Protean Code: Achieving Near-free Online Code Transformations for Warehouse Scale Computers

Michael A. Laurenzano, Yunqi Zhang, Lingjia Tang, Jason Mars
Advanced Computer Architecture Laboratory, University of Michigan - Ann Arbor, MI
{mlaurenz, yunqi, lingjia, profmars}@eecs.umich.edu

***Abstract*—Rampant dynamism due to load fluctuations, co-runner changes, and varying levels of interference poses a threat to application quality of service (QoS) and has limited our ability to allow co-locations in modern warehouse scale computers (WSCs). Instruction set features such as the non-temporal memory access hints found in modern ISAs (both ARM and x86) may be useful in mitigating these effects. However, despite the challenge of this dynamism and the availability of an instruction set mechanism that might help address the problem, a key capability missing in the system software stack in modern WSCs is the ability to dynamically transform (and re-transform) the executing application code to apply these instruction set features when necessary.**

In this work we introduce *protean code*, a novel approach for enacting arbitrary compiler transformations at runtime for native programs running on commodity hardware with negligible (<1%) overhead. The fundamental insight behind the underlying mechanism of protean code is that, instead of maintaining full control throughout the program's execution as with traditional dynamic optimizers, protean code allows the original binary to execute continuously and diverts control flow only at a set of virtualized points, allowing rapid and seamless rerouting to the new code variants. In addition, the protean code compiler embeds IR with high-level semantic information into the program, empowering the dynamic compiler to perform rich analysis and transformations online with little overhead. Using a fully functional protean code compiler and runtime built on LLVM, we design PC3D, *Protean Code for Cache Contention in Datacenters*. PC3D dynamically employs non-temporal access hints to achieve utilization improvements of up to 2.8x (1.5x on average) higher than state-of-the-art contention mitigation runtime techniques at a QoS target of 98%.

Keywords-compiler; dynamic compiler; optimization; cache; resource sharing; datacenter; warehouse scale computer

I. INTRODUCTION

Large enterprises such as Google and Facebook build and maintain large datacenters known as *Warehouse Scale Computers* (WSCs) dedicated to hosting popular user-facing webservices along a variety of support applications. These datacenters are expensive and resource-intensive, with price tags now being measured in the billions of dollars [1, 2] and energy demands that require dedicated power plants.

Maximizing the efficiency of compute resources in modern WSCs is an important challenge that is rooted in finding ways to consistently maximize server utilization to minimize cost. The strategy of co-locating multiple applications on a single server has proved critical for this objective [3–6]. However, a significant challenge that emerges from the unpredictable dynamism in WSCs and limits our ability to co-locate is the threat of violating the *quality of service* (QoS) of user-facing latency-sensitive applications. Sources of dynamism include (1) fluctuating user demand (load) for user-facing applications, (2) highly variable co-locations between user-facing and batch applications on a given machine, and (3) constant turnaround on each server; when an application completes, new applications are mapped to the server.

Despite this dynamism, a capability missing in the WSC system software stack is the ability to dynamically transform and re-transform executing application code, which limits the design space when designing solutions to deal with the dynamism found in WSCs and leads to missed optimization opportunities. An example of such an optimization is to apply software non-temporal memory access hints to an application code to reduce its cache allocation and protect the QoS of its user-facing latency-sensitive co-runners. Modern ISAs, such as x86 and ARMv8 [7, 8], include prefetch instructions that hint to the processor that a subsequent memory access should not be cached. This instruction provides a mechanism that can cause an application to occupy more or less shared cache, and thus can enable higher throughput co-locations while protecting the QoS of high priority co-runners. However, it is difficult to leverage these hints effectively without a mechanism to dynamically add and remove them in response to changing conditions on the server.

The need for such a mechanism is also motivated by recent work [9, 10] that uses a 'napping' mechanism to reduce pressure on shared resources. ReQoS [10], for example, is a static compiler-enabled dynamic approach that throttles low-priority applications to allow them to be *safely* co-located with high-priority co-runners, guaranteeing the QoS of the high-priority co-runners and improving server utilization. However, due to the inability to transform application code online, these approaches are limited to using the heavy handed approach of putting the batch application to sleep, i.e., napping, to reduce pressure on shared resources.

While the advantages of a mechanism for online code transformation are clear, designing such a mechanism that is deployable in production environments has proved challenging. Despite a substantial body of prior work and having been shown to be useful in many problem domains [10–28], dynamic compilation has not been widely adopted, particularly in production and commercial domains. Several challenges have prevented the realization of *deployable* dynamic compilation:

- **Overhead** - It has been reported that companies such as Google tolerate no more than 1% to 2% degradation in performance to support dynamic monitoring approaches in production [29]. The high overhead that is common in traditional dynamic compilation frameworks has served as a barrier to adoption in these performance-critical datacenter environments.
- **Generality and Low Complexity** - To avoid hardware lockin and overly complex software maintenance, a deployable dynamic compilation system should impose little

or no burden on application developers and should require no specialized hardware support.

- **Transformation Power** - Traditional dynamic optimizers raise native machine code to an intermediate representation before applying transformations. This approach limits the power of the transformations due to loss of source level information. Having the ability to apply transformations online that are as powerful as static compilation significantly impacts the flexibility of the dynamic compiler.
- **Continuous Extropsection** - In a highly dynamic environment where multiple applications co-run, specializing code to runtime conditions should be done both *introspectively*, based on a host program's behavior, and *extrospectively*, based on *external* applications that are co-located on the same machine. To accomplish this, a runtime code transformation system must be aware of changing conditions for both itself and its neighbors, applying or undoing transformations accordingly.

In this paper, we introduce *protean code*, a general-purpose, near-free approach to monitoring, regenerating and replacing the code of running applications with semantically equivalent, specialized code versions that reflect the demands of the execution environment. Protean code is a co-designed compiler and runtime system built on top of LLVM [30]. At compile time, the program is prepared by a compiler pass that virtualizes a selected subset of the edges in its control flow and call graphs, providing hooks through which the runtime system may redirect execution. This novel mechanism allows the runtime system, including the dynamic compiler, to operate asynchronously while the application continuously runs. The compiler also embeds a copy of the program's intermediate representation (IR) into the data region, to be utilized by the runtime compiler for rapidly performing rich analysis and transformations on the program. The protean code runtime monitors all running programs on the system, generating and dispatching specialized program variants that are tailored to the particular conditions detected on the system at any given point in time. Protean code addresses the aforementioned challenges in the following ways:

1. *Low Overhead* – Diverting program control flow through selectively virtualized points introduces near-zero (<1%) overhead and provides a seamless mechanism through which the runtime compiler introduces new code variants as they become ready.
2. *General and Flexible* – To enact optimizations, protean code requires no support from the programmer or any specialized hardware. The design of protean code optimizations is in the purview of compiler writers, and protean code can be deployed for large applications on commodity hardware.
3. *Transformation Power* – Protean code embeds the IR into the program at compile time, which in turn is used by the runtime compiler as the starting point for analysis and optimization. Using the IR gives the runtime compiler the flexibility of a static compiler in terms of the analysis and optimization options that are available.
4. *Continuous Extrospection* – The protean code runtime uses program counter samples along with inter- and intra-core hardware performance monitors to detect changes to both *host* and *external* applications co-located on a single machine. This approach allows the runtime to react to highly dynamic environments by revisiting compilation choices introspectively as program phases change or extrospectively as the environment changes.

With the protean code mechanism in place, we design *Protean Code for Cache Contention in Datacenters* (PC3D), an approach that generates and deploys code transformations to change how applications consume shared cache resources. To tune cache occupancy based on dynamically changing system conditions, PC3D monitors changes in the behavior of the host program and its external co-running applications via a lightweight co-phase[1] analysis scheme. PC3D reacts to co-phase changes by generating, dispatching and evaluating code variants to discover how to mix cache pressure reduction transformations with napping in order to both meet the QoS of high-priority applications while maximizing the performance of low-priority applications. The specific contributions of this paper are as follows:

- **Protean Code Compilation** - We introduce *protean code*, a fully functional co-designed compiler and runtime system for enacting general purpose compiler transformations at runtime with an average overhead of less than 1%.
- **Dynamic Cache Pressure Mitigation** - We describe Protean Code for Cache Contention in Datacenters (PC3D), a dynamic approach to mitigating cache pressure in software via online compiler transformations.
- **Code Transformation Search** - We describe a runtime search algorithm that allows for the rapid discovery of an effective set of code transformations for cache pressure reduction, as well as how elements of the search generalize to other classes of online compiler transformations.
- **Real-system Evaluation** - We evaluate PC3D on a real system for a set of CloudSuite [31] webservice workloads, SPEC [32] and PARSEC [33] benchmarks, and SmashBench [4] microbenchmarks. We also perform an analysis of how deploying PC3D can impact energy and server provisioning requirements within full-scale datacenters.

We show that PC3D improves datacenter utilization by up to 2.9x and an average of 1.5x over the state-of-the-art software contention mitigation technique across a range of workloads, while meeting 98% QoS targets for high-priority latency-sensitive applications.

II. Background and Related Work

In this section, we first discuss recently proposed systems that enable safe application co-locations in datacenters. We then provide background on non-temporal memory access hint instructions, which PC3D dynamically exploits to alleviate cache contention and facilitate safe co-locations. Last, we present a comparison between prior dynamic compilation techniques and protean code.

A. Managing Shared Resources for Co-location

The benefits of enabling co-locations of batch applications with user-facing latency-sensitive applications have become well known [4–6, 13, 35]. Recently, techniques have been

[1] A co-phase is defined as the combination of the currently running phases among a program and its co-runners.

Table I. Comparison between protean code and prior dynamic compilation infrastructures

	ADAPT [23]	ADORE [24]	DynamoRIO [34]	Mojo [25]	protean code
Low Overhead	✓	✓			✓
Full Intermediate Representation					✓
Commodity Hardware	✓		✓	✓	✓
Programmer Unneeded		✓	✓	✓	✓
Extrospective					✓

proposed to make co-locations safe by dynamically throttling down the execution of low-priority applications by continuously introducing 'naps' of varying lengths and granularities into application execution [9, 10]. This approach has the effect of alleviating the pressure an application places on shared server resources, which allows high-priority applications to consume a larger share of resources to meet their QoS targets. However, as we show in this work, applying naps is an overly heavy handed approach and results in lower throughput than necessary to enforce QoS. We use one of these recent works, ReQoS [10], as the baseline for our technique.

Another technique, cache partitioning, has been used to explicitly control cache resource allocations, mitigating cache interference among co-running applications to ensure co-location safety. Hardware-based partitioning [36–39] allows for fine-grain control on the assignment of partitions, however it requires customized hardware and therefore has not been deployed in production systems. Software-based cache partitioning has been enacted with page coloring [14, 40–42], which controls the parts of cache an application can access via its page assignment in the operating system. Unfortunately, dynamically changing an application's cache allocation incurs significant performance penalties due to the overhead of page migration [40]. In addition, page coloring cannot be used in the presence of large pages [14, 42].

B. ISA Support For Temporal Locality Hints

The importance of quantifying and managing cache contention has been shown by prior work [10–19]. Temporal locality hints for memory accesses can be exploited to alleviate the pressure an application puts on the shared memory subsystem. Support for these hints is available across a broad range of instruction set architectures [43], including the modern high-performance platforms that appear in datacenter servers such as x86 [7] and ARMv8 [8].

Temporal locality hints can be employed in software to suggest how data should be cached. On x86, the `prefetchnta` instruction hints to the microarchitecture that data should be prefetched in a way that minimizes cache pollution. The motivating premise behind supporting these hints is that there are cases where software can identify and take advantage of the fact that a memory access lacks temporal locality – that is, it is likely to be evicted from cache before being used again. By hinting to the microarchitecture that a memory access lacks temporal locality, it may avoid evicting other, more useful data from the cache. In this work, we demonstrate how to leverage protean code to use temporal locality hints to dynamically change the cache pressure an application places on its co-runners.

C. Dynamic Compilation for Optimization

Dynamic compilation is a well-studied area that has been shown to be useful in a number of problem domains [10–28]. However, as discussed in the introduction, there are several limitations that prevent the wide adoption of these techniques in production datacenter environments for performance optimization. These limitations include high runtime overhead, dependence on programmer support or specialized hardware, limitations on the available optimizations, or inability to react to dynamic execution environments. Protean code is designed to overcome all of these limitations. Table I presents a comparison between protean code and several other dynamic compilation infrastructures.

III. PROTEAN CODE

Protean code is a novel dynamic compilation system specifically designed to address the challenges that prevent the wide adoption of traditional dynamic compilation techniques in datacenters. Figure 1 presents an overview of the design of protean code, composed of the *protean code compiler* (`pcc`) and the *protean code runtime*.

Protean Code Compiler Presented in the left side of Figure 1, `pcc` readies the host program for runtime compilation by making two classes of changes to the program. First, it virtualizes a subset of the edges in the program's control flow and call graphs. These virtualized edges then serve as points in the program's control flow at which the runtime system may redirect execution. Second, the compiler embeds several metadata structures, including an Edge Virtualization Table (EVT) and intermediate representation of the program, within the program's data region, which are used to aid the runtime system in dynamically introducing new code variants into the running program.

Protean Code Runtime Shown in the right side of Figure 1, the protean code runtime is responsible for monitoring a host program and its external execution environment in order to dynamically generate and dispatch code variants when needed. The runtime system first initializes by attaching to the program, discovering the program metadata and setting up a shared code cache from which the program can execute new code variants. To generate and dispatch a code variant, the runtime compiler, an LLVM-based compiler backend, leverages the IR. The new code variant is then inserted into the code cache and dispatched into the running host program by the EVT manager. During host program execution, the lightweight monitoring component of the runtime detects changes in both the host program phases and the external environment, including co-running applications, using samples of program counters and hardware performance monitors. In response to phase and environment changes, a decision engine determines when and how to generate new code variants and selects the appropriate variant for the current execution phase.

Design Principles The primary goal of protean code is to provide a dynamic code transformation solution that is

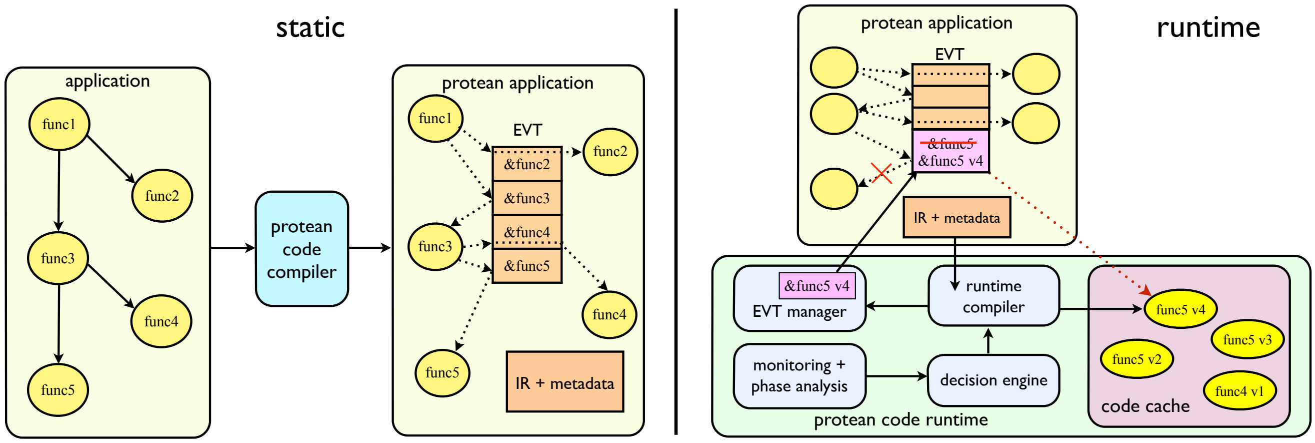

Figure 1. Overview of the protean code compiler and runtime

deployable in production datacenter environments and is powerful enough to enable techniques such as the PC3D runtime described in Section IV. There are three principles used in the design of protean code to realize this goal:

1. Maintaining absolute control of the program throughout execution, as in traditional dynamic compilers such as Dynamo [22] and DynamoRIO [34], leads to high overhead. Protean code instead allows the original binary to continuously execute and diverts the program control flow at a set of virtualized points, introducing negligible overhead. The runtime compiler is invoked asynchronously at controllable granularity, which also limits the overhead.
2. Many traditional dynamic compilers hoist the native machine code into an intermediate format at runtime to perform analysis and transformation [20, 22, 24, 44, 45], leading to overhead and the loss of rich semantic information present in IR from the static compiler [46]. Protean code embeds the IR into the program binaries, allowing the dynamic compiler to perform powerful analysis and transformations online with little overhead.
3. Protean code requires no support from the programmer or any specialized hardware, allowing it to be seamlessly deployed for large applications on commodity hardware. It leverages hardware performance monitors for lightweight monitoring, phase analysis and transformation selection, further minimizing overhead. A useful property of the application binaries produced by `pcc` is that they can be run without the runtime system, incurring negligible extra runtime overhead. In addition, once compiled with `pcc`, any protean code runtime can be used. These are particularly useful features in a datacenter environment, where rapidly changing conditions may dictate applying different classes of optimizations in the pursuit of different objectives to the same application binary.

A. Protean Code Compiler

The Protean Code Compiler (`pcc`) readies the host program for runtime compilation by (1) virtualizing control flow edges and (2) embedding meta-data in the program binary.

1) Control Flow Edge Virtualization: `pcc` adds a compiler pass to convert a subset of the branches and calls in the program from direct to indirect operations. By virtualizing a subset of edges, `pcc` sets up those edges as points in the programs execution where its control flow path may be easily altered by the protean code runtime to route program execution to an alternate variant of the code.

There are some important considerations to be made when selecting which edges to virtualize. Selecting too many edges or edges that are executed too frequently may result in unwanted overheads because indirect branches are generally slightly slower than direct branches. On the other hand, selecting only edges that are rarely executed risks introducing large gaps in execution during which new code variants are not executed. There is a large design space available when choosing how to virtualize edges. Our current approach to selecting edges is to virtualize only function calls, and only those where the callee function has more than one basic block. We find that this approach works well in practice, resulting in negligible overhead while ensuring that execution is promptly routed to the new code variants.

2) Program Metadata: Two types of program metadata are used by the protean code runtime to rapidly generate and dispatch correct, alternate code variants at runtime.

Edge Virtualization Table (EVT) A structure called the EVT contains the source and target addresses of the edges virtualized by `pcc`. The EVT is the central mechanism by which execution of the program is redirected by the runtime. To change execution, the runtime simply rewrites target addresses in the EVT to point to the new code variant.

Intermediate Representation `pcc` serializes, compresses and places the intermediate representation (IR) of the program into its data region, which the runtime decompresses then deserializes, leveraging it to perform analysis and transformations. Having direct access to the IR yields two significant advantages. First, it allows the runtime to avoid disassembling the binary, which can be difficult or impossible without access to fine-grain information about the executing code paths [47, 48]. Second, the alternative of hoisting the binary to IR, as is done in prior work, loses important semantic information and limits the flexibility of the compiler [46]. As an example of the utility of the IR, in this work PC3D gleans loop structure and nesting depth from the IR and uses that information to guide compilation decisions.

B. Protean Code Runtime

The protean code runtime is a set of mechanisms that work together to generate and dispatch code variants as the host program executes.

1) Runtime Initialization: Operating on an executable prepared by `pcc`, the runtime process begins by attaching to the program. It first discovers the locations of the structures inserted by `pcc` at compile-time, including the EVT and the IR. It then initializes a code cache, used to store the code variants generated by the dynamic compiler. Finally, because the EVT and code cache are structures that are shared between the program and the runtime and may be frequently accessed, the runtime sets up a shared memory region via an anonymous `mmap` to encompass both structures.

2) Code Generation and Dispatch: The runtime generates and dispatches code variants into the program asynchronously. When a new variant of a code region is requested, the dynamic compiler leverages the IR of the code region to generate the new variant. Once a new code variant has been generated, it is placed into the code cache. The EVT manager then modifies the EVT so that the target of the corresponding virtualized edge is the head of the newly minted variant in the code cache. The EVT update is a single atomic memory write operation on most modern platforms, and thus requires no synchronization between the host program and the runtime to work correctly.

Throughout these actions of the runtime process, execution of the program proceeds as normal until control flows through the virtualized edge, at which point control reaches the new code variant.

3) Monitoring, Phase Analysis and Decisions: The runtime supports both **introspection**, monitoring changes in the host program, and **extrospection**, monitoring changes in the execution environment. Based on this monitoring, the runtime makes decisions and adapts to changing system conditions such as application input/load fluctuation, starting or stopping of co-running applications, and phase changes among both the host programs and external programs.

Introspection For host programs, the runtime identifies hot code regions by sampling the program counter periodically through the `ptrace` interface. The runtime then associates the program counter samples with high-level code structures such as functions, allowing the runtime to keep track of which code regions are currently hot, as well as how hot regions change over time.

To identify phase changes, the runtime also leverages hardware performance monitors to track the progress of the program. Phases are defined in terms of the hot code identified by program counter samples described above as well as by the progress rate of the running applications using metrics such as instructions per cycle (IPC) or branches retired per cycle (BPC). Since hardware performance monitors are ubiquitous on modern platforms and can be sampled with negligible overhead, this approach allows the runtime to conduct phase identification in a manner that is both lightweight and general across hardware platforms.

Extrospection Similarly, for other external programs, the runtime tracks program progress and identifies phase changes via hardware performance monitors. Microarchitectural status and performance, using metrics such as cache misses or bandwidth usage, are also tracked through the performance monitor interface. Additionally, the runtime can be configured to use application-level metrics reported through application-specific reporting interfaces, such as queries per second or 99th percentile tail query latency for a web search application.

Dynamic Transformation Decisions The decision engine determines (1) when to invoke the dynamic compiler, (2) what transformations to apply, and (3) which variant to dispatch into the running program. The policies for these decisions depend on the optimization technique protean code employs, and can also be designed to control how often compilation occurs to limit any overhead introduced by running the dynamic compiler. We next present details for these policies for as they relate to PC3D, and our evaluation shows that the overhead of the dynamic compiler in PC3D is minimal.

IV. PC3D: ONLINE INTERFERENCE MITIGATION

Protean Code for Cache Contention in Datacenters (PC3D) is a protean code runtime that dynamically applies compiler transformations to insert non-temporal memory access hints, tuning the pressure a host application exerts on shared caches when the QoS of an external application is threatened. PC3D is implemented entirely as a runtime system that operates on an application prepared by the protean code compiler, requiring no changes to the basic protean code compiler setup described in Section III.

A. Overview of PC3D

The goal of PC3D is to find and dispatch variants of the host program code that contain a mix of non-temporal cache hints that allows the host's co-runners to meet their QoS targets while maximizing the throughput of the host.

Intuition of PC3D To ensure co-runner QoS, PC3D searches through a spectrum of program variants that have varying levels of cache contentiousness. The effectiveness of interference reduction of each variant is empirically quantified online by the protean code runtime. The best-performing program variant is then dispatched and runs until a new program phase or external application sensitivity phase is detected. In cases where relying solely on non-temporal cache hints is unable to ensure QoS of the external applications, naps are mixed with cache pressure reduction as a fallback.

B. Code Variant Search Space

PC3D generates and dispatches program variants that contain a selection of non-temporal cache hints. We refer to each such program variant as a bit vector $\overline{M} = \langle M_1, M_2, ..., M_N \rangle$, where N is the number of loads in the host program's code and $M_i \in \{0, 1\}$ represents the absence or presence of a non-temporal cache hint associated with the ith load. The set of program variants of this form is the set of all possible bit vectors of length N, which has a cardinality of 2^N. Figure 2 shows the four variants for a small code region ($N = 2$) within `libquantum`, where each of the

```
prefetchnta (%r14)          // m1
mov    %r13,%rsi
shl    $0x4,%rsi
mov    (%r14),%r8
prefetchnta (%r8,%rsi,1)  // m2
mov    (%r8,%rsi,1),%rax
```

(a) $<m_1, m_2> = <1, 1>$

```
prefetchnta (%r14)          // m1
mov    %r13,%rsi
shl    $0x4,%rsi
mov    (%r14),%r8
                            // m2
mov    (%r8,%rsi,1),%rax
```

(b) $<m_1, m_2> = <1, 0>$

```
                            // m1
mov    %r13,%rsi
shl    $0x4,%rsi
mov    (%r14),%r8
prefetchnta (%r8,%rsi,1)  // m2
mov    (%r8,%rsi,1),%rax
```

(c) $<m_1, m_2> = <0, 1>$

```
                            // m1
mov    %r13,%rsi
shl    $0x4,%rsi
mov    (%r14),%r8
                            // m2
mov    (%r8,%rsi,1),%rax
```

(d) $<m_1, m_2> = <0, 0>$

Figure 2. The set of variants for a small code region within `libquantum` on x86_64. Non-temporal hints and affected loads are shown in bold

Algorithm 1: Greedy search for a code variant $\overline{best}$ that uses the right mix of cache contention reduction and napping to maximize application performance

```
output: $\overline{best}$
/* enact/evaluate $\overline{0}$ to obtain the nap intensity ($nap_{\overline{0}}$) applied to the variant to meet co-runner QoS and the performance ($R_{\overline{0}}$) of the variant at that nap intensity */
$(nap_{\overline{0}}, R_{\overline{0}}) \leftarrow VariantEval(\overline{0}, 0, 1)$
$(nap_{\overline{1}}, R_{\overline{1}}) \leftarrow VariantEval(\overline{1}, 0, 1)$
$nap_{UB} \leftarrow nap_{\overline{0}}$, $nap_{LB} \leftarrow nap_{\overline{1}}$
$\overline{m} \leftarrow \overline{1}$, $\overline{best} \leftarrow \overline{1}$, $R_{\overline{best}} \leftarrow R_{\overline{1}}$
$i \leftarrow 1$
while $i \leq n$ and $nap_{LB} < nap_{UB}$ do
    $\overline{m} \leftarrow \langle m_1, ..., !m_i, ..., m_n \rangle$ // flip $i$th bit in $\overline{m}$
    $(nap_{\overline{m}} R_{\overline{m}}) \leftarrow VariantEval(\overline{m}, nap_{LB}, nap_{UB})$
    if $R_{\overline{best}} < R_{\overline{m}}$ then
        $R_{\overline{best}} \leftarrow R_{\overline{m}}$, $\overline{best} \leftarrow \overline{m}$, $nap_{UB} \leftarrow nap_{\overline{m}}$
    else
        $\overline{m} \leftarrow \langle m_1, ..., !m_i, ..., m_n \rangle$ // reject change
    end
    $i++$
end
return $\overline{best}$
```

four variants contains a different mix of non-temporal cache hints. PC3D searches these variants using a greedy search algorithm whose complexity is $O(N)$, described in detail in Section IV-D. However, even with a search complexity that is linear in the number of load instructions, the number of variants may still be large. To navigate this space efficiently, PC3D employs several heuristics.

C. Variant Search Space Reduction

PC3D focuses on the loads most likely to have a significant impact on application behavior. The heuristics employed to this end are as follows:

- **Exclude Uncovered Code** – Leveraging the PC samples collected for host program phase analysis, we expect code regions that never appear in those samples to have a minimal impact on cache pressure and application performance. Therefore, the loads from regions not appearing in the PC samples are pruned from the search space prior to the search. This reduces the number of loads that must be considered by an average of 12x.
- **Prioritize Hotter Code** – Furthermore, we expect code regions appearing more frequently in the PC samples to have a higher impact. Therefore PC3D prioritizes loads from hotter code regions in the search.
- **Only Innermost Loops** – For a range of contentious applications, we have observed that an average of more than 80% of the dynamic loads come from the maximum-depth loop(s) within each of the program's functions. Leveraging the program's IR, PC3D recognizes loops and their nesting depths, then prunes from the search space loads that are not at the maximum depth.

The number of static loads remaining after applying these heuristics is on average a factor of 44x smaller than the total number of static loads in the program (see Section V-B).

These heuristics focus the optimization decisions made by PC3D on the most important regions of code, a strategy we expect will also prove to be useful among other protean runtimes. After PC3D applies these heuristics, its search is limited to variants that are of the form $\overline{m} = \langle m_1, m_2, ..., m_n \rangle$, where $m_i \in \{0, 1\}$. $\overline{m}$ is a bit vector of the n loads from innermost loops among active code regions in the program phase, ordered roughly by how much impact they are expected to have on execution. For convenience, we refer to the variant where every load lacks a non-temporal hint as $\overline{m} = \overline{0}$ and its converse, the variant where every load has a non-temporal hint, as $\overline{m} = \overline{1}$.

D. Traversing the Variant Search Space

The variant search is guided by Algorithm 1. The search begins by evaluating variants $\overline{0}$ and $\overline{1}$, which are the variants that exert the most and least amount of cache pressure, respectively, out of all the variants in the search space. Because these variants are at the extremes of cache pressure, they are also at the extremes of the nap intensity required to meet co-runner QoS targets, and therefore may be viewed as lower and upper bounds, respectively, for the nap intensity that would theoretically be required to satisfy co-runner QoS for *any* program variant. As we discuss shortly, these bounds are used to limit the range of nap intensities that are evaluated for each variant, improving how quickly PC3D can converge on the right code variant.

Using $\overline{1}$ as a starting point, the algorithm steps through loads in the order of decreasing importance. For each load, the algorithm revokes the load's non-temporal hint, then calls $VariantEval$ (Algorithm 2) to enact the resulting code variant and evaluate whether that revocation improves the application's performance given the particular level of cache pressure produced by that variant along with the level of nap intensity required to allow the application's co-runners to meet their QoS targets. If the incremental change is found to have improved application performance, the change is

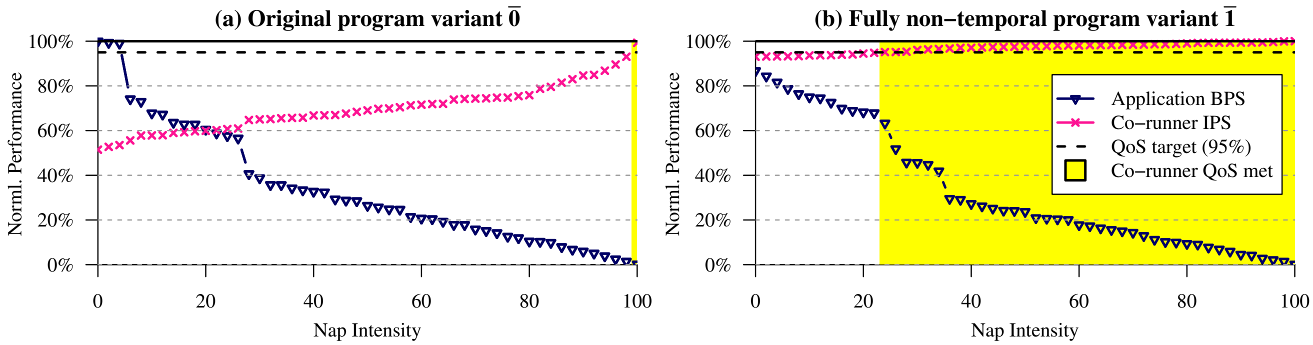

Figure 3. Online empirical evaluation for two variants of `libquantum` (application) running with `er-naive` (co-runner)

Algorithm 2: *VariantEval*, evaluation of a single program variant in PC3D

```
input : $\overline{m}$, $nap_{LB}$, $nap_{UB}$
output: $nap_{\overline{m}}$, $BPS_{\overline{m}}$
$nap_{cur} \leftarrow (nap_{LB} + nap_{UB})/2$
$BPS \leftarrow 0$
generate and dispatch variant $\overline{m}$
while $nap_{LB} < nap_{UB}$ do
    set_nap_intensity ($nap_{cur}$)
    if QoS of co-runners is satisfied then
        $nap_{UB} \leftarrow nap_{cur}$
        $BPS \leftarrow$ BranchesPerSecond $_{\overline{m}}$
    else
        $nap_{LB} \leftarrow nap_{cur}$
    end
    $nap_{cur} \leftarrow (nap_{LB} + nap_{UB})/2$
end
return ($nap_{cur}$, $BPS$)
```

kept and the algorithm repeats these steps on the next load. Otherwise, the change is rejected and the algorithm repeats these steps on the next load.

Note that each variant accepted as the best produces more cache pressure than the previous best version. Similar to the logic that was used to establish program-wide lower and upper bounds on the nap intensity range, upon accepting a variant as the best the upper bound on nap intensity is lowered to the nap intensity of the newly accepted variant.

E. Online Evaluation of Variants

PC3D searches for program variants that improve application performance while meeting co-runner QoS. Guiding the search are empirical evaluations of a sequence of program variants, which are dispatched then evaluated against the current running set of co-runners. Each variant produces a particular level of cache contentiousness, and may need to run with a particular nap intensity to allow its co-runners to hit their QoS targets.

This concept is demonstrated in Figure 3, which presents the performance of two variants of `libquantum` (host application) running with `er-naive` (external high-priority co-runner) as a function of the nap intensity applied to `libquantum`. Performance of `libquantum` is reported as branches per second (BPS) normalized to its BPS while running alone, while performance of `er-naive` is reported as instructions per second (IPS) normalized to its IPS running alone. We use BPS for host applications since, unlike branch counts, their static instruction counts change with the insertion/removal of non-temporal hints. As Figure 3 shows, each of these two variants exerts a different level of cache pressure on `er-naive`, and thus given a hypothetical QoS target of 95% for `er-naive`, a different level of nap intensity is required to allow `er-naive` to hit its QoS target. In this example, the `libquantum` variant in 3(a) requires a nap intensity of 99% to allow `er-naive` to meet its QoS target, while the variant in 3(b) requires a nap intensity of just 23%. At those respective nap intensities, the performance of variant (b) is far better than that of (a).

When evaluating a variant dynamically to discover the minimum nap intensity needed to meet co-runner QoS, PC3D need not evaluate the entire spectrum of nap intensities. The performance of both the application and its co-runners are monotonic as a function of nap intensity, so PC3D organizes the variant evaluation as a binary search over the range of nap intensities, shown in Algorithm 2. To reduce the search even further, PC3D performs the binary search only within the range of nap intensities between the lower and upper bounds established by evaluating other variants.

F. Monitoring Co-runner QoS

PC3D continuously monitors application co-runners to measure their quality of service (QoS). In this work, we use co-runner instructions per second (IPS) relative to the IPS running without the host application as a proxy for QoS. To measure co-runner IPS without the host, PC3D uses a flux approach similar to the mechanism described in [9], in which the host is put to sleep for a short period of time (40ms in our work) and performance measurements are taken while the co-runners execute without interference from the host. We deploy one such measurement every 4 seconds, allowing the flux technique to be deployed with very little (1%) overhead.

V. Evaluation

Methodology The protean code static compiler and runtime compiler are implemented on top of LLVM version 3.3. When compiling protean code or non-protean code benchmarks, compilation is done with `-O2`. All experiments are performed on a quad core 2.6GHz AMD Phenom II X4 server. Applications used throughout the evaluation are drawn from CloudSuite [31], the SPEC CPU2006 benchmark suite [32], the PARSEC benchmark suite [33] and SmashBench [4].

A. Performance of Protean Code

Virtualization Mechanism Overhead First we investigate the baseline cost of virtualizing execution with protean code and compare this cost with that of virtualizing execution with DynamoRIO [34]. DynamoRIO is a state of the art binary translation-based dynamic compiler, chosen as a baseline because it is a mature software project that is actively maintained and is well known for its low overhead relative to other dynamic compilers [49, 50].

Figure 4 shows the overhead for SPEC applications compiled as protean code relative to the non-protean code version of the benchmark. The base performance overhead of protean code mechanism is shown to be negligible, less than 1% on average. DynamoRIO, on the other hand, introduces an average of 18% overhead when performing no code modification. The primary distinction between binary translation and protean code is that protean code performs compilation asynchronously, out of the application's control flow path. Running protean code is low overhead because the application is allowed to continually execute, even when code is being compiled and dispatched. Binary translation incurs higher overhead because it requires all execution to occur from the code cache or interpreter, and thus control is continually diverted from the application back to the binary translation system.

Dynamic Compilation Overhead The protean code runtime runs in its own process and performs compilation asynchronously with respect to the running host application, employing a dynamic compiler to introduce new code variants into the running host program. We next present a set of dynamic compilation stress tests to demonstrate the impact of the level of activity of the dynamic compiler. In these experiments, the host program is run with a protean runtime configured to periodically recompile randomly selected functions throughout the life of the running application.

Figure 5 shows the results of these experiments for the SPEC benchmarks for a range of different time intervals between recompilations, where the runtime process (including the dynamic compiler) uses a dedicated physical core. The results show that this causes the dynamic compiler to generate very little overhead to the host program, even when performing recompilation every 5ms. We note that the LLVM compiler backend uses an average of around 5ms to compile a function, so the 5ms trigger interval results in the dynamic compiler being active almost continuously. Figure 6 presents, for the SPEC benchmarks, the average performance overhead of performing the same dynamic compilation stress tests, with the runtime on the same core as the host or on a separate core from the host.

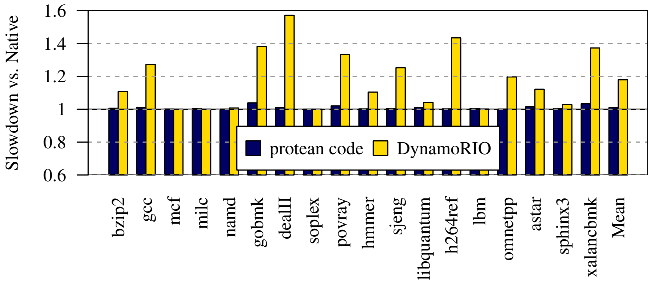

Figure 4. Dynamic compiler overhead when making no code modifications (normalized to native execution)

While executing the runtime on a separate core introduces minimal overhead no matter how frequently code generation is performed, the overheads of performing the compilation on the same core as the host program can be significant in extreme cases where compilation is nearly continuous. In an era of multicore and manycore processors, and particularly in datacenter environments, the common case is for cores to be heavily underutilized. For example, Google reports typical server utilization levels of 10-50% [51]. Nevertheless, in such instances where no separate core is available for the runtime, this overhead can be controlled by limiting the frequency of recompilation. As shown in Figure 6, the overhead of recompilation on the same core becomes negligible at 800ms.

Cycles Dedicated to the Runtime A unique feature of protean code is that the work of dynamic compilation of a host program may be offloaded to use otherwise spare cycles on the host server, putting those cycles to work for the benefit of the running applications. While the demand on the runtime to generate new variants is inevitably a function of the optimization objective, in PC3D the CPU utilization levels of the dynamic compiler and the entire runtime are quite low. Figure 7 presents the percentage of the server's cycles used by the PC3D runtime to manage a variety of batch applications, which is less than 1% in all cases.

B. PC3D Variant Search Heuristics

PC3D searches a set of program variants to arrive at a variant that improves the host application performance in the presence of some set of external applications. One of the keys to making this approach effective is to locate good code variants quickly. To accomplish this, PC3D employs several heuristics, described in detail in Section IV-C, to reduce the number of load instructions considered in the search. Figure 8 evaluates how effective these heuristics are across a set of contentious applications. Each cluster shows the number of loads that must be considered by the search as each successive heuristic is applied, normalized to the total number of loads in the application. Where there are multiple phases in a program, Figure 8 presents the average number of loads across all phases. Absolute counts of the number of loads that appear in each program are also included as numbers at the top of the plot.

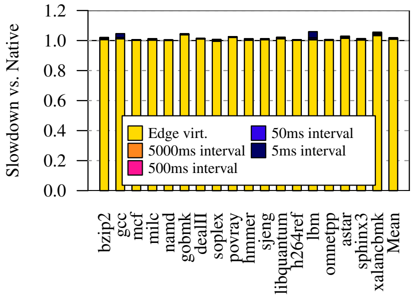

Figure 5. Dynamic compilation stress tests; compilation occurs on a separate core from the host application

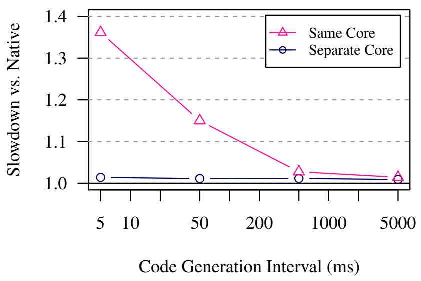

Figure 6. Dynamic compilation stress tests on separate vs. same core

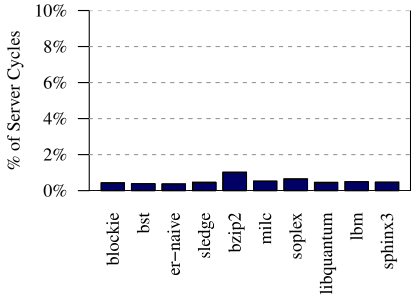

Figure 7. Average fraction of server cycles consumed by the PC3D runtime

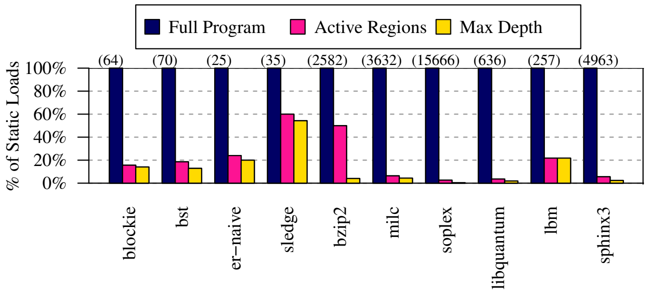

Figure 8. Heuristics significantly reduce the search space for PC3D. Static load counts of the full programs are presented in parentheses above the bars

Table II. APPLICATIONS USED IN DATACENTER EXPERIMENTS

	Host (batch) applications	**External (latency-sensitive) apps**
CloudSuite	-	`web-search`, `media-streaming`, `graph-analytics`
SPEC CPU2006	`bzip2`, `milc`, `soplex`, `libquantum`, `lbm`, `sphinx3`	`libquantum`, `mcf`, `milc`, `omnetpp`, `xalancbmk`
SmashBench	`bst`, `blockie`, `er-naive`, `sledge`	`bst`, `er-naive`
PARSEC	-	`streamcluster`

As described in Section IV-C, PC3D first discards loads from uncovered code – code regions that appear to the runtime system to have never executed during the current phase. On average, discarding loads from uncovered code results in a reduction of the search space by a factor of 12x. Second, PC3D extracts loop structure from the IR and discards each load that is not at the maximum loop depth within each function.

Overall, these heuristics are effective, reducing the number of static loads examined in the search by an average factor of 44x while covering more than 80% of the dynamic loads. It is notable that the reduction in number of loads is largest for programs with high load counts, such as `soplex` (15666 loads reduced to 57) and `sphinx3` (4963 loads reduced to 116), showing that the heuristics help keep the variant search manageable even for programs that have large code bases.

C. Utilization Improvements from PC3D

In this section we evaluate PC3D, showing its impact on server utilization and application QoS when running batch applications with latency-sensitive webservice applications, including `web-search`, `media-streaming` and `graph-analytics` from CloudSuite. The set of latency-sensitive and batch applications we evaluate are presented in Table V-C. For these experiments, QoS is presented as the instructions per second (IPS) an application achieves normalized to its IPS running alone on the server. Using IPS in this fashion as a proxy for QoS is consistent with practices in industry [52], where simple performance monitors are collected regularly and ubiquitously via mechanisms such as the Google Wide Profiler (GWP) [29] and used for making QoS estimates. Likewise, we present application utilization as the branches per second (BPS) measured by PC3D as a fraction of the BPS the non-protean version of the application achieves while running alone on the server. BPS is a useful metric in this case because PC3D introduces control-invariant code transformations that may include executing extra non-temporal access hint instructions in key code regions.

In these experiments, the latency-sensitive application runs on a single core of the server, while the contentious batch application runs on another single core. The contentious batch application is compiled with the protean code compiler, and may be modified dynamically to be less cache contentious if PC3D detects that the latency-sensitive application fails to meet its QoS target. The PC3D runtime consumes only a small fraction of the cycles on the server (Figure 7), monitoring all running applications to detect co-phase changes, checking that the latency-sensitive application meets its QoS target, and potentially introducing transformations that improve the cache contentiousness of the batch application.

Live Webservices Figures 9, 10 and 11 show the utilization gains achieved by PC3D over a policy of disallowing co-locations on a series of benchmarks as they run with `web-search`, `graph-analytics`, and `media-streaming`. Each cluster of bars shows the results of a particular batch application running against one of the webservices. The three bars in each cluster show the utilization gained with QoS targets of 90%, 95% and 98%. As applications co-run with `web-search`, they show an average utilization gain of 49% when a 98% QoS target is used. When less stringent QoS targets are in place, PC3D must mitigate contention to a lesser degree, which allows them to achieve higher utilization rates. With a 95% QoS target, the av-

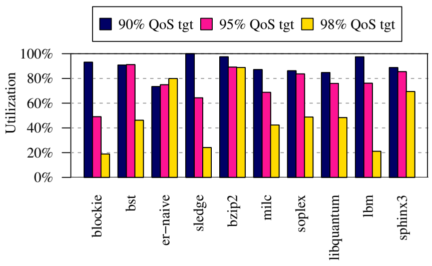

Figure 9. Utilization improvement of various applications running with `web-search`

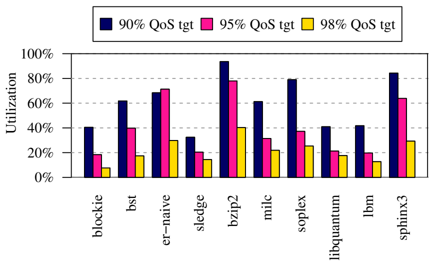

Figure 10. Utilization improvement of various applications running with `media-streaming`

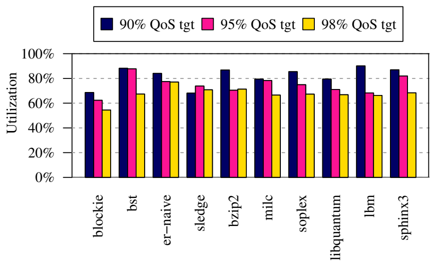

Figure 11. Utilization improvement of various applications running with `graph-analytics`

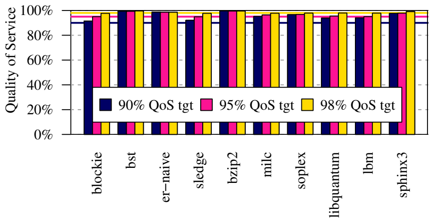

Figure 12. QoS of `web-search` running with various applications

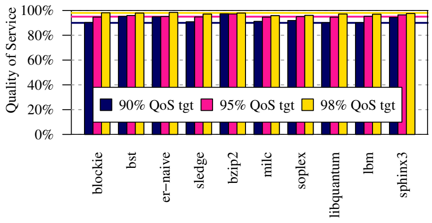

Figure 13. QoS of `media-streaming` running with various applications

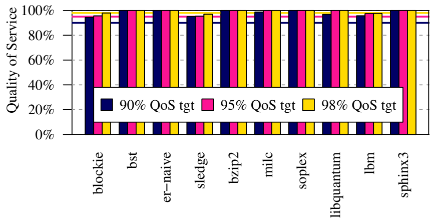

Figure 14. QoS of `graph-analytics` running with various applications

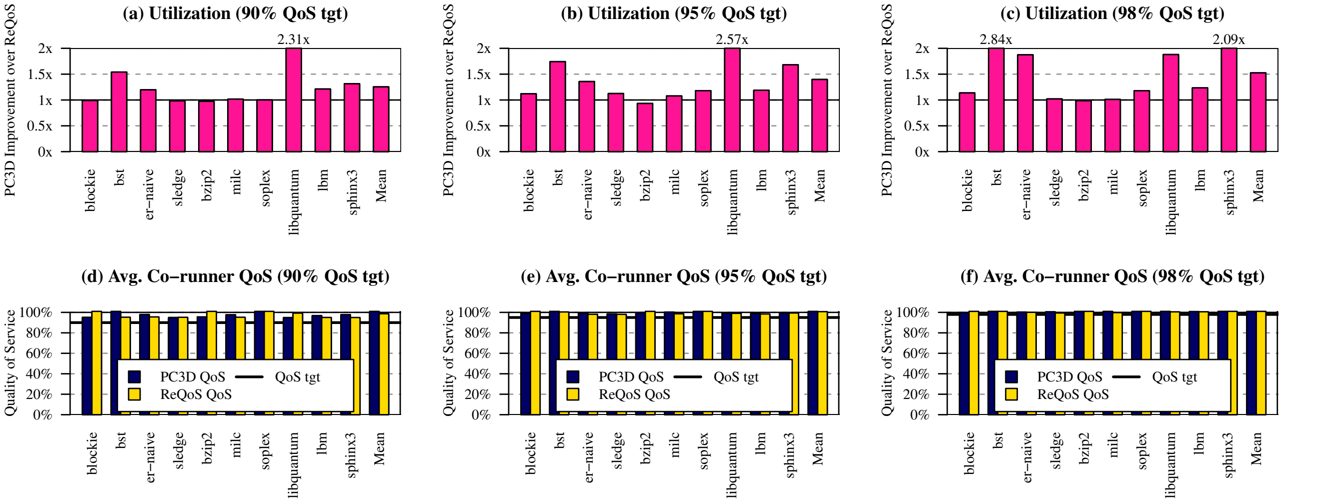

Figure 15. Utilization (top) and QoS (bottom) of PC3D vs. ReQoS, presented as the average across all CloudSuite, SPEC and SmashBench applications

erage utilization is 67% and with a 90% QoS target the utilization gain is 81%. Similarly, utilization improvements for `graph-analytics` are 67%, 75%, 82% for the three QoS targets. `media-streaming` is more sensitive to contention than `web-search` and `graph-analytics`, where we observe utilization improvements of 22%, 40% and 60%. Overall, these results show that PC3D consistently delivers substantial utilization gains, even in the presence of heavily contentious applications such as `libquantum` and `lbm`.

Figures 12, 13 and 14 present the QoS of the co-running webservice applications during the same set of experiments. These results show that PC3D reliably meets its QoS targets.

Comparison to State-of-the-Art Figure 15 presents the utilization achieved by PC3D compared to ReQoS [10], an approach for reducing application contentiousness that employs a hybrid static/dynamic approach to introduce naps into the running application. The results shown are the average utilization improvement of PC3D over ReQoS for a number of batch applications averaged over the entire spectrum of CloudSuite, SPEC and SmashBench co-runners. PC3D employs a napping mechanism similar to the mechanism used by ReQoS to throttle applications when reducing cache contention by dynamically inserting non-temporal hints is insufficient to allow the latency-sensitive co-runner to meet its QoS target, so in several cases ReQoS and PC3D show similar utilization levels. In a number of cases, however, PC3D gains far more utilization than ReQoS. For example, at a 98% QoS target, PC3D delivers over 2x the utilization

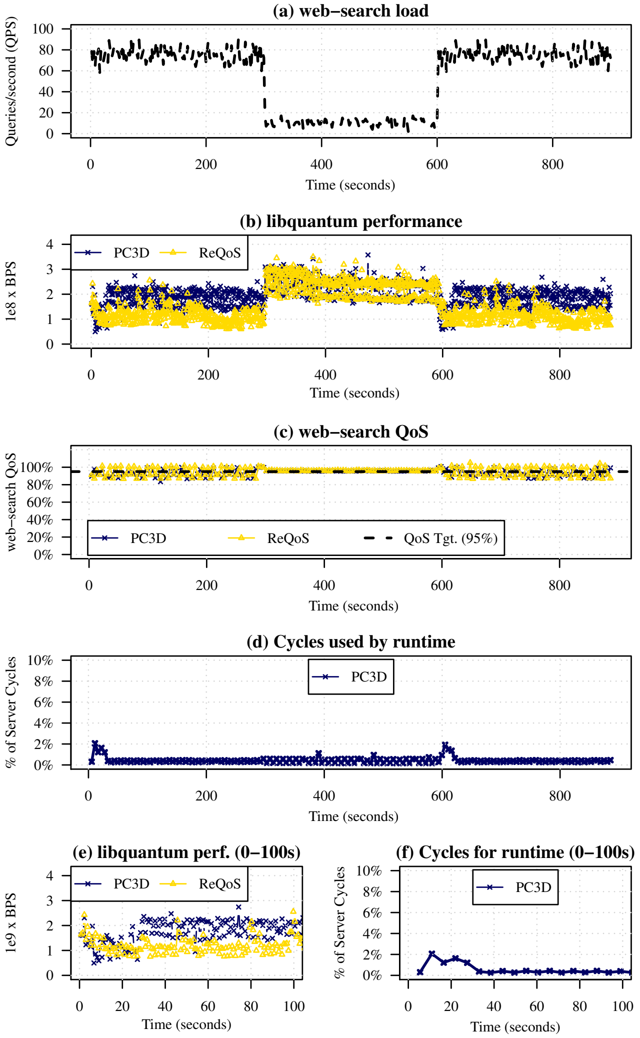

Figure 16. Dynamic behavior of `libquantum` running with `web-search` using the PC3D runtime

of ReQoS on `sphinx3` by finding an improved code variant, leading to far lower cache contentiousness at relatively small performance overhead to `sphinx3`. On average, PC3D improves utilization by a factor of 1.25, 1.45 and 1.52x at QoS targets of 90%, 95% and 98%, respectively. Figure 15 also includes the co-runner QoS, again presented as the average over the entire spectrum of co-runners. Both PC3D and ReQoS consistently meet the co-runner QoS targets.

D. Webservice with Fluctuating Load

To further evaluates how PC3D adapts to the dynamism in the application and its execution environment, Figure 16 presents the dynamic behavior of PC3D and ReQoS as `libquantum` runs with `web-search`. The load on `web-search` shifts over the course of the run, with the load pattern shown in 16(a). 16(b) shows a trace of the performance (branches per second) of `libquantum` over the same time frame. 16(c) shows the QoS of `web-search`, and 16(d) shows the cycles spent running the PC3D runtime.

Table III. Workload mixes for scale-out analysis

LS	`web-search`, `graph-analytics`, `media-streaming`
WL1	`libquantum`, `bzip2`, `sphinx3`, `milc`
WL2	`soplex`, `bst`, `milc`, `lbm`
WL3	`sledge`, `soplex`, `sphinx3`, `libquantum`

PC3D Dynamic Behavior `libquantum` initially (t=0) begins to execute alongside `web-search`. PC3D continuously monitors `web-search` as an external application, and detects that `libquantum` jeopardizes `web-search` QoS, so PC3D begins to search for alternate code variants for `libquantum` that allow `web-search` to meet its QoS while allowing `libquantum` to make better progress. The performance of `libquantum` during the variant search is shown in greater resolution in 16(e). By t=20, PC3D has arrived at an improved variant of `libquantum`, and PC3D allows it to run with this variant until a co-phase change is detected at t=300.

At t=300, the demand placed on `web-search` shifts, at which point PC3D detects a change in the behavior of `web-search`, causing it to revert `libquantum` back to its original (no non-temporal hints) variant. Until t=600, the original variant of `libquantum` runs at full speed because `web-search` is not sensitive to contention at low load.

At t=600, the load to `web-search` picks up and PC3D again searches for an improved variant that reduces cache contention. At t=620, the variant search ends and the improved variant of `libquantum` runs until the end of the experiment (t=900).

Cycles Consumed by PC3D Figure 16(d) shows the fraction of server cycles used by the PC3D runtime. Activity is minimal, kept to well below 1% of the server's cycles for the majority of the run. Two brief mini-spikes of up to 2% appear at t=0 (a higher-resolution view of this spike is presented in 16(f)) and t=600 as PC3D generates code to search for variants that improve the performance of `libquantum`.

ReQoS Dynamic Behavior Figures 16(b) and (c) also show the impact of ReQoS on the same run of `libquantum` and `web-search`. ReQoS adjusts the nap intensity, reacting to load changes at t=300 and t=600. During periods of high load it allows `web-search` to meet its QoS target strictly by applying naps to `libquantum`, causing `libquantum` to make significantly slower progress than it makes when running with PC3D.

E. Impact of PC3D at Scale

This section discusses the impact of deploying PC3D in a large-scale datacenter cluster that houses a mix of webservice and batch applications, showing that, by substantially improving server-level utilization, PC3D can have a large impact on the number of servers needed to house a particular workload and on the energy efficiency of the datacenter.

Server Requirements Figure 17 presents an analysis of the number of servers required to house a variety of webservice and batch application mixes. This analysis assumes a datacenter with 10k machines and the workload mixes described in Table III, with 10k instances of a latency-sensitive webservice (LS) with 95% QoS target along with 10k batch application instances comprised equally of one of the mixes shown in the table (WL). Running with PC3D, the 10k machines are able to achieve a particular level of throughput on each

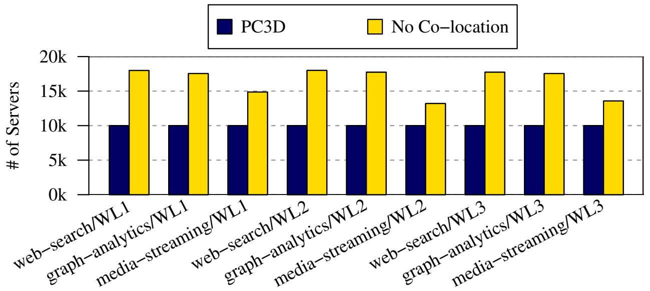

Figure 17. Server count required to run workload mixes for PC3D vs. no co-location

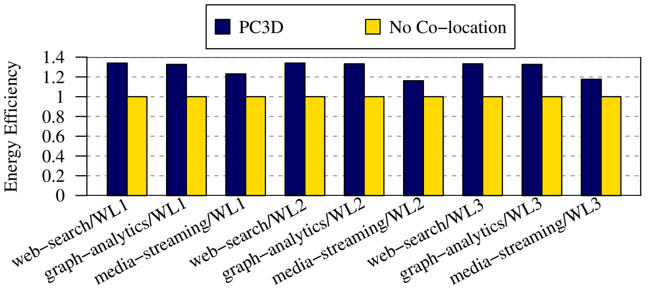

Figure 18. Normalized energy efficiency of workload mixes for PC3D vs. no co-location

application. Using a policy of disallowing co-locations, extra servers are needed to run the batch applications to achieve an equivalent level of throughput as the PC3D-enabled datacenter. Figure 17 shows that between 3.5k and 8k extra servers are needed on top of the original 10k servers to achieve a level of batch throughput that matches a PC3D-enabled datacenter.

Energy Efficiency Using a large number of extra servers also has a significant impact on the overall energy efficiency of the datacenter. Using a similar setup to the previous experiment, we employ a linear CPU utilization model to derive the power consumption of the servers within the datacenters, from which we compute the overall performance per Watt of each datacenter and derive energy efficiency comparisons of the datacenters. Figure 18 presents a comparison of the energy efficiency of the PC3D-enabled datacenter normalized to the No Co-location datacenter running the same workload at the same throughput, from which we observe that PC3D improves energy efficiency at the datacenter level by 18-34% across a spectrum of webservice and batch workloads.

VI. CONCLUSION

This work presents protean code, a novel approach to dynamic compilation designed to be deployable for performance optimization in datacenter environments. Protean code is nearly free of performance overhead, operates without any special hardware or programmer support, and has the flexibility of a robust static compiler. This combination of features gives protean code the capability to cope with the dynamism of modern datacenters by transforming and re-transforming running code to reflect the execution environment.

Using this lightweight dynamic compilation capability, we design Protean Code for Cache Contention in Datacenters (PC3D), a runtime approach to mitigating cache contention for live datacenter applications by dynamically inserting and removing software non-temporal cache hints, allowing batch applications to achieve high throughput while meeting latency-sensitive application QoS. On a spectrum of webservice and benchmark applications, PC3D achieves utilization improvements of up to 2.8x (average of 1.5x) higher than a recently published state-of-the-art contention mitigation runtime at a QoS target of 98%.

ACKNOWLEDGMENT

We thank our anonymous reviewers for their feedback and suggestions. We also thank Balaji Soundararajan for his help setting up experimental infrastructure. This research was supported by Google and by the National Science Foundation under grants CCF-SHF-1302682 and CNS-CSR-1321047.

REFERENCES

[1] Miller, Rich, "The Billion Dollar Datacenter," http://www.datacenterknowledge.com/archives/2013/04/29/the-billion-dollar-data-centers/, 2013, Online; accessed 23-May-2014.

[2] Metz, Cade, "Facebook Catapults $1.5 Billion Datacenter int Iowa," http://www.wired.com/2013/04/facebook-iowa-data-center/, Online; accessed 23-May-2014.

[3] J. Mars and M. L. Soffa, "Synthesizing contention," in *Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[4] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *International Symposium on Microarchitecture (MICRO)*, 2011.

[5] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[6] J. Mars and L. Tang, "Whare-map: heterogeneity in homogeneous warehouse-scale computers," in *International Symposium on Computer Architecture (ISCA)*, 2013.

[7] "Intel 64 and IA-32 Architectures Software Developers Manual. Volume 2: Instruction Set Reference A-Z," http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2014, Online; accessed 23-May-2014.

[8] "ARMv8 Instruction Set Overview," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.genc010197a/index.html, 2011, Online; accessed 23-May-2014.

[9] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *International Symposium on Computer Architecture (ISCA)*, 2013.

[10] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[11] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using os observations to improve performance in multicore systems," *IEEE Micro*, 2008.

[12] Y. Jiang, K. Tian, and X. Shen, "Combining locality analysis with online proactive job co-scheduling in chip multiprocessors," in *High Performance Embedded Architectures and Compilers (HiPEAC)*, 2010.

[13] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[14] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *High Performance Computer Architecture (HPCA)*, 2008.

[15] B. Bao and C. Ding, "Defensive loop tiling for shared cache," in *Code Generation and Optimization (CGO)*, 2013.

[16] S. W. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti, "A compiler-directed data prefetching scheme for chip multiprocessors," in *Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[17] L. Tang, J. Mars, and M. L. Soffa, "Compiling for niceness: Mitigating contention for qos in warehouse scale computers," in *Code Generation and Optimization (CGO)*, 2012.

[18] S. Rus, R. Ashok, and D. X. Li, "Automated locality optimization based on the reuse distance of string operations," in *Code Generation and Optimization (CGO)*, 2011.

[19] A. Sandberg, D. Eklöv, and E. Hagersten, "Reducing cache pollution through detection and elimination of non-temporal memory accesses," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[20] K. Ebcioğlu and E. R. Altman, "Daisy: Dynamic compilation for 100% architectural compatibility," in *International Symposium on Computer Architecture (ISCA)*, 1997.

[21] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *IEEE Symposium on Computers and Communications (ISCC)*, 2006.

[22] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *Programming Language Design and Implementation (PLDI)*, 2000.

[23] M. J. Voss and R. Eigemann, "High-level adaptive program optimization with adapt," in *Principles and Practices of Parallel Programming (PPoPP)*, 2001.

[24] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, "The performance of runtime data cache prefetching in a dynamic optimization system," in *International Symposium on Microarchitecture (MICRO)*, 2003.

[25] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies, "Mojo: A dynamic optimization system," in *Feedback-Directed and Dynamic Optimization (FDDO)*, 2000.

[26] W. Zhang, B. Calder, and D. M. Tullsen, "An event-driven multithreaded dynamic optimization framework," in *Parallel Architectures and Compilation Techniques (PACT)*, 2005.

[27] A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and J. M. M. Caamano, "Dynamic and speculative polyhedral parallelization using compiler-generated skeletons," *International Journal of Parallel Programming*, 2014.

[28] B. Breech, A. Danalis, S. Shindo, and L. Pollock, "Online impact analysis via dynamic compilation technology," in *International Conference on Software Maintenance (ICSM)*, 2004.

[29] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE Micro*, 2010.

[30] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization (CGO)*, 2004.

[31] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[32] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, 2006.

[33] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[34] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Code Generation and Optimization (CGO)*, 2003.

[35] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise qos prediction on real system smt processors to improve utilization in warehouse scale computers," in *International Symposium on Microarchitecture (MICRO)*, 2014.

[36] H. Kasture and D. Sanchez, "Ubik: efficient cache sharing with strict qos for latency-critical workloads," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[37] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *International Symposium on Computer Architecture (ISCA)*, 2013.

[38] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *International Symposium on Microarchitecture (MICRO)*, 2006.

[39] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *International Symposium on Computer Architecture (ISCA)*, 2011.

[40] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *European conference on Computer Systems (EuroSys)*, 2009.

[41] L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer," in *International Symposium on Microarchitecture (MICRO)*, 2008.

[42] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared l2 caches on multicore systems in software," in *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2007.

[43] R. Fu, J. Lu, A. Zhai, and W.-C. Hsu, "A study of the performance potential for dynamic instruction hints selection," in *Asia-Pacific Computer Systems Architecture Conference (ACSAC)*, 2006.

[44] J. Mars and M. L. Soffa, "Mats: Multicore adaptive trace selection," in *Workshop on Software Tools for MultiCore Systems (STMCS)*, 2008.

[45] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Code Generation and Optimization (CGO)*, 2003.

[46] V. Chipounov and G. Candea, "Enabling sophisticated analyses of x86 binaries with revgen," in *Dependable Systems and Networks (DSN)*, 2011.

[47] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *Performance Analysis of Systems and Software (ISPASS)*, 2010.

[48] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh, "Bird: Binary interpretation using runtime disassembly," in *Code Generation and Optimization (CGO)*, 2006.

[49] Q. Zhao, D. Bruening, and S. Amarasinghe, "Umbra: Efficient and scalable memory shadowing," in *Code Generation and Optimization (CGO)*, 2010.

[50] P. Feiner, A. D. Brown, and A. Goel, "Comprehensive kernel instrumentation via dynamic binary translation," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[51] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: an introduction to the design of warehouse-scale machines, 2nd edition," *Synthesis Lectures on Computer Architecture*, 2013.

[52] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *European Conference on Computer Systems (EuroSys)*, 2013.