application cache for the application, and a separate cache for the OS/Hypervisor. This scheme helps us in completely eliminating the interference between the working sets of the application and the OS/Hypervisor. Although this naive cache structure is guaranteed to have a better performance than the baseline design, this approach is evidently wasteful. Clearly, the application, the OS, and hypervisor are not all running at the same point of time. Hence, most of the time, only one cache is being used. Secondly, we are doubling the area of the L1 cache (instruction or data).

To avoid such problems, we had proposed the advanced OS cache scheme in our prior work [26]. This scheme proposes two caches. Memory requests from the application go to the application cache, and memory requests from the OS go to the OS cache. However, if the working sets of the application or OS exceed their respective cache sizes, then it is possible for them to steal some lines from the other cache. For example, if the application needs to evict a line from the application cache, then it sends it to the OS cache. Any read access issued by the application first searches the application cache, and then the OS cache. We had implemented this method for the L2 cache, and demonstrated speedups for a suite of multi-threaded programs.

B. Triangle Cache

1) Design: In this paper, we propose the triangle cache that extends the advanced OS cache scheme by incorporating three caches (one each for the hypervisor, OS, and application). Secondly, we design a more elaborate protocol that supports different sizes for each constituent subcache, and novel methods to search, replace, and evict blocks.

Access Protocol : The access protocol uses the CPL (current privilege level) bits to route the request to the appropriate sub-cache in the triangle cache. In x86 based systems the application, guest OS, and the hypervisor run at privilege levels 3, 1, and 0 respectively.

If the current privilege level is equal to 3, then the memory request of the core is routed to the application cache. If there is a hit, then the data is supplied to the core. Otherwise, it is necessary to check the rest of the caches. Naively checking the rest of the caches on every miss will lead to a lot of contention and wasteful consumption of dynamic power. Hence, we propose an optimization.

Along with the data and tag arrays of each sub-cache, we maintain an array called the *set-array*. The set-array contains an entry for each set in the sub-cache. Each entry contains information that lets the memory system decide whether there is a possibility of the other sub-caches having a line that belongs to this set. The entry is organized as follows. Let us explain this with an example illustrated in the Figure 6. Let us assume that the application cache has 1 set with 4 lines, the OS cache has 4 sets with 1 line each, and the hypervisor cache has 2 sets with 2 lines each. Each cache has the same cache line size. We observe that each set in the application cache maps to 4 sets in the OS cache and 2 sets in the hypervisor cache. Thus, we create an entry in the set-array (corresponding to the application cache) with 6 sub-entries as shown in the Figure 6. Each sub-entry maps to one set of the OS or hypervisor cache and is a single bit 0/1. Each bit indicates whether the corresponding set in the OS or hypervisor cache contains an application line or not. In a similar manner we have set-arrays in the rest of the sub-caches.

If all the entries in the set-array are 0, then the request can be directly sent to the L2 cache. If one of the entries is equal to 1, then the request is directly sent to that cache. Note that it is not necessary to compute the index of the tag array in the target sub-cache that the request has been sent to. This is because this computation is already done. Now, if the data is not available in the target sub-cache, then we can declare a L1 miss. Now, let us assume that

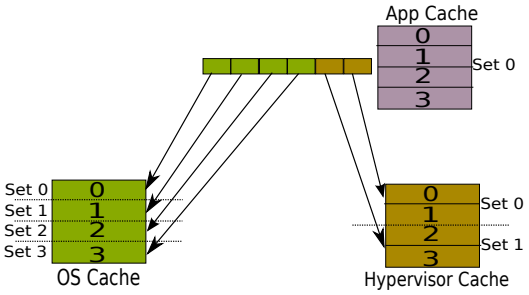

Figure 6: An entry of the set-array in the application cache

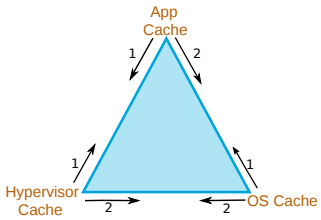

Figure 7: Search order of caches

the set-array indicates that the line might be present in both the sub-caches. It is then necessary to check the sub-caches in a certain order. We use the order shown in Figure 7. For the application cache, we check the hypervisor cache first, and the OS cache later. The reason for this is that there is a large amount of destructive interference between application and OS lines as compared to application and hypervisor lines (see Figure 4). Note that the scheme shown in Figure 7 represents a static scheme. We can make the scheme dynamic by monitoring the destructive interference between different classes of lines. In our experiments, a dynamic scheme was not found to be significantly better; hence, we preferred the static approach as shown in Figure 7.

Eviction and Replacement Policy : We use a replacement policy, which gives higher priority to native lines than to foreign lines (e.g. the application sub-cache will try to evict a cache line belonging to the hypervisor/OS before evicting an application line). The algorithm for evicting a line is as follows. If we evict a line from its native sub-cache (e.g. the application line in the case of application sub-cache), then we send it to the sub-cache that is its first preference (as shown in Figure 7). If it is evicted from there also, then we send it to the second preference of its native sub-cache. When we evict a foreign line, we check the rest of the lines in the set to find out if all the foreign lines of the same class are evicted. If it is the case, then we set the corresponding bit to 0 in the set-array of the native sub-cache of the evicted foreign line.

Cache Coherence : Since instructions are typically not written to in the middle of the execution of the program, the need for cache coherence is minimal (unless we have self modifying code). However, to support just-in-time compilers, we propose to make the triangle cache participate in the MOESI based directory protocol in the following fashion. When a cache receives an invalidate request, it first checks the native cache of a line, and checks the sibling sub-caches if necessary. If it finds the entry, it invalidates it. Other directory events are processed in a similar fashion.

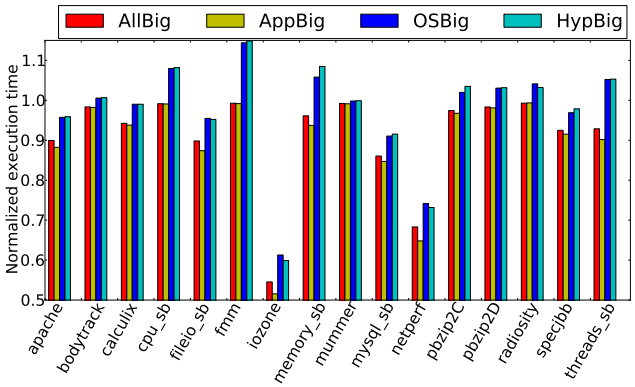

Figure 8: Variations in the size of the caches in the triangle cache

2) Area Overhead: The cache in our baseline design is a 32 KB, 8 way associative cache (referred to as 32x8). Having three big (32x8) caches in a triangle cache represents a large area overhead. Hence, we experiment with one big cache, and two small caches (32x4) connected in a triangle configuration for the instruction cache. To justify this design decision, we consider other variants of triangle caches, and perform simulations to understand their execution time(see Figure 8). The performance is normalized to a baseline design without triangle caches. $appBig$ (design that we choose) represents the case when the application cache is big and the others are small. In the $allBig$ configuration, all the caches have the 32x8 configuration. Different configurations have different latencies (obtained using Cacti 5.3). We observe that the $appBig$ configuration is the best (outperforms $allBig$ by 2%).

Let us now compare the triangle cache ($appBig$) with the advanced OS cache [26], and a triangle cache without the provision of migrating lines between the sub-caches (*triple* cache). Figure 9 shows the results normalized to the performance of the baseline design. The triangle cache clearly performs better for all the benchmarks especially for the OS intensive ones with $iozone$ reporting a 45% improvement in performance. This is due to the large working set of the OS, and the high degree of destructive interference between the OS and the application. For other OS intensive benchmarks too, the average performance increase is more than 15%. We show the hit rate for various sub-caches in the triangle scheme in Table III. The high performance of $netperf$ and $iozone$ can be attributed to the fact that a large number of OS and hypervisor requests overflow to the application

Benchmark	App Cache			OS Cache			Hypervisor Cache		
	App	DomU OS	Hypervisor	App	DomU OS	Hypervisor	App	DomU OS	Hypervisor
apache	87.80	4.36	5.92	0.00	74.86	0.23	2.23	14.28	97.00
bodytrack	99.36	9.77	19.34	85.58	95.49	47.44	75.76	65.81	98.79
calculix	99.89	28.54	57.23	34.76	91.80	2.57	33.79	22.92	96.76
cpu_sb	100.00	73.00	88.04	0.61	95.17	3.71	4.87	73.59	98.92
fileio_sb	99.21	42.13	53.88	0.21	92.24	0.78	0.59	17.04	97.42
fmm	100.00	41.49	77.32	3.91	97.17	4.01	2.06	80.49	99.19
iozone	99.91	94.06	92.12	0.00	86.13	0.00	0.00	51.29	98.75
memory_sb	99.98	80.62	59.77	6.57	98.85	1.22	1.93	59.67	99.85
mummer	99.99	39.24	86.96	5.95	93.29	0.44	0.00	51.63	97.08
mysql_sb	92.46	3.22	36.40	2.59	92.25	17.73	7.96	22.31	98.13
netperf	99.94	89.51	70.96	0.00	70.45	0.00	0.00	60.00	99.68
pbzip2C	99.99	73.14	92.18	4.24	95.24	5.23	0.86	63.98	97.67
pbzip2D	99.99	67.05	52.52	1.21	94.46	1.42	1.37	51.42	97.71
radiosity	100.00	43.25	64.42	17.05	96.84	20.71	7.75	75.87	99.08
specjbb	97.24	3.64	5.97	14.82	92.86	13.21	15.00	21.51	97.63
threads_sb	99.75	95.56	89.72	5.55	98.72	5.34	19.09	66.58	99.90

Table III: Instruction cache hit rate for various environments in the different caches

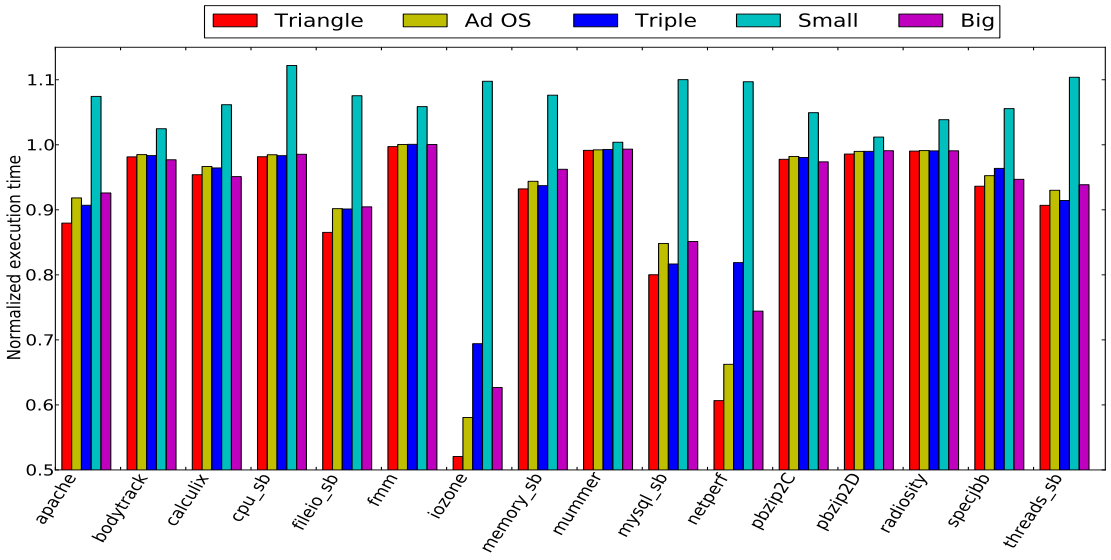

Figure 9: Comparison of a system with a triangle cache with other alternatives

cache with most of them being hits.

C. Small and Big Caches

We also consider the impact of having large caches (64x8) and small caches (32x4) in our system. We compare them with the triangle and advanced OS cache. We want to create a system that is a mix of cores with small, large, and triangle caches such that its area is roughly equal to that of our baseline system.

Figure 9 shows the comparison in the execution time of small and large caches against the triangle and advanced OS cache normalized to the baseline design. Compute intensive applications such as $pbzip2D$, $bodytrack$, and $calculix$ have acceptable performance with small caches and thus can be scheduled on such a core. For the OS intensive applications such as $apache$, $iozone$, $fileio\_sb$, and $mysql\_sb$, large caches are able to give sufficient performance improvements. Note that large caches are smaller than triangle caches as we shall show in Section V. Hence, they can be considered a worthy alternative for some benchmarks.

V. Heterogeneous System Design

A. Scheduling Algorithm

Let us now outline the details of a hardware based scheduling algorithm that takes into account the heterogeneity of cores. Our basic algorithm is $SampleAndSchedule$ [7]. The execution of the application is divided into two phases namely $Sample$ and $Schedule$. We start with a random assignment of threads to cores. During the $Sample$ phase, each thread is run on each type of core for at least one epoch. The IPC of each <*thread,core*> mapping is recorded

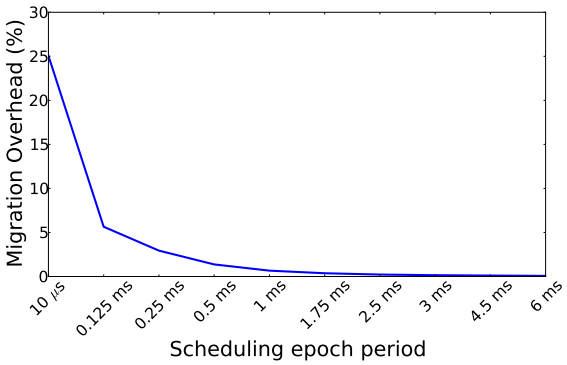

Figure 10: Performance overhead of thread migration

Size (KB)	Assoc	Latency (Cycles)	Area (mm^2)
16	4	2	0.326
16	8	4	0.569
32	4	3	0.361
32	8	4	0.641
64	4	3	0.606
64	8	4	0.704

Table IV: Area and latency

at the end of every epoch. After the sampling phase is over, the global scheduler needs to decide the optimal mapping of threads to cores. We order the *<thread,core>* tuples in descending order of improvement in terms of relative IPC with respect to a baseline core. We pick the best mapping and if this mapping is unavailable due to a lack of cores, we move to the next mapping.

The epoch period for the sampling phase determines the efficacy of the scheduling algorithm. After each epoch, threads are migrated from one core to another. Once a thread switches from one core to another, its data needs to be transferred to the new core. Figure 10 shows this migration overhead (in terms of loss in IPC due to the transfer of cache lines) as a function of the epoch size. We observe that an epoch size of less than 1 ms is not advisable. These observations are in line with the conclusions drawn by Craeynst et al. [27]. We choose an epoch size of 3 ms.

B. System Setup

Table IV mentions the area, and latency of different L1 cache configurations that we have used for our experiments. The numbers have been calculated using Cacti 5.3 [28]. Scaling to 22nm technology is done using the techniques described in [29].

We have evaluated four different core designs : $Small$, $Baseline$, Big, and $Triangle$. The detailed design of the $Baseline$ core is shown in Table II. Other core designs differ from the $Baseline$ core only in respect of the configuration

Core Type		Size (KB)	Assoc
Baseline (I/D cache)		32	8
Triangle	App (I cache)	32	8
	OS (I cache)	32	4
	Hypervisor (I cache)	32	4
	D cache	32	8
Small (I/D cache)		32	4
Big (I/D cache)		64	8

Design	Area Overhead (%)	Baseline	Small	Triangle	Big
Baseline	-	32	-	-	-
Design A	2	-	13	13	6
Design B	5	-	10	15	7
Design C	10	-	0	22	10

Table V: Design Space: Heterogeneous System

of the instruction and data cache. The cache configuration of each core design is shown in Table V.

We consider three core designs for our heterogeneous system, *TriKon*: $Small$, Big and $Triangle$. The $Triangle$ core design is suited for applications, which experience a significant amount of destructive interference between the hypervisor, application, and OS. The Big core design is suited for applications, which have a large memory footprint and are unable to accommodate their working set in the baseline cache. The $Small$ core design has cores with smaller level 1 caches, and it is suited for compute intensive applications with small working sets. It does not provide extra performance benefits, but it dissipates less power and consumes lesser area than the other 2 cache designs. We use a system of cores with varying cache sizes to ensure that the proposed heterogeneous system has roughly the same area as the baseline design.

We evaluate three different configurations as shown in Table V: $Design\ A$, $Design\ B$, and $Design\ C$. $Design\ A$ has the least area overhead of 2% as compared to a baseline homogeneous system of 32 cores, whereas $Design\ B$ and $Design\ C$ have an area overhead of 5% and 10% respectively.

C. Evaluation

We consider two sets of applications for our evaluation: $Mix1$ and $Mix2$. Each of these application mixes contain a subset of applications from our original list of 16 applications (see Table I). Note that we have both single threaded as well as multi-threaded applications. Both our application mixes contain a total of 32 threads (albeit from different applications). The performance results are shown in Figure 11 and 12 respectively.

$Mix1$ has a higher number of applications with large amounts of destructive cache interference and memory footprints. Predictably, as we increase the area budget and move towards larger/sophisticated cache designs, the overall performance of the system increases. Since $Design\ A$ has

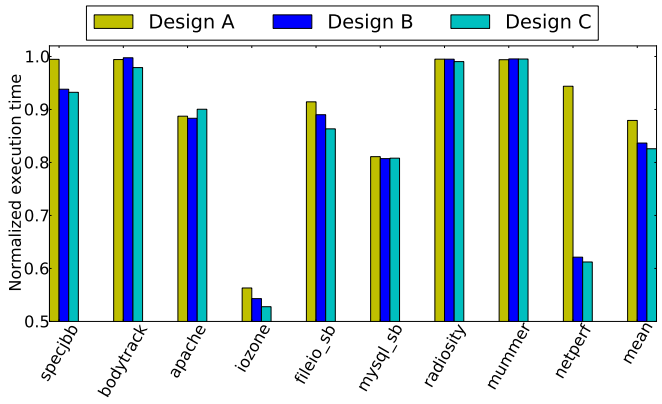

Figure 11: Heterogeneous scheduling results: Mix1

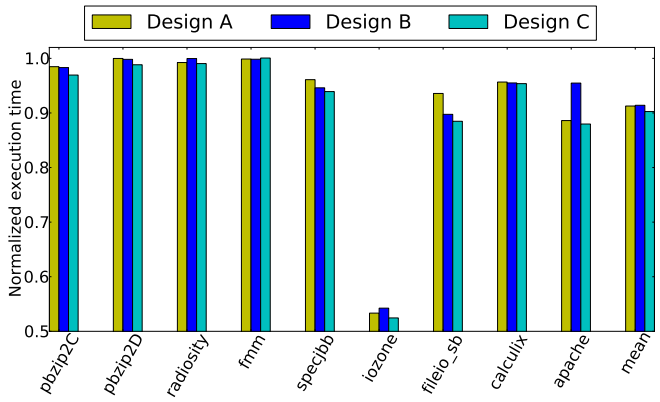

Figure 12: Heterogeneous scheduling results: Mix2

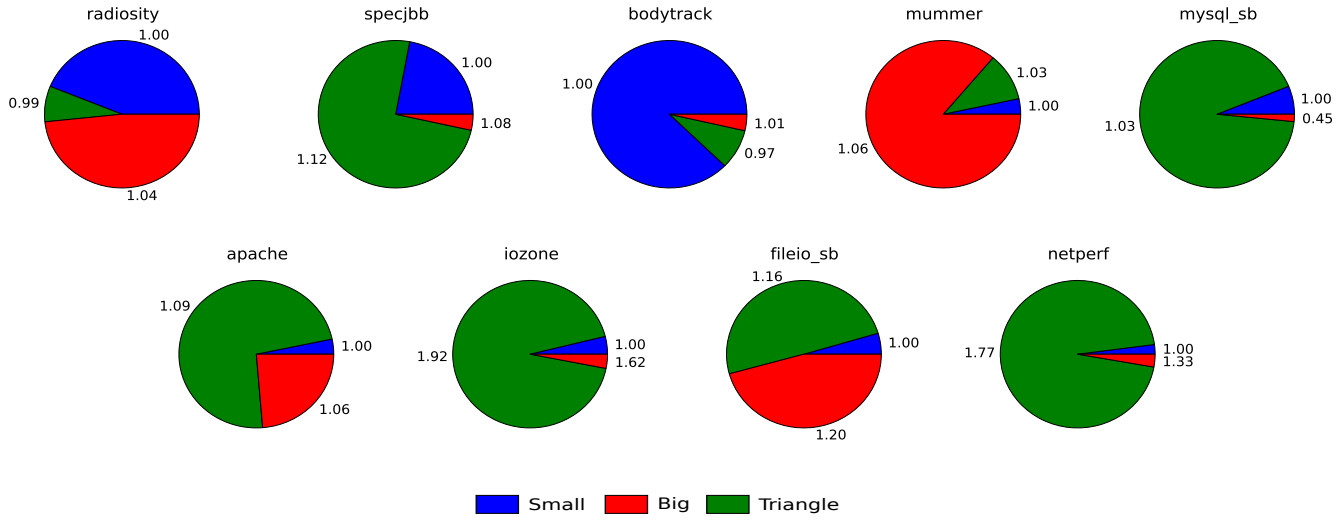

Figure 13: Fraction of time applications spend on different core designs

fewer cores with $Triangle$ caches, benchmarks such as $iozone$, $apache$ and $mysql\_sb$ could leverage the $Triangle$ core design. The average performance improvement for these benchmarks is 20%. However it was not possible to allocate $Triangle$ cores to benchmarks such as $netperf$, which was mapped to a Big core in $Design\ A$. When we moved from $Design\ A$ to $Design\ B$, $netperf$ was allocated to $Triangle$ cores, and consequently its performance increased by 30%. In contrast, for the compute intensive benchmarks such as $radiosity$ and $bodytrack$, nearly the same performance is observed for all the designs reiterating the fact that these benchmarks can be scheduled on any core without any appreciable performance degradation. It must be noted that few benchmarks benefited when we moved from $Design\ A$ to $Design\ B$ (such as $specjbb$ and $netperf$). These benchmarks were primarily system intensive and required $Triangle$ caches. However, we do not see an appreciable improvement when we invest another 5% in the area overhead and move from $Design\ B$ to $Design\ C$. This is because all the system intensive benchmarks get cores with Big or $Triangle$ caches.

In comparison, $Mix2$ has a higher number of compute intensive applications. Hence, we do not notice any significant performance variation with increasing area budgets.

D. Mapping of Benchmarks to Cores

Figure 13 shows the fraction of time each benchmark from the $Mix1$ set of applications spent on different cores of $Design\ B$ along with its IPC relative to the $Small$ core. Application intensive benchmarks such as $radiosity$ and $bodytrack$ spend most of their execution time on $Small$ cores. System intensive benchmarks such as $iozone$, and $netperf$ need a sophisticated cache design for mitigating interference and hence they are mapped to a $Triangle$ core.

Another point to be noted is that benchmarks such as $radiosity$, and $fileio\_sb$ have their execution times evenly

distributed over multiple core designs, which is due to the fact that different threads of these benchmarks are mapped to different core designs and we do not distinguish between the individual threads in these pie-charts. Also, we can observe that application intensive benchmarks such as $radiosity$, $bodytrack$, and $mummer$ do not show any appreciable improvement in relative IPC for any of the core designs. In contrast, $iozone$ and $netperf$ which are highly system intensive have an increase of 60% and 92% respectively in their relative IPC on the $Triangle$ design.

VI. Conclusion

In this paper, we looked at the design of the *TriKon* processor for executing applications in an on-chip cloud environment. We started by characterizing the benchmarks and depending on their behavior, they were classified into three basic categories (1) system intensive applications with high memory interference between the application, OS and hypervisor, (2) applications with a large memory footprint, and (3) compute intensive applications. We tackled the problem of high memory interference by segregating the memory accesses of the different contexts into three different caches: application, OS, and hypervisor. We assigned a cache to each entity and designed a dynamic scheme, which allows data to flow among sibling caches without significant amount of destructive interference. We also optimally size this $Triangle$ cache, and experimentally demonstrate that it is required for instructions (not for data).

We proceed to design a processor with $Small$, Big, and $Triangle$ cores. We used a sampling based scheduling algorithm to map different application threads to different types of cores. We designed a heterogeneous system with an area overhead of 2% and obtained a performance improvement of 12% for a mix of system, memory, and compute intensive applications. Among these applications, we observed an average gain of 20% for the system intensive benchmarks while the performance of compute intensive benchmarks remained unaffected. Also, by allocating extra area for the $Triangle$ and Big cores, we improved the performance of the same system intensive benchmarks by further 10%.

References

[1] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, Sept 2007.

[2] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob *et al.*, "An 80-tile 1.28 tflops network-on-chip in 65nm cmos," in *IEEE Solid-State Circuits Conference*, 2007.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM Operating Systems Review (SIGOPS)*, vol. 37, no. 5, 2003.

[4] C. A. Waldspurger, "Memory resource management in vmware esx server," *ACM Operating Systems Review (SIGOPS)*, vol. 36, 2002.

[5] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the xen virtual machine environment," in *ACM/USENIX International conference on Virtual execution environments*, 2005.

[6] L. Cherkasova and R. Gardner, "Measuring cpu overhead for i/o processing in the xen virtual machine monitor." in *USENIX ATC*, vol. 50, 2005.

[7] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," in *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2, 2004.

[8] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *ICS*, 2007.

[9] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "Hass: a scheduler for heterogeneous multicore systems," *ACM Operating Systems Review (SIGOPS)*, vol. 43, no. 2, pp. 66–75, 2009.

[10] Y. Kwon, C. Kim, S. Maeng, and J. Huh, "Virtualizing performance asymmetric multi-core systems," in *ACM Computer Architecture News (SIGARCH)*, vol. 39, no. 3, 2011, pp. 45–56.

[11] A. Landau, M. Ben-Yehuda, and A. Gordon, "Splitx: Split guest/hypervisor execution on multi-core," in *USENIX Workshop on I/O Virtualization (WIOV)*, 2011.

[12] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "Eli: bare-metal performance for i/o virtualization," *ACM Computer Architecture News (SIGARCH)*, vol. 40, no. 1, pp. 411–422, 2012.

[13] A. Gordon, N. Har'El, A. Landau, M. Ben-Yehuda, and A. Traeger, "Towards exitless and efficient paravirtual i/o," in *International Systems and Storage Conference*, 2012, p. 10.

[14] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, and X. Li, "A simple cache partitioning approach in a virtualized environment," in *IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2009.

[15] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto, "Maximizing power efficiency with asymmetric multicore systems," *ACM Communications*, vol. 52, no. 12, pp. 48–57, 2009.

[16] R. T. Fielding and G. Kaiser, "The apache http server project," *IEEE Internet Computing*, vol. 1, no. 4, pp. 88–90, 1997.

[17] R. Jones, "Netperf," *http://www. cup. hp. com/netperf/NetperfPage. html*, 1996.

[18] J. Gilchrist, "Parallel bzip2," http://compression.ca-/pbzip2/.

[19] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "Biobench: A benchmark suite of bioinformatics applications," in *ISPASS*, 2005.

[20] A. Kopytov, "Sysbench: a system performance benchmark," 2004.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ACM Computer Architecture News (SIGARCH)*, vol. 23, no. 2, 1995, pp. 24–36.

[22] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *PACT*, 2008.

[23] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM Computer Architecture News (SIGARCH)*, vol. 34, no. 4, pp. 1–17, 2006.

[24] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[25] G. Malhotra, P. Aggarwal, A. Sagar, and S. R. Sarangi, "ParTejas: A parallel simulator for multicore processors," in *ISPASS*, 2014.

[26] S. Chandran, P. Kallurkar, P. Gupta, and S. Sarangi, "Architectural support for handling jitter in shared memory based parallel applications," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 5, May 2014.

[27] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *ISCA*, 2012.

[28] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi, "Cacti 5.3," *HP Laboratories, Palo Alto, CA*, 2008.

[29] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron, "Scaling with design constraints: Predicting the future of big chips," *IEEE Micro*, vol. 31, no. 4, 2011.