pTask: A Smart Prefetching Scheme for OS Intensive Applications

Prathmesh Kallurkar*, Smruti R. Sarangi†
Department of Computer Science, Indian Institute of Technology, New Delhi, India
Email: {prathmesh.kallurkar*, srsarangi†}@cse.iitd.ac.in

***Abstract*—Instruction prefetching is a standard approach to improve the performance of operating system (OS) intensive workloads such as web servers, file servers and database servers. Sophisticated instruction prefetching techniques such as PIF [12] and RDIP [17] record the execution history of a program in dedicated hardware structures and use this information for prefetching if a known execution pattern is repeated. The storage overheads of the additional hardware structures are prohibitively high (64-200 KB per core). This makes it difficult for the deployment of such schemes in real systems. We propose a solution that uses minimal hardware modifications to tackle this problem. We notice that the execution of server applications keeps switching between tasks such as the application, system call handlers, and interrupt handlers. Each task has a distinct instruction footprint, and is separated by a special OS event. We propose a sophisticated technique to capture the instruction stream in the vicinity of such OS events; the captured information is then compressed significantly and is stored in a process's virtual address space. Special OS routines then use this information to prefetch instructions for the OS and the application codes. Using modest hardware support (4 registers per core), we report an increase in instruction throughput of 2-14% (mean: 7%) over state of the art instruction prefetching techniques for a suite of 8 popular OS intensive applications.**

I. Introduction

Prefetching instruction streams to reduce i-cache misses is a standard approach for improving the performance of codes that have low i-cache hit rates. Most of the highly cited recent work [12], [13], [16], [17] in this area has focused on operating system (OS) intensive programs such as file servers, web servers, and database servers. This is primarily because such applications have a fair amount of interference between the application threads, and kernel threads. The reason for such interference can be attributed to frequent system calls, interrupts, and intense activity within the operating system's kernel. As a result, these programs have benefited from sophisticated i-cache prefetching algorithms. In comparison, serial codes such as the Spec CPU benchmarks [14] or numerically intensive parallel codes such as the Splash2 [30] and Parsec [8] suites are associated with relatively higher i-cache hit rates (98.5-99.9%) and thus do not stand to significantly benefit from advanced i-cache prefetching strategies.

Note that prefetching is not the only method of increasing the performance of system intensive workloads. Some other techniques include core specialization [23], [26] (dedicating a core to process interrupts and system calls), OS caches [5], [7], [11], [21] (dedicating a cache to store OS instruction/data), and DVFS techniques to compensate for OS induced non-deterministic execution [11]. Having a separate core, or a separate cache represents a large area overhead. DVFS based approaches that predict the loss in performance due to interference and subsequently try to boost the frequency for small durations of time are associated with appreciable power overheads. In comparison, state of the art prefetching based approaches have lesser overheads in terms of area and power.

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

Our aim in this paper is to consider a suite of standard system intensive benchmarks running on a multi-core processor and increase their performance by prefetching i-cache lines. We compare our proposed scheme, $pTask$, with state of the art techniques (RDIP [17], PIF [12]) in Section V, and demonstrate a roughly 2-14% improvement in performance (defined as instruction throughput). Moreover, $pTask$ outperforms classic schemes with low area overheads such as Markov [15], [22] and call graph based prefetching [3] by around 10%. We achieve these speedups with a minimal amount of additional hardware (2 256-bit registers and 2 integer registers). In comparison competing techniques have larger hardware overheads: 200 KB for PIF [12] and 64 KB for RDIP [17].

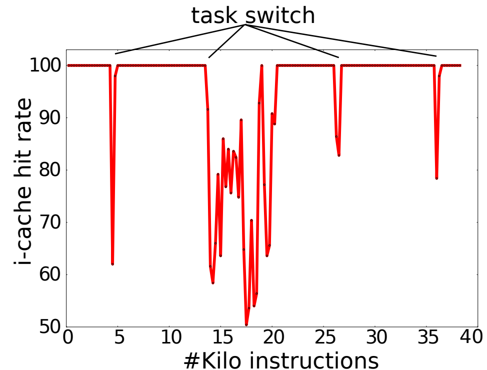

Fig. 1: Impact of task switches on the i-cache hit rate (Apache web server: representative execution)

The insight that we use to achieve a speedup with reduced storage overheads is shown in Figure 1. Figure 1 shows that any system intensive application can be divided into two distinct phases. The i-cache hit rates on a core are typically high during the steady state execution of application or OS threads. The misses peak whenever there is a context switch, or we switch between one type of tasks to another. An example of the latter will be a switch from an interrupt handler task to the OS scheduler. We propose a sophisticated method to capture instruction streams in the vicinity of these task switches, and ignore the rest of the execution, which in any

case has relatively higher i-cache hit rates. In comparison competing works are oblivious of the nature of tasks and treat the entire execution identically. This insight allows us to get better performance with reduced storage requirements.

We shall briefly discuss related work in Section II, discuss the characterization of benchmarks in Section III, move on to discuss the implementation of $pTask$ in Section IV, and finally conclude with evaluation results in Section V.

II. RELATED WORK

A. Mitigating Application/OS Interference

Additional Caches: The combined footprint of the OS and applications typically overwhelms the smaller private caches. [5], [7], [11], [21] replicate the existing cache; application lines are stored in the regular cache and the OS lines are stored in a special OS cache. The main drawback of this line of work is the 100% area overhead of adding an extra cache.

Core specialization: [9], [10], [21], [26], [29] reduce the OS-application interference by offloading the OS code execution to dedicated OS cores. This increases code locality and hence improves performance. However, Nellans et al. [21] show that the cost of moving application data to OS cores and back is prohibitive in most cases, and thus this approach is not the best choice for the system intensive benchmarks that we consider.

B. Instruction Prefetching

The work on hardware and software assisted prefetching is extensive. We discuss some of the sophisticated techniques here.

1) Hardware Based Prefetching: One of the earliest instruction prefetching techniques is *Nextline* prefetching. If a cache line with address x is not found in the i-cache, the Nextline prefetcher reads cache lines with addresses from $x + 1$ to $x + n$ from the lower level cache. This is a simple scheme and requires a negligible amount of additional area. However, Nextline prefetchers perform poorly in the presence of frequent jumps and function calls.

Markov prefetchers [15], [22], [27] rely on the correlations in the i-cache access sequence to predict future i-cache misses. They work very well for traditional single and multi-threaded programs. However, for OS-intensive applications whose execution is punctuated by a lot of non-deterministic events such as interrupts and context switches, the i-cache access sequence changes frequently. As a result, the i-cache miss sequence predictor performs poorly, and we shall see in Section V that such prefetching schemes are inferior to the scheme that we propose.

Recent hardware prefetching proposals [12], [13], [17] rely on the observation that the instruction miss sequences are highly repetitive and are often predictable. TIFS [13] records the order of instruction fetches in a dedicated per-core instruction buffer and uses other index based structures to map PCs of missed instructions to entries in the instruction buffers. Whenever there is a miss, we prefetch the set of addresses stored at the corresponding entry in the instruction buffers. PIF [12] improves over TIFS by recording instruction commit sequences instead of instruction fetch sequences. However, the storage requirement of its dedicated hardware units is around 200 KB per core. To put this number into perspective, the size of the instruction cache is 32 KB. A later paper (SHIFT [16]) reduces the area overhead of PIF by sharing the prefetch information across all the cores. However, this scheme gives lesser performance benefits and is suitable only for multi-threaded applications as mentioned in the original paper. RDIP [17] predicts future i-cache accesses using the function call sequence (expressed in the return address stack). The function call sequence is stored in a 64 KB per-core buffer. We compare our work against RDIP [17] and PIF [12] in Section V.

2) Software Based Prefetching: Typically, in software prefetching techniques, the compiler adds special *prefetch* instructions to the generated code. A plethora of sophisticated compiler techniques [3], [20], [28] have been proposed in this area. However, these techniques use offline profiling to generate the prefetch information. Offline profiling is not suitable for server applications where regular updates to the OS and the application can change the execution of an application significantly. Our scheme also uses software prefetch instructions; however, the actual addresses to prefetch are calculated using an online profiler. We compare our work against a seminal software prefetching technique, CGP [3] (call graph prefetching), in Section V.

III. CHARACTERIZATION

A. Definition of a HyperTask

OS Event	HyperTask	Execution Block
System call begin	System call handler	Code processing a system call request
System call end	Application	Application's execution between two consecutive system call requests
Hardware interrupt	Interrupt	Code processing an interrupt
Start of a bottom half handler's routine	Bottom half handler	Code processing a bottom half handler
Start of the *schedule* routine	Scheduler	Scheduler's code

TABLE I: Events related to HyperTasks

We broadly define a HyperTask as a piece of code that runs between two OS events. We identified five OS events to define HyperTasks: (a) system call begin, (b) system call end, (c) hardware interrupt, (d) start of a bottom half handler's routine, and (e) start of the *schedule* routine. Table I mentions the type of HyperTask and the actual code that is associated with each OS event.

We observed a pattern here: the code that executes immediately after an OS event remains more or less the same, each time it is invoked. Let us consider the *timer* interrupt. Each time the core receives a timer interrupt, it executes the routines related to the timer interrupt handler. We leverage this pattern to prefetch the instructions belonging to the timer interrupt handler. When the core receives the timer interrupt for the first time, *pTask* records the instructions executed by the timer interrupt handler. If the core receives the same interrupt again, *pTask* prefetches the instructions belonging to the interrupt handler. The entire process: identify the HyperTask, record its execution, and prefetch its instructions, is done at runtime (details in Section IV).

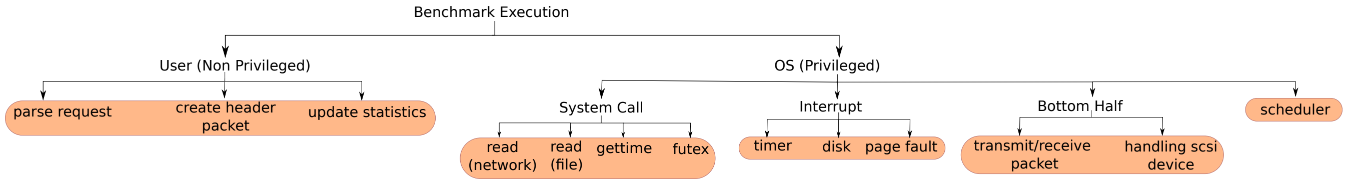

Fig. 2: Decomposition of the *Apache* benchmark's execution into HyperTasks

As an example, Figure 2 shows a simple decomposition of the *Apache* benchmark into its constituent HyperTasks. Broadly speaking, a HyperTask is supposed to be a block of code that is predictably fetched into the i-cache after an event of interest (as defined in Table I). Skeptics can argue that the code to (for example) handle a timer interrupt need not be exactly the same all the time, between two OS events we can have many other types of code executing such as OS book keeping code, and the definition is vague for applications. To ensure that HyperTasks are better defined in practice we place a limit on their footprint (typically 125 functions). We evaluate the efficacy of our definition in Section III-F, and observe that for most system intensive benchmarks it is possible to define HyperTasks in this manner. For applications, the code sequence after an OS event (such as a context switch) is roughly predictable because most of this code is actually inside a library or is a part of the application's code dedicated to preparing/processing system call data, and remains approximately identical across invocations.

B. Setup

We used the full system emulator, Qemu [6], to get the execution trace of the entire system (application+OS). The execution trace includes executed instructions, branch outcomes, interrupts/system calls, and memory addresses (loads/stores). We subsequently fed the traces to the Tejas [25] simulator, a detailed cycle accurate simulator for multi-core processors (fully validated against native hardware [24]).

Table II shows the details of our simulated system. We simulate a 16 core machine, where each core has a private L1 cache, and a shared L2 cache with directory based cache coherence.

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	GShare
IW Size	54	Bmispred penalty	14 cycles
LSQ Size	64	Float RF (phy)	160
iTLB	128 entry	dTLB	128 entry
L1 i-cache, d-cache			
Write-Mode	Write-Back	Block Size	64
Associativity	4	Size	32 kB
Latency	3 cycles	Port	2
Coherence	Directory based MOESI (fully mapped, 256 KB, 8-way)		
Shared L2			
Write-Mode	Write-Back	Block Size	64
Associativity	8	Size	4 MB
Latency	20 cycles		
Main Memory and NOC			
Latency	250 cyc	Memory Controllers	2
NOC	2-D Mesh	Flit Size	16 bytes
Routing	XY	Router + Hop latency	3 cycles
OS		Debian GNU/Linux 6.0.1 squeeze	

TABLE II: Details of the baseline system

C. Benchmarks

We shall evaluate the *pTask* scheme for a suite of 8 OS intensive applications. Some of these benchmarks are part of well-known benchmark suites such as Sysbench [18], Filebench [1], and TPC-H [2], while others are representative utility applications available on Linux. (1) *Apache* captures the execution of the Apache web server that services a set of static web-pages. In our experiments, the web-client requests 48 web pages at a time, meaning that each core serves on average 3 web pages at any point in time. (2) *DSS* captures the execution of a decision support system on large volumes of business data. Specifically, we execute `query 2` of the TPC-H benchmark on a 1 GB database. (3) *FileSrv* simulates the execution of a file server using Filebench. In this workload, 200 threads perform a sequence of creates, deletes, appends, reads, writes and attribute operations on a directory tree. (4) The *Find* benchmark uses the Linux command `find` to search for a file in a large file system staring from /. (5) *MailSrvIO* simulates the file operations of a mail server using Filebench. In this workload, 48 threads perform a sequence of create-append-sync, read-append-sync, read and delete operations in the `/var/mail` directory. (6) *OLTP* is a benchmark from the Sysbench benchmark suite. It mimics the operations of a database server for a company owning a large number of warehouses. It contains transactions that implement order creation, order entry, order status, payment, and stock handling. (7/8) *Iscp* and *Oscp* copy a file from a remote system to the native system and vice-versa respectively using the Linux utility `scp`. In these benchmarks, a large amount of data stored in files is sent over secure network channels. This mimics the execution of back-end servers of popular platforms such as YouTube and Netflix.

Apache, *FileSrv*, *MailSrvIO*, and *OLTP* are multi-threaded benchmarks, and the remaining benchmarks are single threaded. For single threaded benchmarks, we simulate the execution of one instance of the application on each core of the system. We instrument the Linux kernel to track the context switches between applications. Next, we characterize each benchmark for a representative execution block of 1 billion instructions per core (akin to PIF [12]).

D. Instruction Mix in HyperTasks

We decompose (see Section III-A) the execution of a benchmark into HyperTasks belonging to five categories: application, system call handler, interrupt (top half) handler, bottom half handler, and scheduler. Figure 3 shows the contribution (in terms of instructions) of each HyperTask to the

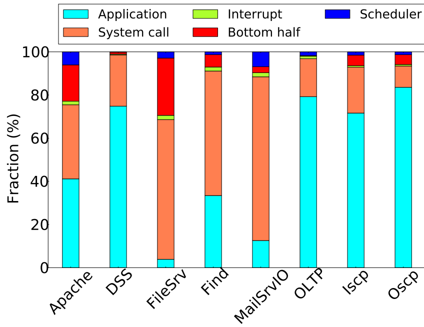

Fig. 3: Instruction mix in HyperTasks

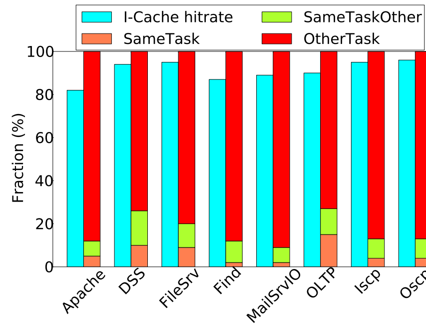

Fig. 4: I-cache hit rate and breakup of evictions

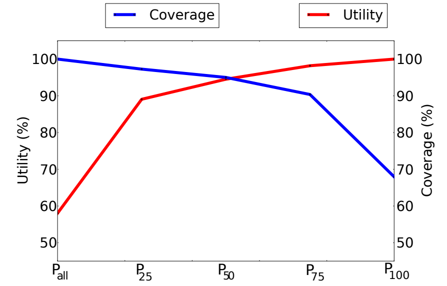

Fig. 5: Coverage and utility

overall execution of each benchmark. We shall use the terms *instructions* and *activity* in this section interchangeably.

As we observe from Figure 3, the OS activity for a benchmark varies from 20% (for *OLTP*) to 97% (for *FileSrv*). Let us now look at each benchmark individually.

Apache has roughly 40% application activity and 60% OS activity. In this case, most of the work done by the OS is in processing and validating network packets. We thus see an elevated amount of activity (18%) in the bottom half handlers. *DSS* has roughly 80% application activity because the application code spends a lot of time in servicing the large aggregate queries of the client. In comparison, the system call activity in *FileSrv* is significantly more, because of the file system's operations. Additionally, because of heavy interaction with the hard disks, a lot of interrupts need to be serviced; hence, interrupts and bottom half handlers account for roughly 27% of the benchmark's instructions. Another benchmark that has a lot of system call activity (65%) is *Find*. It has a high system call component because there are frequent calls to the OS for browsing through the file system. In comparison, *OLTP* services a lot of database requests, and thus the application instructions are more numerous given the amount of query processing that is done in modern databases. It has a moderate amount of OS activity (21%). *MailSrvIO* (like *FileSrv*) spends a lot of time in the OS mode(>93%). This is primarily because of the file operations for the mail server. Finally, *Iscp* and *Oscp* have around 75% of application activity, and 20% of system call activity. *Iscp* shows more system call activity because it has additional system calls to check if new packets have arrived. In all the benchmarks the contribution of the interrupt handler (top half) varies from 2-4%. Finally, for all these applications, we notice that the scheduler is invoked quite often; its contribution to the instruction mix lies between 2-5%.

The primary conclusion from this study is that the application HyperTasks, and the system call handlers form a major chunk of the executed instructions. Nonetheless, other HyperTasks such as the scheduler, the bottom half handler, and the interrupt handler also account for up to 25% of the total instructions.

E. I-cache Hit Rates/Evictions

Figure 4 shows the characterization of the i-cache accesses. For each benchmark, we show two bars. The first bar represents the i-cache hit rate for the benchmark. The hit rates for the benchmarks lie between 80% (*Apache*) to 95% (*FileSrv*). It must be noted that the i-cache hit rates are on the lower side as compared to benchmarks without OS activity (hit rates $\approx$ 98-99.9%). To understand the reasons for such low hit rates, let us look at the breakup of i-cache evictions. In Figure 4, the second bar for each benchmark shows the breakup of i-cache evictions into three categories: $SameTask$, $OtherTask$, and $SameTaskOther$.

$SameTask$ refers to the lines evicted by other lines of the same HyperTask, and $OtherTask$ refers to one HyperTask evicting lines populated by another HyperTask. $SameTask-Other$ is slightly more subtle. Let us assume that a HyperTask adds lines A, B, and C to the same set. Then there is a context switch, and another HyperTask brings in line D into that set. Then, we switch back to the first HyperTask again. Now, if it needs to immediately bring in another line say E, it will evict either line, A, B, or C. It will not evict D because it has a higher priority (as per the LRU replacement scheme). In this case, we have an eviction because of an intervening HyperTask that has changed the priorities within a set. To evaluate the effect of other HyperTasks, we are interested in the categories: $OtherTask$ and $SameTaskOther$. We can conclude from Figure 4 that most (80-90% of the time) of the lines in the i-cache are evicted because of other HyperTasks. The category, $OtherTask$, clearly dominates. However, $SameTaskOther$ can account for 5-10% of i-cache evictions in some benchmarks. The crucial insight that we obtain from this figure is that it is necessary to reduce the destructive interference between different HyperTasks, and intra-HyperTask evictions are not significant.

F. Prefetching HyperTasks

We define the *prefetch list* of a HyperTask as a set of cache lines that should be prefetched into the i-cache before the execution of the HyperTask begins. A HyperTask is invoked multiple times during the execution of a benchmark. Depending on the input parameters, and the state of global variables, the instructions executed by a HyperTask may vary. It is thus important to decide which cache lines should be added to the prefetch list, and which should be left out.

1) Coverage and Utility: A prefetch list should have two desirable characteristics: (a) coverage: it should cover most execution paths inside a HyperTask (b) utility: a cache line

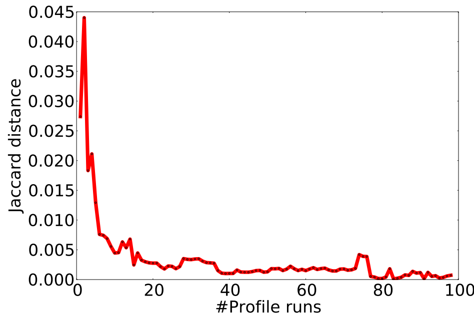

Fig. 6: Jaccard distance vs #runs

which is added to the prefetch list should be used during the execution of the HyperTask. We quantify these characteristics as:

$$coverage = \frac{\#accessed\ lines\ found\ in\ the\ prefetch\ list * 100}{\#lines\ accessed\ by\ the\ HyperTask}$$

$$utility = \frac{\#lines\ in\ the\ prefetch\ list\ that\ were\ accessed * 100}{\#total\ lines\ in\ the\ prefetch\ list}$$

Let us consider different methods of constructing a prefetch list. We evaluate the coverage and utility of five different prefetch lists: P_{all}, P_{25}, P_{50}, P_{75}, and P_{100}. For a HyperTask, prefetch list P_x is a set of cache lines that are accessed in at least $x\%$ of the HyperTask invocations. P_{all} is the set of cache lines that are accessed at least once during the invocation of a HyperTask. Note that $P_{100} \subseteq P_{75} \subseteq P_{50} \subseteq P_{25} \subseteq P_{all}$.

Figure 5 shows the average value of coverage and utility of these five prefetch lists averaged across all the benchmarks. We profile and test the efficacy of the prefetch lists on the same set of execution blocks. We observed a similar pattern across all the benchmarks. The coverage of the prefetch list varies from 100% for P_{all} to 68% for P_{100}. The utility of prefetch list varies from 100% for P_{100} to 58% for P_{all}. We want to have the best of both criteria: coverage and utility. Here, P_{50} seems to be the most balanced option. It has around 92% coverage and 91% utility. We pick this criteria for adding a cache line to the prefetch list.

2) Profiling Duration: Each HyperTask needs to be profiled multiple times, before its prefetch list is created. Each additional profiled invocation of a HyperTask may change its prefetch list. A HyperTask can be considered to be profiled, if the prefetch lists constructed after successive invocations do not show large variation. It is necessary to decide the ideal number of profiling runs for a HyperTask.

We calculate the variation between two prefetch lists as the Jaccard Distance between them.

$$JaccardDistance(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

A value of 0 indicates completely similar lists, while a value of 1 indicates completely dissimilar lists. Figure 6 shows the Jaccard Distance between successive invocations of HyperTasks averaged across all the benchmarks. *Jaccard Distance(n)* is defined as the Jaccard Distance between the prefetch lists created at the end of invocation n and invocation $n + 1$. We observe that the Jaccard Distance between successive invocations do not vary by more than 0.005 after 10 invocations. We use this observation to limit the number of profiling runs of a HyperTask to 10.

G. Detailed Characterization of HyperTasks

Table III shows the characterization results for each category of HyperTasks for all the benchmarks. For each HyperTask category, we first report the number of HyperTasks we observed in that category. Next, for each HyperTask category, we report the average number of instructions executed between two consecutive invocations of a HyperTask, the average number of instructions executed each time a HyperTask is invoked, the average size of the instruction footprint(in terms of cache lines), the average number of unique functions in the HyperTask, the average i-cache hit rate during the execution of the HyperTask, and the fraction of i-cache misses which were present in the prefetch list of the executing HyperTask.

The total number of HyperTasks for each benchmark is limited (<500 for most of the benchmarks). Additionally, we observe that the number of instructions between two successive invocations of the same HyperTask is high (100,000-200,000) mainly because HyperTask execution exhibits high temporal locality. The application and system call HyperTasks are always found in a pair, hence the number of such tasks is the same. Let us consider the application HyperTasks first. On one end of the spectrum is *MailSrvIO*, which executes around 1,000 instructions between two consecutive system calls, and on the other hand of the spectrum is *Oscp* which executes more than 15,000 instructions between two successive system calls. Although the instruction count for each benchmark is high, these instructions are mostly executed in loops, and they are clustered across a small set of i-cache lines. For most of the HyperTasks, the average number of i-cache lines accessed is less than 200. Considering that our i-cache has 512 lines (32 KB cache size for 64 byte blocks), capacity misses are not an important issue. However, as we observe, the i-cache hit rate during the execution of many HyperTasks is around 85% mainly due to conflict misses.

We make a conclusion similar to [13] that most misses are part of long and repeating sequences on the basis of the last column (*iFrac*), which shows that around 90% of the i-cache misses happen for cache lines that are a part of the prefetch list (generated using the P_{50} criteria). Note that for this experiment no prefetching is being done; the lists are just being constructed once at the beginning. Considering that the average i-cache miss rate during a HyperTask's execution is around 10-15% and the average fraction of cache misses found in the prefetch list is 80-90%, if we can successfully fetch the lines belonging to the prefetch list before a task starts execution, the i-cache's miss rate should reduce by at least 5-10%, leading to a significant improvement in overall performance.

The prefetch list for a HyperTask is created from the body of frequently accessed functions. We observe that the average number of functions accessed during the HyperTask execution is very small. It ranges from 13 functions for the top half interrupt handlers of *Apache* to 55 functions for system call handlers of *Find*. These functions are called repeatedly using loops (also see Table III). Note that after applying the P_{50}

Type	#Hyper Tasks	#Insts between invocations	#Insts	#Lines	#Funcs	i-cache hit rate	iFrac (%)
		Per HyperTask					
Apache							
app	464	173975	1519	56	21	84	80
sys	464	179109	1242	78	24	81	87
bhalf	5	251813	12819	136	34	81	44
thalf	9	190192	454	39	13	67	96
sched	1	27742	1775	111	31	83	78
FileSrv							
app	172	980935	1154	49	15	83	77
sys	172	718406	14039	79	26	96	57
bhalf	6	271253	23339	99	29	96	71
thalf	11	285403	809	51	18	81	95
sched	1	61672	1472	112	30	80	81
OLTP							
app	211	619628	4850	224	59	92	62
sys	211	617311	1014	78	26	85	90
bhalf	5	960944	1081	35	10	95	90
thalf	9	920550	1990	67	20	92	90
sched	1	65352	1141	103	29	76	95
Iscp							
app	196	635961	8680	57	18	98	82
sys	196	629592	2402	99	30	89	80
bhalf	7	368556	5827	130	36	86	67
thalf	8	375618	818	79	26	77	93
sched	1	112470	1475	113	31	72	78

Type	#Hyper Tasks	#Insts between invocations	#Inst	#Lines	#Funcs	i-cache hit rate	iFrac (%)
		Per HyperTask					
DSS							
app	41	105258	3895	60	18	95	80
sys	41	110689	1308	139	44	92	87
bhalf	3	816911	2868	92	28	77	44
thalf	3	800198	1358	125	38	68	96
sched	1	437482	674	65	20	78	78
Find							
app	40	207670	2798	52	17	93	77
sys	40	181112	4671	171	55	86	57
bhalf	3	148911	2974	105	32	75	71
thalf	3	122682	1176	98	32	64	95
sched	1	60345	786	73	22	74	81
MailSrvIO							
app	182	430686	1086	52	16	82	62
sys	182	279713	4561	82	28	91	90
bhalf	4	486127	3846	138	39	84	90
thalf	10	117108	301	43	15	78	90
sched	1	13955	1005	94	26	82	95
Oscp							
app	90	597384	15448	45	15	98	82
sys	90	600707	1568	100	29	82	80
bhalf	5	368719	6447	142	37	84	67
thalf	6	370267	719	67	23	75	93
sched	1	99243	1048	96	26	74	78

app → application, sys → system call handler, bhalf → bottom half handler, thalf → top half handler, sched → scheduler
iFrac → fraction of i-cache misses found in the prefetch list of the executing HyperTask

TABLE III: HyperTasks: detailed characterization

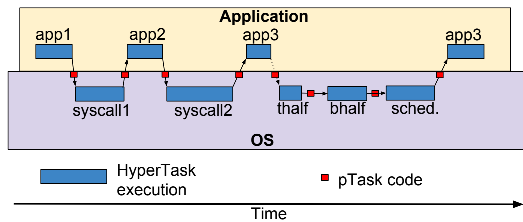

Fig. 7: $pTask$: Execution overview

criteria, the number of functions that will be a part of the prefetch will be lower (20-30% reduction in our experiments).

IV. Prefetching HyperTasks

The crucial problems that need to be solved in any prefetching algorithm are: (1) What to prefetch? and (2) When to prefetch? Let us consider the first problem. We need to decide whether we need to fetch instructions at the granularity of a single instruction, or at coarser granularities such as at the level of basic blocks, functions, or groups of functions. We observed experimentally that the best strategy for our set of benchmarks is to prefetch a set of functions that are deemed to be *frequent* in a HyperTask. Next, we observed that the best time to prefetch these functions is once, right before executing the instructions in the HyperTask. We shall justify these choices in Section V.

Figure 7 shows an overview of the $pTask$ strategy. The small blocks correspond to *pTask's* software routines, which reside in the kernel's address space. These routines additionally have access to the hardware registers that we add and the user process's address space as well. The central components of $pTask$ are these OS routines that are called at the start of each HyperTask (Table I shows the trigger event for each HyperTask). A HyperTask goes through two disjoint modes of execution: profiling mode, and normal mode. Any new HyperTask starts execution in the profiling mode of execution. In the profiling mode, the execution of the HyperTask is recorded. We use the recorded information to create a prefetch list. Once a HyperTask has been adequately profiled, we run it in the normal mode. In this mode once the HyperTask is invoked, we first read its prefetch list and then start executing it. Unlike the profiling mode, the normal mode is very non-intrusive and is not associated with timing overheads. Figure 8 summarizes the data structures used by $pTask$ with the help of an example scenario, where we have 3 modules with 16 functions. Please refer to this figure as we introduce *pTask's* data structures in the rest of this section.

First, let us discuss the profiling mode of execution. There are several problems that need to be solved. We need to first define a mechanism to identify a function that is frequently executed. Second, we need to be able to identify HyperTasks in executing programs (including the OS), track the behavior of their functions, and create a data structure that can save the frequency of executed functions.

A. Profiling: Recording Function Execution

We profile multiple runs of a HyperTask before creating its prefetch list. However, before that, we must record the set of functions executed during a single run of the HyperTask. We represent the set of functions as a bit vector called the FuncVector (saved in software). The FuncVector contains one bit for each function inside the application/OS code. It is a software structure in the process or kernel's address space, and there is one such vector for each core.

During the profiling mode of execution, all the bits of the FuncVector are first set to false. When a function with index x is executed, the bit numbered x in the FuncVector is set to true. This is achieved using a special assembly instruction – *recordFunc x* . Now, let us discuss the process of assigning an index x to each function.

1) OS Kernel: Let us first consider the OS kernel. We shall explain all our mechanisms with respect to the Linux kernel in this paper. Note that our methods are not specific to any particular type of kernel. Now, the Linux kernel consists of

mostly statically linked code; however, it does support modules that are chunks of dynamically linked code. The compiler adds the $recordFunc\ x$ assembly statement at the start of each function. Here, the value of x is undefined. Once all functions are compiled, the linker iterates over the functions, and assigns the function id's sequentially from 0 to n-1; n being the number of functions inside the kernel.

We evaluated our approach for the Linux kernel (version 2.6.32). It contains around 20k functions (as obtained from the System.map file), 1 bit for each function leads to a FuncVector of around 2.5 KB.

Now, let us create a generic mechanism for dynamically loaded modules. Each module is compiled separately; hence in each module, we assign ids to functions in a monotonically increasing sequence starting from 0. Let the starting n bits in the FuncVector refer to kernel functions, and let the remaining bits refer to functions in modules. Subsequently, when a module is loaded, we assume that its functions have been appended to the kernel's functions. For example, let the kernel have 10,000 functions. Subsequently, we load a module that has 100 functions. We assume that the functions of the module have ids from 10,001 to 10,100. To implement this we need to map the code in each module to an offset in the FuncVector.

This is achieved by maintaining a separate table (*module map*) that maps a module (identified by the path and version) to its offset in the FuncVector. At the time of linking a module (at run time), we need to create space for the module's functions inside the FuncVector. We achieve this by invoking a dedicated software routine, $ModuleManager$, which manages (allocate/lookup/deallocate) the *module map*. If a module is not mapped to locations in the FuncVector, the ModuleManager assigns entries in the FuncVector to the module. At the time of invoking a function in a module, we get the offset from the ModuleManager, and subsequently save it in a special register called FuncVectorBase. The $recordFunc\ x$ instruction sets the $(x+FuncVectorBase)^{th}$ bit inside the FuncVector. When a module is unloaded, the corresponding locations in the FuncVector are freed and are available to be mapped by other modules.

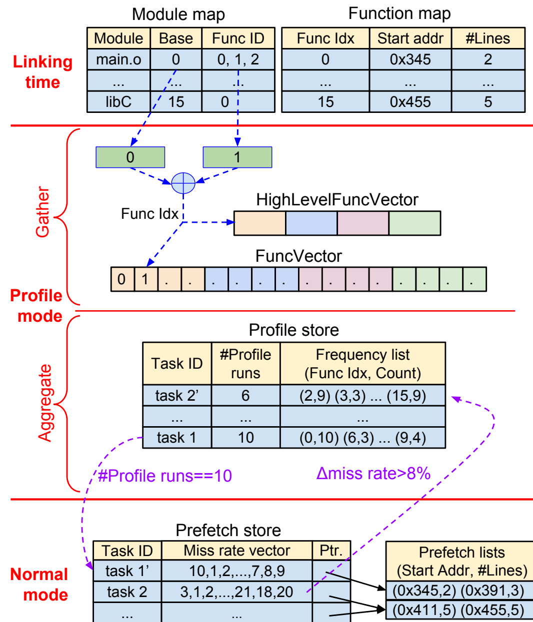

Fig. 8: Data structures used by $pTask$

We maintain the number of unique functions accessed during the HyperTask execution in a register FuncCounter. This register is initialized to 0 at the beginning of a HyperTask, and whenever the recordFunc instruction changes the value of a function's bit from 0 to 1, the value of FuncCounter is incremented by 1. When its value reaches K (typically 125), we unset the profile mode bit of the core. This ensures that the HyperTask size is not unreasonably big, and profiling overheads are low. This is also done for application HyperTasks.

2) Application: We follow the same process for an application. We can assign unique ids to each function at link time, and treat dynamically linked code as modules. Additionally, we treat sections of code that are self modifying as dynamically linked libraries (DLL). Each function within a DLL has a function id that is added to the FuncVectorBase register. When the DLL changes, we need to flush its entries from the *module map* and recreate it. This is done by a custom software routine similar to the ModuleManager.

B. Hypertask Annotation

We now have a method of uniquely identifying functions in an application and in the kernel. Our approach is to fetch a set of functions in a HyperTask that satisfy certain coverage criteria. For this purpose it is necessary to identify HyperTasks at runtime.

1) Identifying Application Hypertasks: We define an application HyperTask as a sequence of application functions that are executed between two consecutive system calls. If there is an asynchronous event such as an interrupt/exception in the middle of an application, we do not terminate the application HyperTask. Instead we start the interrupt HyperTask, and after it completes, we resume the application HyperTask.

As explained in Section IV-A2, each application function is assigned a unique id, which is hard-coded inside its recordFunc instruction. We use the recordFunc instruction to modify two 256 bit registers: PathVector, and PathSum. PathVector encodes the set of functions accessed during the HyperTask's execution, and PathSum encodes the order in which functions are accessed during the HyperTask's execution.

Both, PathVector and PathSum are initialized to all 0s at the beginning of an application HyperTask. When the $recordFunc\ n$ instruction executes, we set the $n \bmod 256^{th}$ bit in the PathVector. Whenever the value of the ith bit in the PathVector is changed from 0 to 1, we increment PathSum by i, and also perform a left rotation operation for i positions. A *left rotation* is defined as a left shift operation, where the bit shifted out of the most significant position is inserted in to the least significant position. Note that the operation of updating PathSum is a non-commutative operation, and is thus ideally suited for encoding the order of function invocations. The tuple$< PathVector, PathSum >$ ensures that the execution of a HyperTask is more or less uniquely encoded. There is

a vanishingly small probability that two disparate tasks will have a matching pair of PathSum and PathVector.

The identifier of an application HyperTask is essentially an encoding of its control flow. This is constructed at the end of the HyperTask's execution. We use this identifier to predict the instructions executed in the next application HyperTask.

2) Identifying OS HyperTasks: We define four types of OS HyperTasks: (1) interrupt handler, (2) bottom half handler, (3) scheduler, and (4) system call handler. Identifying the id of the top half of the interrupt handler is simple. We just use the id of the interrupt. For the bottom half handler, we use the program counter of the first instruction of the function that encapsulates the bottom half handler. For the scheduler, we embed an instruction in the beginning of the scheduler code that lets the hardware know that the scheduler is running. If the scheduler invokes some other kernel function that is not defined as one of our HyperTasks, then also this function is counted as a part of the scheduler's HyperTask. When the scheduler returns from a context switch, the previous OS HyperTask resumes execution.

In the case of a system call, this definition fails to hold because the same system call can have very different behaviors depending on the arguments. For example, the *read* system call in Linux has a file descriptor as its argument; the descriptor may refer to a file or a network connection or some other device. It is not possible to find the type of the argument before the system call invocation. To solve this problem, we need to look at the code of the application prior to executing the system call and try to correlate it with the behavior of the system call. We thus uniquely identify a system call HyperTask by the id of the previous application HyperTask that was executing. This is a generic method and is agnostic to the OS.

C. Prefetch List

We have up till now discussed methods of annotating functions (user and OS) and HyperTasks. Let us now propose a method to maintain a count of the number of times a function executes. Let us define a new structure called a *profile store*. For each HyperTask, the profile store maintains the list of functions in it, and the number of times that they have executed. We keep 32 bits for the function id, and 4 bits for the count. Additionally, it also maintains a count that depicts the number of times the HyperTask has been profiled.

When a HyperTask completes execution in the profiling mode, we need to update the counts of functions in the profile store. We iterate through the FuncVector, and for each bit that is set, we increment the count of the function in the profile store's entry for the HyperTask. We optimize this process by maintaining a high-level map of the FuncVector, we call it *HighLevelFuncVector*. 1 bit of HighLevelFuncVector represents 512 bits (one cache line) of the FuncVector. If the bit is 0, then it means that none of the bits in the 512 bit set are set to 1. This takes care of sparse accesses in large codes such as the kernel, which is most often the case. This optimization helps us iterate through FuncVector much faster.

Let the number of times a given HyperTask has been profiled be N, and let the count of function f for this HyperTask be N_f. Only those functions for which $N_f/N > 0.5$ satisfy the P_{50} criterion of prefetching (see Section III-F1). The cache lines containing the instructions of the frequently accessed functions are obtained using the *function map*, which is a data structure that stores the starting address of a function and the number of subsequent cache lines that store the instructions in a function. These lines are added to the prefetch list of the HyperTask. As described in Section III-F1, the prefetch list is generated after 10 profiling runs of a HyperTask.

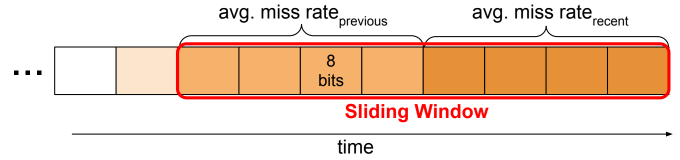

Fig. 9: Miss Rate Vector (conceptual view)

The prefetch list created at the end of the first profiling phase works well for most of the HyperTasks. However, the instruction footprint of some HyperTasks changes at run-time; such HyperTasks need re-profiling. We propose a scheme to tackle this problem at run time. For each HyperTask, we maintain the i-cache miss rates (quantized to 8 bits) for the last W invocations in a sliding window called the *miss rate vector* (see Figure 9). This window is maintained in software. Let the elements be number $M_1 \dots M_W$, where M_W is the latest. The criterion for re-profiling is:

$$\frac{\sum_{i=l+1}^{W} M_i}{W-l} - \frac{\sum_{i=1}^{l} M_i}{l} > \phi \quad (1)$$

The idea is that the average miss rate for the last $W - l$ invocations should be substantially higher than that of the average miss rate of the earlier l invocations. Empirically, we found the best combination to be: $W = 16$, $l = 8$, and $\phi = 0.08 (8\%)$.

We apply two techniques to reduce the size of prefetch lists. The first technique called *encode* uses run-length encoding. Instead of storing all addresses, it stores blocks of addresses. This technique reduces the size of the prefetch lists by around 300%. Next, we apply the *unionEncode* technique to combine similar prefetch lists. If the Jaccard distance between two prefetch lists, P_a and P_b is less than 0.05, instead of storing both lists, we store a union of both the lists, $P_a \cup P_b$. This technique reduces the number of prefetch lists by around 100%.

We now define a new data structure called the prefetch store, which maintains a pointer to a prefetch list and the miss rate vector for each HyperTask. It is possible for multiple HyperTasks to point to the same prefetch list because of the *unionEncode* compression technique that we use.

We save all our data structures such as the prefetch store, profile store, and FuncVector in the kernel's address space. This data is stored separately for each user process and the kernel. All the $pTask$ routines run as simple function calls during the execution of the kernel. Even if an application HyperTask's profiling terminates after encountering 125 unique functions, we still wait for the next system call to process its FuncVector (in kernel mode). This design choice eliminates security issues.

Prefetching technique	Hardware Support
markov [22]	Table of 1024 entries per core
CGP [3]	Used h/w support for online profiling. Call graph history cache of 1024 entries (one entry contains 8 callee slots) per core
PIF [12]	4 Stream address buffers per core. History buffer of 32k entries per core. Index table of 8K entries per core
RDIP [17]	Miss-table of 4k entries per core. 4-entry return address stack
pTask	4 registers per core

TABLE IV: Hardware support: Prefetching Techniques

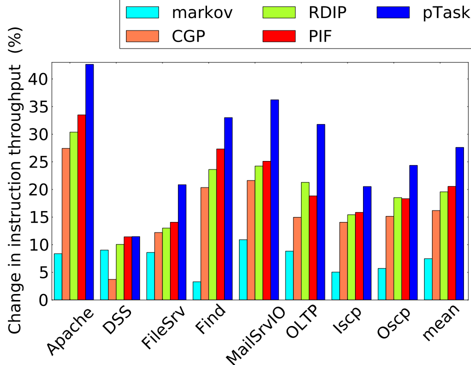

Fig. 10: Performance impact

D. Normal Mode

In the normal mode of execution, we first map the HyperTask identifier to its prefetch list. Each entry in the prefetch list contains the addresses of multiple i-cache lines. For each such line, we issue a non-blocking i-cache read operation. Only after the prefetches are issued, the HyperTask starts execution. The prefetches may slowdown the execution in the beginning, but as we show in Section V, the prefetch operations compensate for this delay during the later part of the HyperTask's execution.

In the normal mode of execution, we do not modify any profiling data structures. The processor uses the current privilege level bits to distinguish between the OS and user process. For the OS, the processor commutes the $recordFunc$ instruction to a nop. For the user process, the processor does not modify the FuncVector, but it still modifies the PathVector and PathSum registers. Notice that these registers are essential for identifying application and system call HyperTasks. Hence, they need to be updated in profiling mode, as well as in the normal mode of execution.

Finally, note that the hardware component of $pTask$ includes 4 additional registers for each core – FuncVectorBase, FuncCounter, PathSum, and PathVector – along with some logic for updating these registers.

V. Results

The primary motivation of this work is to improve the performance of a server processor, which is possibly running multiple server class applications simultaneously. We first analyze the impact of *pTask* for each benchmark individually and then discuss its impact on multi-programmed workloads in Section V-H.

A. Comparison: Performance Improvement

We compare the performance benefits gained with five instruction prefetching techniques: *markov* [22], *CGP* [3],

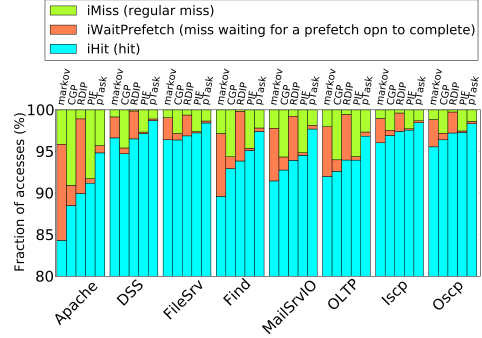

Fig. 11: Breakup: i-cache accesses

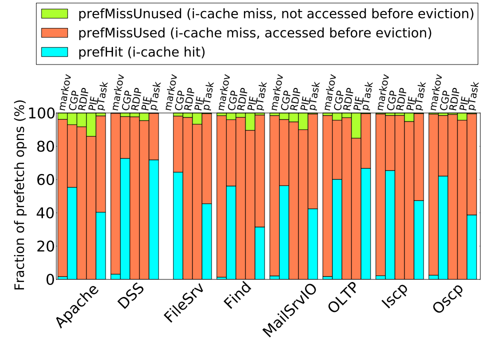

Fig. 12: Breakup: prefetch operations

PIF [12], *RDIP* [17] and *pTask*(proposed). Table II shows the details of our simulated system and Table IV shows the configurations for each prefetching technique. We simulate the *pTask* technique by feeding instructions corresponding to the profiling and prefetching stages to the simulation engine. The addresses for the software data structures maintained by the *pTask* technique are mapped to unmapped regions in the kernel's address space. Recall that the entire simulation setup has been discussed in Section III.

Figure 10 shows the performance benefits for all instruction prefetching techniques as compared to a baseline system with no prefetching. We compare the instruction throughput (#insts/cycle) (as proposed by [19]) of these techniques for an execution block of 1 billion instructions per core (same as PIF [12]). The mean performance benefits of these schemes are: *markov*(6.27%), *CGP*(15.86%), *RDIP*(19.78%), *PIF*(20.51%) and *pTask*(27.38%). The *pTask* technique outperforms the state of the art prefetch techniques, *PIF* and *RDIP*, by ≈7% (geom. mean). Note that our simulations consider all overheads associated with profiling and prefetch operations. We discuss these overheads in detail in Section V-B.

Figure 11 and Figure 12 show breakups of the i-cache accesses, and the i-cache prefetch operations respectively. Let us analyze these results to understand the gains in performance. The low performance of the *markov* scheme can be attributed to the high fraction of *iWaitPrefetch* events (miss waiting for prefetch to finish). The reason for a high fraction of *iWaitPrefetch* events is that the delay between the prefetch operation and the i-cache line access is lesser than the prefetch latency.

CGP records the sequence of function calls and issues prefetch operations for the next function while the current

Benchmark	Fraction of HyperTask invocations (exec. in profiling mode) (%)	Fraction of HyperTasks that are re-profiled (at least once) (%)	Slowdown in profiling run(%), Time in profiling mode(%)	Fraction of prefetch insts (%)
Apache	0.56	1.79	1.31 , 0.57	3.2
DSS	0.21	0.01	2.99 , 0.22	1.65
FileSrv	1.82	2.93	1.02 , 1.84	0.77
Find	0.26	3.88	1.59 , 0.27	1.23
MailSrvIO	0.51	12.73	1.27 , 0.53	2.61
OLTP	0.96	0.25	1.53 , 0.97	1.81
Iscp	1.18	0.24	0.92 , 1.19	0.75
Oscp	0.85	0.45	1.01 , 0.86	0.55

TABLE V: Details: profiling and prefetching

Benchmark	naive	encode	unionEncode
Apache	45.98	15.16	6.89
DSS	5.61	1.85	0.68
FileSrv	26.14	8.27	4.12
Find	59.39	19.31	6.73
MailSrvIO	32.85	10.2	4.47
OLTP	163.59	45.71	13.85
Iscp	44.51	13.86	7.74
Oscp	23.94	7.43	5.04

TABLE VI: Size of the prefetch store (KB)

function is being executed. We notice two shortcomings of the *CGP* scheme: (1) The context of the current function is not used to predict the next function Hence, its predictor performs poorly for the OS codes. This explains the high fraction of *iMiss* events (regular misses) in Figure 11. (2) If two functions a and b are called in succession in a loop, *CGP* issues a prefetch operation for function b, each time function a is called. After the first iteration, additional prefetches are typically not necessary. Hence, many prefetch operations of *CGP* are unwanted, which explains the high fraction of *prefHit* events (prefetched lines already in the cache) for the *CGP* scheme in Figure 12.

RDIP uses the last 4 functions in the call stack to predict the next function. Since the predictor of *RDIP* is context sensitive, its accuracy is better than that of *CGP*. This is reflected in its lower fraction of *iMiss* events. However, the time between two functions is not enough to hide the prefetch latency; hence, the fraction of *iWaitPrefetch* events is high for *RDIP*.

Compared to *RDIP*, the prefetch operations of *PIF* are more timely. This is because the *PIF* scheme issues prefetch operations for a spatial region. However, *PIF* suffers from a poor prediction accuracy; hence, its fraction of *prefMissUnused* events (prefetched line is not used before eviction) is high. Incorrect prefetches waste i-cache bandwidth, and can potentially pollute the i-cache.

RDIP and *PIF* use special hardware (line buffer) to filter the prefetch operations for lines that are already present in the i-cache. Hence, the fraction of *prefHit* events is almost zero for both these schemes. As we shall see in Section V-B, the number of prefetch operations is low for the *pTask* technique (<3% for most of the benchmarks). Hence, we do not see any appreciable benefits of adding such a hardware unit for $pTask$.

As discussed in Section III-F, the prefetch list of *pTask* has a high coverage and utility. Hence, the fraction of *prefMissUnused* events is low for *pTask*. Additionally, the prefetch operations of *pTask* are more timely and accurate. Hence, its i-cache hit rates are higher than those of competing strategies, *RDIP* and *PIF*, by 2-6% (see Figure 11). This is the primary reason for the superior performance of the *pTask* technique. All benchmarks follow similar trends except *DSS* (high baseline i-cache hit rate + almost no reprofiling).

B. Overheads: Profiling and Prefetching

Table V shows the details of the profiling and the prefetching operations of *pTask*.

Profiling: The second column shows the fraction of HyperTask invocations that are executed in the profiling mode. The reason for such low numbers($<2\%$) is basically because most HyperTasks have this flow of control: accept request, service it, and return to the baseline state. The code for servicing a request remains mostly similar. Hence, the need for profiling and re-profiling HyperTasks does not arise very frequently.

The third column of Table V shows the fraction of HyperTasks that are re-profiled (at least once). For most of the benchmarks, less than 4% of the HyperTasks are re-profiled. Once a HyperTask is re-profiled, its performance (as compared to no re-profiling) improves by 8-10$\%$. This justifies the additional profiling runs.

The first entry in the fourth column shows the mean slowdown during profiling runs (as compared to a baseline system without prefetching). The slowdown during the profiling mode is due to the execution of additional instructions for updating the FuncVector and the profile store. The second entry in the fourth column shows the fraction of total time spent in profiling mode. This is directly proportional to the fraction of invocations that are executed in the profiling mode and the slowdown observed during each profiling run. We can conclude that benchmarks spend a very small amount of time ($<2\%$) in the profiling mode and hence the gross overheads of the profiling runs are inconsequential.

Prefetching: The last column in Table V shows the fraction of prefetch instructions. The fraction of prefetch instructions is low (0.5-3.2%) because of two reasons: (1) only one prefetch instruction is executed for each block of i-cache lines (because of our compression techniques), and (2) during one invocation of a HyperTask, many instructions execute multiple times (loops). However, the corresponding prefetch instructions are executed only once – at the beginning of the HyperTask.

Figure 13 shows the reduction in the d-cache hit rate due to the additional overhead of saving all our data structures in the kernel's address space. We compare two cases: (1) $pTask$, and (2) a system where all the $pTask$ data structures are saved in a separate hypothetical storage area. Since we use a highly compressed prefetch store, the reduction in the d-cache hit rate is low (0.5-2.5%). Previous works [4], [7] have observed that OS intensive benchmarks are not very sensitive to the performance of the d-cache. So even a 2% reduction in the d-cache hit rate does not affect the performance of the benchmarks significantly, and we still get speedups. Note that an OOO pipeline is very effective in mitigating the slowdowns due to d-cache misses and additional non-blocking prefetch instructions.

C. Prefetch store

Table VI shows the size of the prefetch store (in KB) for three configurations: naive (no compression), *encode*, and *unionEncode* (see Section IV-C). The compression ratios are 5X-12X. To put these numbers into perspective, with an 8X compression ratio, the *unionEncode* technique uses around 3 bits to store the prefetch entry of one i-cache line.

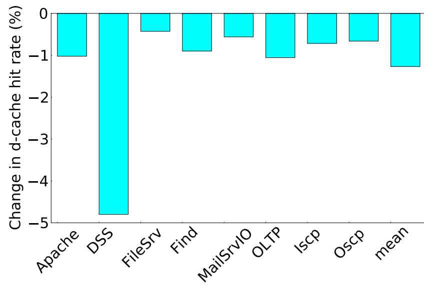

Fig. 13: Impact of profiling and prefetching on the hit rate of the d-cache

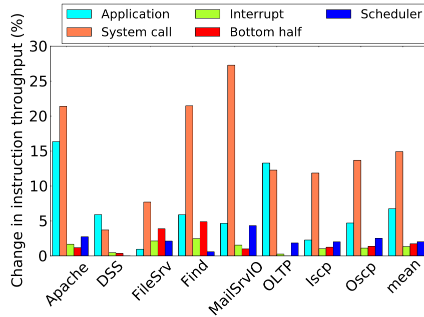

Fig. 14: Breakup: performance improvement

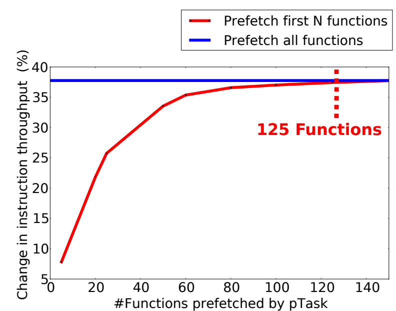

Fig. 15: Impact of prefetching first N unique functions of a HyperTask

D. pTask: Performance Breakup

Figure 14 shows the overall speedup of our benchmarks only when a specific type of HyperTasks are prefetched (using $pTask$). The speedups are in the range of 8-27% when we prefetch either only applications or system calls. However, the net speedup is much lower (2-4%) when we prefetch top/bottom half handlers or the scheduler. The reason for this is that application and system call HyperTasks account for a bigger portion of the overall execution(also see Figure 3). Note that all of these individual speedups are (and should be) less than the overall speedup when we prefetch all types of HyperTasks.

E. Granularity: Basic block vs Function

Figure 16 shows the performance gains of the *pTask* technique when we profile and prefetch the HyperTasks at the granularity of a function versus a basic block (figure normalized to $pTask$ with functions). For the basic block strategy, we still identify the application HyperTasks using the *recordFunc* instruction. We use another assembly instruction *recordBasicBlock* to record the execution of a basic block. Considering that an average basic block has around 5-6 assembly instructions, *recordBasicBlock* instructions constitute around 15-20% of all the executed instructions. Although the processor treats these instructions as nops during the normal mode of execution, they consume fetch bandwidth. Hence, even though the implementation with basic blocks is more fine-grained, it performs at least 4-7% slower than a *pTask* system running at the granularity of functions. Hence, we decided to use $pTask$ at the granularity of functions.

F. HyperTask Size

As discussed in Section IV-A, we only consider a limited portion of the HyperTask's execution that follows an OS event. Figure 15 shows the impact of prefetching the first K unique functions (not function calls) of a HyperTask for the *Apache* benchmark (we observe similar results for other benchmarks). The performance of the HyperTask increases as we increase the number of profiled functions. However, after prefetching around 125 functions, the benefit of prefetching additional functions is close to zero. Hence, we stop the profiling of a HyperTask after it executes 125 functions. We observe that more than 99% of the HyperTasks access less than 125 unique functions. So our choice of 125 functions covers most of the benchmarks' execution.

G. pTask: Using Extra Hardware

We see two opportunities to improve the performance of the *pTask* technique with extra hardware support.

Hardware Prefetch Store: The prefetch store is a software structure and can cause d-cache pollution. So, we use additional hardware structures to completely eliminate the d-cache accesses for the prefetch store. We propose to add two units on each core. First is a scratch pad memory of 8 KB. This stores the prefetch lists of the frequent HyperTasks, and the other structure is used to map the HyperTask ID to its prefetch list in the scratch pad memory. Figure 17 shows the performance gains achieved by this technique, *hwPrefStore*, versus *pTask*. The mean performance benefits of *pTask* and *hwPrefStore* are 27.38% and 28.57% respectively.

Oracle pTask: An astute reader can observe that the time between the prefetch operation and the first function of the HyperTask is not enough to bring its i-cache lines from the lower level cache to the i-cache. To avoid the miss penalty, we need at least two more structures: (i) a structure to predict the next HyperTask, and (ii) a structure to start fetching the i-cache lines of the next HyperTask before the current HyperTask completes execution.

Instead of implementing these structures, we evaluate the upper bound on the performance benefits that can be expected out of these schemes. We perform experiments with *oracle_pTask*. At the start of a HyperTask's execution, *oracle_pTask* instantaneously reads all the lines in its prefetch list into the i-cache without incurring any overhead. Figure 17 shows the performance comparison of the oracle prefetcher, *oracle_pTask*, with *pTask*. While *pTask* gives a mean performance improvement of 27.38%, the oracle prefetcher gives a performance improvement of around 31.09%. *pTask* is thus close to optimal.

H. Multi-programmed Workloads

Next, we show the impact of all prefetching techniques on the execution of multi-programmed workloads. Table VII shows the constituent benchmarks of each randomly chosen multi-programmed workload, and Figure 18 shows the performance benefit for each prefetching technique. We simulate a

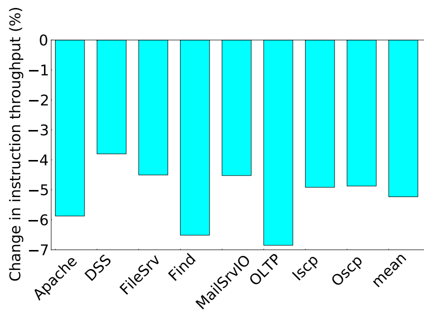

Fig. 16: *pTask* granularity: basic-block vs function

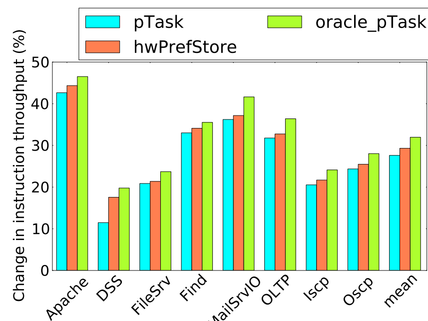

Fig. 17: Performance Impact: h/w support

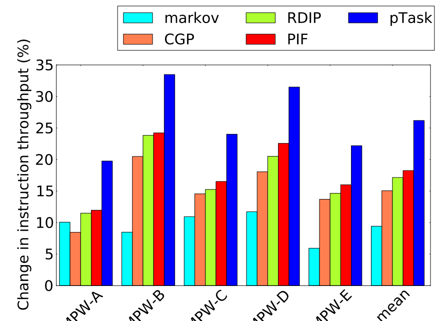

Fig. 18: Performance Impact: Multi-programmed Workloads

Workload ID	Constituent benchmarks
MPW-A	DSS, FileSrv
MPW-B	Apache, OLTP
MPW-C	Apache, DSS, FileSrv, Iscp
MPW-D	Apache, DSS, Find, OLTP
MPW-E	Iscp, Find, FileSrv, Oscp

TABLE VII: Multi-programmed Workloads

16-core system (see Table II) and we allocate an equal number of cores for each benchmark. The mean performance benefits of these schemes are: *Markov* (9.42%), *CGP* (15.04%), *RDIP* (20.66%), *PIF* (21.76%) and *pTask* (28.70%). A closer observation of these numbers shows that the performance of a multi-programmed workload is very strongly correlated with the performance of each constituent benchmark (also see Section V-A). Note that we simulate simultaneous execution of profiling modes (across cores), and simultaneous accesses by different processes to prefetch/profile stores. The reported numbers include all of these overheads.

VI. Conclusion

In this paper, we proposed $pTask$, as an alternative to state of the art prefetching techniques. The basic insight that we use is that it is not necessary to have prefetching activated all the time. There are portions of an execution where the i-cache hit rates are high, and thus do not stand to benefit from prefetching. In fact we can have a reverse effect of unnecessarily polluting our prefetching structures if we prefetch all the time. Instead, it is a better idea to use prefetching when it is needed. We show that during task switches, prefetching is needed because the i-cache miss rate peaks, and prefetching is in fact beneficial. For a popular suite of 8 OS intensive benchmarks, we demonstrate an average increase in instruction throughput of 7% over highly optimized state of the art proposals. In 4 benchmarks the increase is between 10-14%.

References

[1] "Filebench," http://filebench.sourceforge.net/wiki/index.php/Main_Page.
[2] "Tpc-h," http://www.tpc.org/tpch/.
[3] M. Annavaram, J. M. Patel, and E. S. Davidson, "Call graph prefetching for database applications," *ACM TACO*, 2003.
[4] I. Atta, P. Tozun, A. Ailamaki, and A. Moshovos, "Slicc: Self-assembly of instruction cache collectives for oltp workloads," in *MICRO*, 2012.
[5] A. Belay, D. Wentzlaff, and A. Agarwal, "Vote the os off your core," *MIT Technical Report MIT-CSAIL-TR-2011-035*.
[6] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX ATC*, 2005.
[7] R. Bhalla, P. Kallurkar, N. Gupta, and S. R. Sarangi, "Trikon: A hypervisor aware manycore processor," in *HiPC*, 2014.
[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *PACT*, 2008.
[9] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai *et al.*, "Corey: An operating system for many cores." in *OSDI*, 2008.
[10] K. Chakraborty, P. M. Wells, and G. S. Sohi, "Computation spreading: employing hardware migration to specialize cmp cores on-the-fly," in *ACM SIGARCH*, 2006.
[11] S. Chandran, P. Kallurkar, P. Gupta, and S. Sarangi, "Architectural support for handling jitter in shared memory based parallel applications," *IEEE TPDS*, 2014.
[12] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *MICRO*, 2011.
[13] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *MICRO*, 2008.
[14] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH*, 2006.
[15] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *ACM SIGARCH*, 1997.
[16] C. Kaynak, B. Grot, and B. Falsafi, "Shift: Shared history instruction fetch for lean-core server processors," in *MICRO*, 2013.
[17] A. Kolli, A. Saidi, and T. F. Wenisch, "Rdip: return-address-stack directed instruction prefetching," in *MICRO*, 2013.
[18] A. Kopytov, "Sysbench: a system performance benchmark," http://sysbench.sourceforge.net, 2004.
[19] K. M. Lepak, H. W. Cain, and M. H. Lipasti, "Redeeming ipc as a performance metric for multithreaded programs," in *PACT*, 2003.
[20] C.-K. Luk and T. C. Mowry, "Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors," in *MICRO*, 1998.
[21] D. Nellans, R. Balasubramonian, and E. Brunvand, "Interference aware cache designs for operating system execution," *University of Utah, Technical Report UUCS-09-002*, 2009.
[22] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *MICRO*, 2005.
[23] F. Petrini, D. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q," in *ICS*, 2003.
[24] S. R. Sarangi, R. Kalayappan, P. Kallurkar, and S. Goel, "Tejas simulator : Validation against hardware," *CoRR*, vol. abs/1501.07420, 2015. [Online]. Available: http://arxiv.org/abs/1501.07420
[25] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, "Tejas: A java based versatile micro-architectural simulator,," in *PATMOS*, 2015.
[26] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *OSDI*, 2010.
[27] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak, "Branch history guided instruction prefetching," in *HPCA*, 2001.
[28] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, "Guided region prefetching: a cooperative hardware/software approach," in *ACM SIGARCH*, 2003.
[29] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *ACM Special Interest Group on Operating Systems*, 2009.
[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ACM SIGARCH*, 1995.