Performance Characterization of .NET Benchmarks

Aniket Deshmukh*§, Ruihao Li*§, Rathijit Sen†, Robert R. Henry‡, Monica Beckwith‡, Gagan Gupta‡
*The University of Texas at Austin †Microsoft Gray Systems Lab ‡Microsoft
{a.deshmukh, liruihao}@utexas.edu, {rathijit.sen, robhenry, monica.beckwith, gagg}@microsoft.com

***Abstract*—Managed language frameworks are pervasive today, especially in modern datacenters. .NET is one such framework that is used widely in Microsoft Azure but has not been well-studied. Applications built on these frameworks have different characteristics compared to traditional SPEC-like programs due to the presence of a managed runtime. This affects the tradeoffs associated with designing hardware for such applications.**

Our goal is to study hardware performance bottlenecks in .NET applications. To find suitable benchmarks, we use Principal Component Analysis (PCA) to find redundancies in a set of open-source .NET and ASP.NET benchmarks and use hierarchical clustering to create representative subsets. We perform microarchitecture and application-level characterization of these subsets and show that they are significantly different from SPEC CPU17 benchmarks in branch and memory behavior, and hence merit consideration in architecture research. In-depth analysis using the Top-Down methodology reveals that .NET benchmarks are significantly more frontend bound. We also analyze the effect of managed runtime events such as JIT (Just-in-Time) compilation and GC (Garbage Collection). Among other findings, GC improves cache performance significantly and JITing could benefit from aggressive prefetching and transformation of hardware microarchitectural state to prevent frequent cold starts. As computing increasingly moves to the cloud and managed languages grow even more in popularity, it is important to consider .NET-like benchmarks in architecture studies.

I. Introduction

.NET is an open source, cross-platform software development framework comprising object-oriented programming languages and libraries [18]. Introduced by Microsoft, .NET is used to develop applications for a variety of platforms, ranging from the web to mobile to desktop to cloud to internet-of-things and more. .NET includes C#, F# and Visual Basic languages, provides a Common Language Runtime (CLR), and is supported on Windows, Linux, and macOS. The CLR generates code using a just-in-time (JIT) compiler and manages its execution while providing services for memory management, garbage collection (GC), type safety, thread management, exception handling, etc. ASP.NET is a framework for developing web apps and services using .NET and C# [17]. For example, .NET is commonly used to develop Microsoft Azure applications [7], Azure SQL DB services [2], serverless programming, Cosmos DB services [1], the Bing search engine [3], Stack Overflow [25], etc. Over 3700 companies contribute to .NET and it is one of the most popular development platforms with 6M users and growing [13].

Computer architects routinely use benchmark suites like SPEC [11], Renaissance [36], PARSEC [21], CloudSuite [28], MLPerf [37], etc., to explore hardware and software proposals. These suites primarily include programs written in non-managed languages such as C/C++/Fortran that execute directly on the host machine, and programs written in Java whose execution is managed by a runtime that provides services such as Garbage Collection (GC), Just-In-Time (JIT) compilation, etc. The microarchitectural behavior of managed languages is different from traditional programming languages. Besides Java, .NET is an important managed framework [9], but has not been studied well. Given its growing popularity [10], we believe that an in-depth study of .NET programs has long been due.

Although Java and .NET are both managed frameworks, they differ in implementation. For example, Java's adaptive and dynamic JIT optimization techniques and heuristics are different from .NET. Threading, algorithms and heuristics for GC are also different between the two. These and other differences can lead to different performance profiles. Also, Java benchmarks [12], [39], [22] are geared more towards Java Virtual Machine (JVM) studies. Hence, we posit that studying .NET benchmarks is essential for optimizing future architectures for managed languages. We focus on Online Transactional Processing tasks and core library functions that are key components in important real-world .NET applications.

To the best of our knowledge, this work is the first to present a detailed characterization of .NET benchmark suites — a microbenchmarks suite [19], and an ASP.NET benchmark suite containing datacenter web app frameworks [20].

- We use principle component analysis (PCA) and hierarchical clustering to create representative subsets of 8 benchmarks each from 2906 .NET microbenchmarks and 53 ASP.NET benchmarks to reduce experimentation time in architecture studies. We use a time-based metric similar to SPECspeed [16] to validate the accuracy of our subset. (§IV.)
- PCA based comparison shows that many .NET and ASP.NET applications exhibit significant differences compared to SPEC CPU17, particularly with respect to branch and memory hierarchy behavior, and hence should be included in architecture studies. (§V-C.)
- Instruction mix analysis shows that ASP.NET and .NET programs have similar distributions of branch, load and store instructions, but are different from SPEC CPU17 programs, which have more diverse branches, more loads and fewer stores. (§V-B.)
- Through raw performance counter results and Top-Down analysis, we show that I-cache, I-TLB and BTB misses contribute significantly to pipeline stalls in ASP.NET and

§Work done while interning at Microsoft. Both authors made equivalent contributions to this work.

.NET benchmarks unlike SPEC CPU17. We also show that contention at the Last Level Cache (LLC) is a major bottleneck when scaling ASP.NET-like web applications. (§VI-B2.)

- Experiments on an Arm machine reveal that Arm architectures and the associated software stack may not yet be tuned for .NET-like workloads as compared to the more mature Intel architectures, e.g., Arm TLB performance can be an order of magnitude worse than Intel. (§V-D.)
- While the adverse impact of frequent Garbage Collection has been studied ([31],[33]), we show that GC has a positive impact on cache performance and thus provides additional benefits when accelerated in hardware. Increasing GC aggressiveness shows an average $0.59\times$ reduction in LLC-MPKI for .NET applications. (§VII-A2.)
- Our studies also reveal that code JITing leads to frequent cold starts in the caches, TLB and program counter indexed micro-architectural structures such as the branch predictor. This necessitates better prefetching and transformation of micro-architectural state to avoid the performance penalty due to cold starts. (§VII-A1.)

II. .NET Benchmarks Overview

We obtained open source .NET [19] and ASP.NET [20] benchmarks, curated to test performance, from their respective Github repositories. Repositories are periodically updated; for this study, we used the snapshots listed in the references. The suites are briefly described below.

A. .NET

The .NET suite, written in C#, includes a total of 2906 microbenchmarks, divided into 44 categories, 21 of which are a collection of system-level benchmarks and 23 are application-level benchmarks. System-level benchmarks include basic .NET libraries, for mathematical functions (e.g., ABS, cosine, sine), file IO, network transmission, etc. Application-level benchmarks include real-world applications or algorithms which use data structures like linked lists, stacks, trees, etc. When experimenting, either a category as a unit or each benchmark individually can be run in a dedicated process resulting in either 44 or 2906 runs.

B. ASP.NET

The ASP.NET suite contains a total of 53 benchmarks, including those from the TechEmpower Web Framework Benchmarks [14], which measure the performance of various aspects of client-server constructs. The benchmarks consist of four components: server, client, database and benchmark driver to model the application's environment. The client sends requests to the server, which processes these requests while communicating with the database and returns a response to the client. The benchmark driver coordinates building and execution of the benchmark on both the client and the server, and gathers statistics. The server, client and database are all launched within docker containers. Once started, the benchmark driver can send commands to each component to run a variety of scenarios.

TABLE I: Characterization metrics.

Categories	Metrics	Normalization Unit	ID
ISA			
Inst Mix	Kernel instructions	Percentage	0
	User instructions	Percentage	1
	Branch instructions	Percentage	2
	Memory loads	Percentage	3
	Memory stores	Percentage	4
Micro-architecture Events			
CPI	Cycle per instruction	Per instruction	5
CPU Usage	CPU utilization	Percentage	6
Branch	Branch misses	MPKI	7
Cache	L1-dcache misses	MPKI	8
	L1-icache misses	MPKI	9
	L2 cache misses	MPKI	10
	LLC misses	MPKI	11
TLB	iTLB misses	MPKI	12
	dTLB load misses	MPKI	13
	dTLB store misses	MPKI	14
Memory	Memory read bandwidth	MB per sec	15
	Memory write bandwidth	MB per sec	16
	Memory page miss rate	Percentage	17
	Page faults	PKI	18
Run-time Events			
Garbage Collection	GC/Triggered	PKI	19
	GC/AllocationTick	PKI	20
JIT	Method/JittingStarted	PKI	21
Exception	Exception/Start	PKI	22
Contention	Contention/Start	PKI	23

III. Evaluation Methodology

A. Characterization Metrics

We characterized the benchmarks along three vectors, (i) ISA, (ii) microarchitecture events, and (iii) application-level run-time events, summarized in Table I. For ISA characterization, we examined the instruction mix. Following other studies [34], [32], [36], we measured CPI, CPU usage, branch, cache, TLB and memory subsystem events for microarchitecture characterization.

Run-time events can have significant impact on the performance of managed language workloads. We measured four types of events in our study. GC manages allocation and release of memory for an application, which affects cache, TLB, and memory behavior [23], [24]. JIT compilation intercedes the regular course of execution to generate and optimize machine code, which can impact I-cache, branch and memory performance [29]. Exceptions and thread contention are also included in our metrics, since they can be major factors in large real-world multi-threaded applications [27], [40].

The .NET microbenchmarks are short running applications, up to a few seconds long. To amortize their warmup overheads, we ran them 15 times and discarded the data from the first run. To measure steady state performance for ASP.NET (whose run-time is user-specified), we ran the benchmarks in warmup mode for a long duration and progressively reduced the warmup period while ensuring the steady state measurements had a variance of less than 5%.

TABLE II: Hardware configurations of 3 different machines. CPU = physical core, vCPU = logical core.

	Intel(R) Xeon(R) E5-2620 v4	Intel(R) Core(TM) i9-9980XE	Arm
ISA	x86-64	x86-64	AArch64
#CPU/#vCPU	16/32	18/18	32/32
OS	Ubuntu 16.04	Ubuntu 20.04	Ubuntu 20.04
L1d Cache	32KiB	32KiB	32KiB
L1i Cache	32KiB	32KiB	32KiB
L2 Cache	256KiB	1MiB	256KiB
L3 Cache	20MiB×2	24.8MiB	32MiB
Nom Freq	2.1GHz	3.0GHz	1.6GHz
Max Freq	3.0GHz	4.5GHz	2.2GHz

B. Hardware & Software Platforms

We characterized the benchmarks on two x86-64 and one AArch64 machines, summarized in Table II. The majority of our experiments were run on the Intel i9-9980XE machine with Ubuntu 20.04. Although widely used in embedded systems, Arm processors are being increasingly considered for servers, as seen in the NO.1 FUGAKU supercomputer [38]. Hence we included a commercial Arm platform in our evaluation. The CPU core in the single-socket Arm system can decode up to 4 and issue up to 6 micro-ops each cycle. The core includes 2 LSUs, a special store unit, a 128-entry loop buffer, a 180-entry ROB, dedicated I-TLB and D-TLB, and a 2K-entry secondary TLB. To validate our methodology for creating a representative subset of the benchmark suites, we use the Intel E5-2620 v4 machine as a baseline machine running Ubuntu 16.04.

To run ASP.NET benchmarks, we used two Intel i9-9980XE machines (with identical configurations). The server was run on one machine, and the client, database and benchmark driver on the other. All measurements were taken on the server machine. The latest stable .NET Core compiler, version 3.1.7 [8] was used to compile benchmarks. We used the *Linux perf* tool [4] for collecting microarchitecture hardware performance counters, and the *Lttng* tool [26] for collecting run-time traces. We used the *toplev* tool [15] for top-down analysis. We also logged the runtime stats generated by the .NET framework itself.

IV. Redundancy Analysis and Subset Creation

As a first step, we reduce the large corpus of benchmarks into a smaller representative set. Using such a subset simplifies architecture studies and is shown to be adequate [34], [35]. We create two subsets, one each for .NET and ASP.NET. For comparison, the SPEC CPU17 suite is also reduced to a subset.

A. Characterization Metrics Redundancy Analysis

As prior works suggests, four metrics can cover 90% of the variance in different workloads [34], [36]. We use a similar strategy to find potential redundancy within the 24 metrics in Table I.

Characterization metrics should cover factors contributing to the performance variance between workloads. Ideally, all metrics would be independent of each other without any correlation between them. This would imply that there is no redundancy in the characterization metrics. In reality, however, metrics are interrelated. For instance, changes in the LLC behavior can affect not only CPI but also L1/L2 cache performance [30]. This is true for runtime events as well. For example, GC settings can impact LLC performance [23].

To remove the correlation among these metrics, we leverage the PCA technique [41], [34], [36]. PCA reduces the dimension of data by converting an i-element input vector $X(X_1, X_2, ..., X_i)$ into a j-element, j<i, output vector $Y(Y_1, Y_2, ..., Y_j)$. In vector Y, each element is linearly uncorrelated, and the j elements in it are called Principal Components (PRCOs). For example, Y_1 represent the first PRCO. Each PRCO is a linear combination of various elements of the input vector with certain weights known as loading factors (matrix W in Equation 1).

$$Y_1 = \sum_{n=1}^{i} W_{1,n}X_n; Y_2 = \sum_{n=1}^{i} W_{2,n}X_n; ... \quad (1)$$

We picked the top four PRCOs for clustering since four PRCOs can cover the majority of the variance in benchmark suites [36]. This helped reduce the number of characterization metrics from 24 to 4.

Table III lists the loading factors of each metrics in descending order (we only list the top 3) for the four PRCOs. There are negative loading factors since we perform data standardization before the PCA. The variance of each PRCO is also listed; the top 4 PRCOs we selected cover 79% of the variance.

B. Benchmark Suites Redundancy Analysis

Prior works have demonstrated the redundancies in the SPEC CPU17 suite [34] using PCA. However, we cannot directly apply the same methodology to the .NET and ASP.NET suites. In the ASP.NET suite, performance is evaluated using throughput instead of execution time. Also, the .NET microbenchmarks can be analyzed as a set of 44 categories or 2906 individual benchmarks.

The first step in finding potential redundancies is to establish similarity between workloads which we define by the linkage distance of the first four PRCOs. We then use hierarchical clustering to group the workloads with high similarity based on the linkage distance table. The cluster hierarchy of the .NET microbenchmark suite when analyzed as a set of 44 categories is shown in Figure 1 as a tree. Each leaf node represents one benchmark. Two nodes with the shortest linkage distance merge into a parent node recursively until the root node is generated. Since benchmarks under a node are similar, a representative subset can be generated by picking one benchmark from each of the nodes at a given level. For example, based on the tree level in Figure 1, a 2-element representative subset can be generated by selecting one benchmark out of the first 42 benchmarks (from System.IO to SeekUnroll) and one out of the last two benchmarks (from System.Diagnostics to CscBench). When more than one choice was available, we picked one randomly.

TABLE III: Loading factors of the top 3 metrics on the four principal components.

PRCO1 (0.306)		PRCO2 (0.229)		PRCO3 (0.148)		PRCO4 (0.107)	
Metric	Load	Metric	Load	Metric	Load	Metric	Load
L2 MPKI	0.320	D-TLB store-MPKI	0.353	inst_mix_mem-stores	0.347	inst_mix_branch-instructions	-0.431
I-TLB MPKI	0.322	memory_bandwidth_read	0.451	branch MPKI	-0.357	inst_mix_mem-loads	-0.404
D-TLB load-MPKI	0.323	memory_bandwidth_write	0.407	gc/triggered	-0.318	inst_mix_mem-stores	-0.310

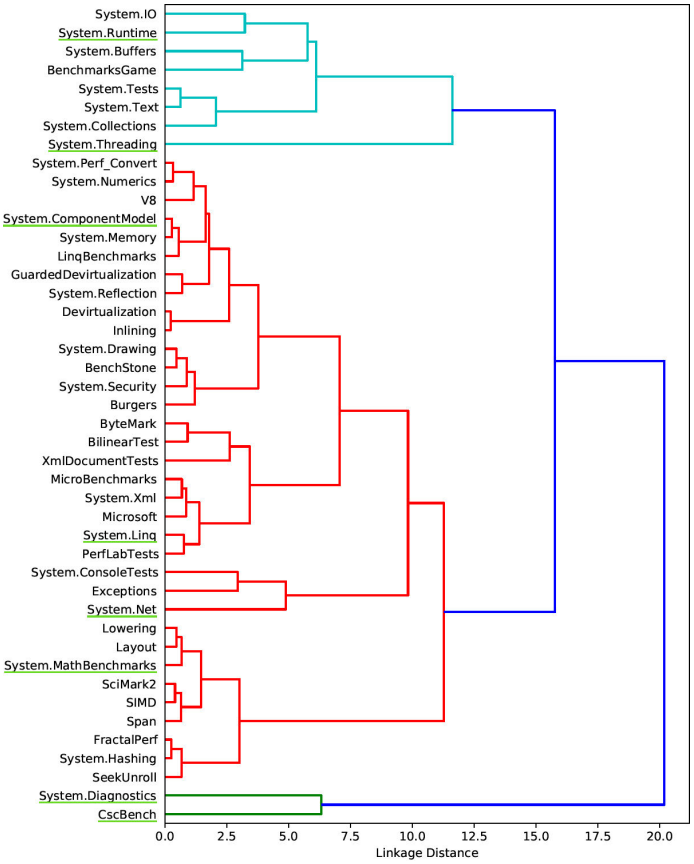


Fig. 1: Similarity between benchmarks in .NET suite. The subset we picked is underlined, and listed in Table IV.

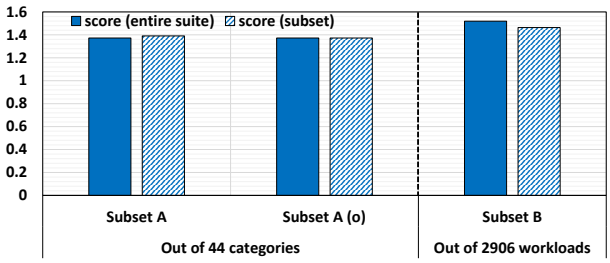


Fig. 2: Validation of .NET representative subset.

For the .NET benchmarks an 8-category representative subset, containing 305 workloads, is generated out of the 44 categories. We also generated a 64-element representative subset out of the 2,906 workloads. The same methodology is applied to the ASP.NET suite to obtain an 8-element representative subset. We also created an 8-element subset of the SPEC CPU17 suite. The representative subsets for the three benchmark suites are listed in Table IV.

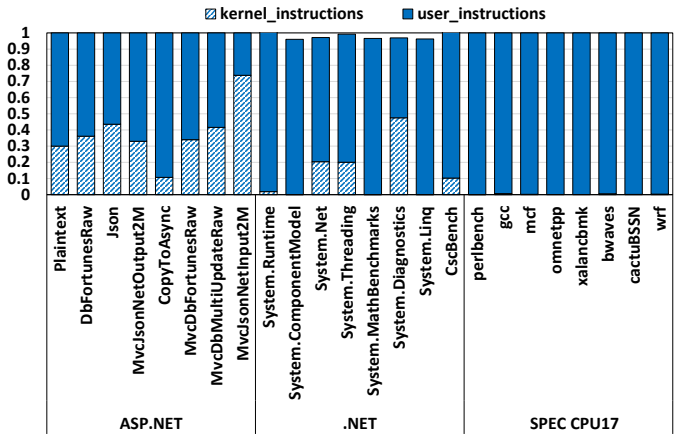


Fig. 3: Fraction of kernel instructions in each benchmark.

C. Validating Subsets

Once a subset is created, it is important to verify that it is representative. Prior work [34] used the **score** in SPEC to verify their selected subsets. The **score** of machine A is defined as $\frac{execution\ time\ on\ the\ baseline\ machine}{execution\ time\ on\ machine\ A}$. A single composite score can be computed by taking the geomean of the scores of the included benchmarks. We create similar scores for the .NET and ASP.NET suites using the Intel Xeon machine in Table II as our baseline, and the Intel Core machine as the machine A.

Figure 2 shows the validation results. We compute the composite score for the entire benchmark suite and compare with a composite score of only the reduced subset. The accuracy of Subset A (containing 8 out of 44 categories) and Subset B (containing 64 out of 2906 workloads) are 98.7% and 96.3% respectively. The Subset A(o) represents the "optimum" score of the 8-category subset, 99.9%. This is obtained by iterating over all possible combinations, and hence the higher accuracy. Subset A is more accurate than Subset B since it covers 8 categories containing 305 workloads as opposed to 64 individual workloads. We use Subset A in this study.

V. Comparison between Benchmark Suites

We use basic hardware counters and PRCOs to gain a better understanding of the coverage of the .NET, ASP.NET and SPEC CPU17 benchmarks and the differences between them.

A. Instruction Footprint

Both .NET and ASP.NET have a significant kernel contribution as compared to SPEC CPU17, shown in Figure 3. While the CLR is responsible for some of the kernel instructions, ASP.NET shows a much larger percentage of kernel instructions executed. We analyzed the performance

TABLE IV: Representative subsets of the benchmark suites.

.NET		ASP.NET		SPEC CPU17
System.Runtime	Basic scalar and array tests.	*DbFortunesRaw*	Renders sorted DB query results to HTML.	*mcf*
System.Threading	Thread kernel functions.	*MvcDbFortunesRaw*	Renders DB queries to HTML, MVC backend.	*cactuBSSN*
System.ComponentModel	Type converters.	*MvcDbMultiUpdateRaw*	Serializes multiple DB queries as JSON objects.	*wrf*
System.Linq	Language integrated query tests.	*Plaintext*	Returns plaintext strings from pipelined queries.	*gcc*
System.Net	Network kernel functions.	*Json*	Serializes a simple JSON document.	*omenetpp*
System.MathBenchmarks	Math libraries.	*CopyToAsync*	Reads POST query, returns plaintext result.	*perlbench*
System.Diagnostics	Kernel functions.	*MvcJsonNetOutput2M*	Sends 2MB JSON document, MVC backend.	*xalancbmk*
CscBench	Compiler and dataflow tests.	*MvcJsonNetInput2M*	Receives 2MB JSON document, MVC backend.	*bwaves*

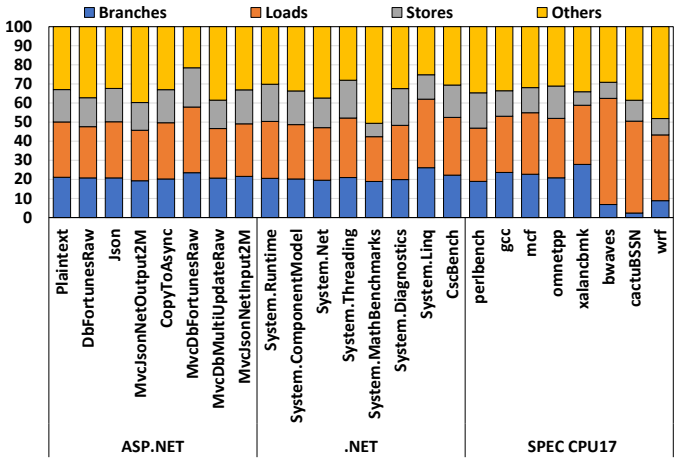
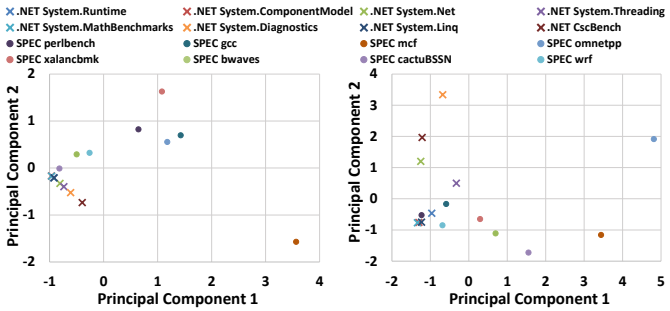
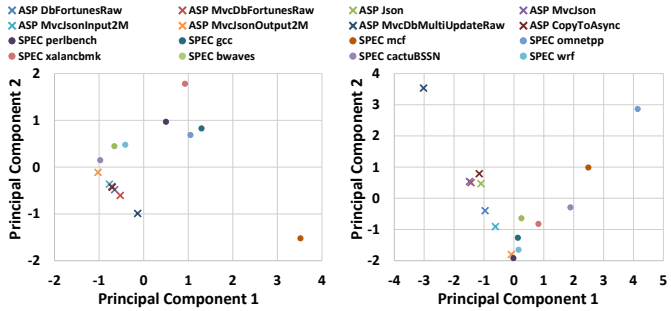


Fig. 4: Percentage of instructions types in each benchmark.



(a) Control flow behavior. (b) Memory behavior.

Fig. 5: Comparison between .NET and SPEC CPU17.



(a) Control flow behavior. (b) Memory behavior.

Fig. 6: Comparison between ASP.NET and SPEC CPU17.

counter samples in our data and found that this is primarily due to the code in the networking stack. Therefore for studying such benchmarks, full system simulations are needed to model kernel behavior and the impact of the networking stack (including the networking hardware) on performance.

B. Instruction Mix

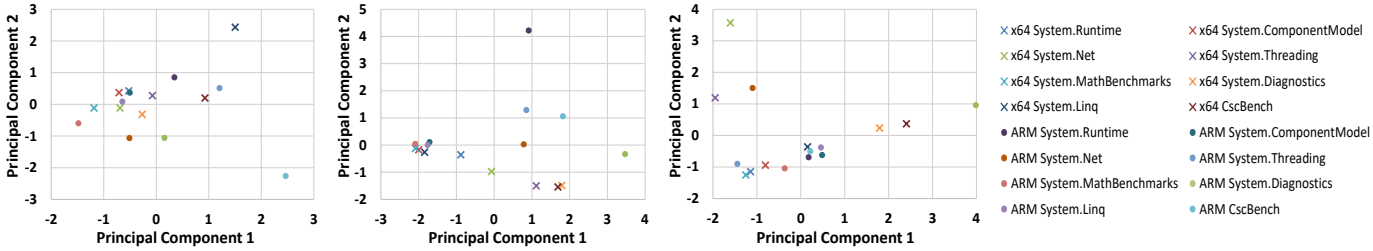
Figure 4 shows the breakdown of instructions in each benchmark. The ASP.NET and .NET benchmarks do not show much variety in the percentage of branches, loads and stores due to fairly simple user code and the common runtime among all the applications. SPEC programs are more diverse: xalancbmk has a higher proportion of branches, but the FP programs (bwaves, cactuBSSN, wrf) have much smaller proportion. SPEC programs have slightly more loads, with geomean (GM) of 35.2% vs ∼29% in ASP.NET and .NET, but fewer stores (GM of 11.5% vs ∼16% in ASP.NET and .NET). Individual program characteristics, e.g., data structure initialization in System.Diagnostics, and GC operations contribute to the higher stores in ASP.NET and .NET programs.

C. Using Principle Components for Comparison

The top four PRCOs obtained in §IV-A show that control flow behavior (Metrics 2, 7 in Table I) and memory behavior (Metrics 8-14 in Table I) contribute the most loading factors. Given this, we use PCA again on control flow and memory related metrics separately. This gives us two sets of PRCOs, out of which we pick the top two from each set. This allows us to plot all the benchmarks on two graphs and compare their control flow and memory behavior, shown in Figures 5 and 6 respectively. Since only two metrics contribute to control flow behavior, the loading factors of both metrics is the same for the two PRCOs. For memory behavior, PRCO1 is dominated by LLC misses and D-TLB misses and PRCO2 is dominated by I-cache misses and I-TLB misses.

The two figures show that the .NET, ASP.NET and SPEC CPU17 benchmarks exhibit different architectural behavior since the data points corresponding to their performance characteristics do not coincide. The .NET and ASP.NET benchmarks differ significantly from SPEC CPU17 in both control flow and memory behavior to merit consideration in architecture research which is often dominated by SPEC. The standard variation of SPEC CPU17 programs is $5.73\times$ and $4.73\times$ that of the .NET and ASP.NET respectively for control flow behavior, and $1.71\times$ and $1.27\times$ respectively for memory related metrics. This indicates that SPEC CPU17 includes a wider variety of benchmarks.

We see that the control flow behavior for ASP.NET and .NET applications is similar, which can be attributed to the large share of CLR code in both. This also explains the



(a) Control flow behavior. (b) Memory behavior. (c) Runtime behavior.

Fig. 7: Comparison between x86-64 and AArch64.

lower variance between .NET and ASP.NET applications. In traditional C/C++ workloads, memory management and the nature of the compiled code are highly source code dependent which leads to greater diversity in architectural behavior. It is worth noting that this generally makes blanket compiler-level optimizations more beneficial in managed languages.

D. Comparison between x86-64 and Arm

Since Arm server processors are more recent entrants, they are not as mature as Intel processors. Nonetheless, we briefly compare performance of .NET microbenchmarks on the two architectures to identify possible areas Arm processors could focus on for improvement. As before, we compare control flow (Metrics 2, 7) and memory behavior (Metrics 8-14). We also compare performance of runtime management (Metrics 19-23).

Figure 7 shows the comparison between x86-64 and Arm platforms. For control flow behavior, the PRCO loading factors are the same since only 2 metrics are included. The standard deviation among benchmarks on the Arm platform is $1.36\times$ that of the x86-64 platform for PRCO1, and $1.20\times$ for PRCO2. For the memory behavior, PRCO1 is dominated by L1, L2 Cache misses and I-TLB misses, and PRCO2 is dominated by LLC misses and D-TLB misses. The standard deviation among benchmarks on the Arm platform is $1.19\times$ that of the x86_64 platform for PRCO1, and $2.32\times$ for PRCO2. We also use PCA for runtime events. PRCO1 is dominated by GC and JIT events, and PRCO2 by exception and contention events. The standard deviation among benchmarks on the Arm platform is $1.02\times$ of the x86-64 platform for PRCO1, while it is for $0.58\times$ for PRCO2.

In summary, the x86-64 platform shows more variance among benchmarks on exception and contention events, while Arm shows much more variance on LLC misses and D-TLB misses. Comparing raw performance of the x86-64 and Arm machines reveals that Arm does $80\times$ worse on I-TLB MPKI and $8\times$ worse on LLC-MPKI. Such a large disparity cannot be only due to microarchitecture differences. Differences in the software stack also contribute to this. The Intel stack, including the runtime and the compiler, have undergone years of cross-stack optimizations, unlike Arm. We have highlighted the variance within the workloads and the differences in the performance profiles of the .NET applications on the two architectures in this work. Teasing out more differences requires further cross-stack analysis.

E. Performance Counters

Figure 8 plots some basic characteristics of the three benchmark suites on x86-64. Data from performance counters shows that in general, the instruction memory interface performs poorly on ASP.NET and .NET, with higher I-TLB and L1 I-cache MPKIs as compared to SPEC. Although SPEC shows diverse branch MPKIs because the underlying programs are more complex, ASP.NET programs have slightly higher branch MPKIs than several SPEC programs. ASP.NET programs also have significantly higher CPI than SPEC due to larger total frontend and backend stalls (see §VI). ASP.NET programs are more realistic than the .NET microbenchmarks in general, and hence show comparatively higher MPKIs and CPI. Nonetheless, System.Net, System.Threading, System.Diagnostics, and CscBench are also realistic and exhibit behavior similar to ASP.NET. These performance artefacts in ASP.NET and .NET arise from their large CLR code footprint and JIT events. After JITing, code pages are given new addresses, leading to branch predictor cold starts and I-cache/I-TLB/branch misses.

Additionally, compared to SPEC, ASP.NET programs have lower L1 D-cache MPKI, with GM of 15.9 vs 29 for SPEC, larger L2 MPKI (GM of 20.4 vs 11 for SPEC), but lower LLC MPKI (GM of 0.16 vs 0.98 for SPEC). The .NET microbenchmarks have much lower MPKIs (GM of 2.3, 2.2, and 0.01, respectively). We study the impact of these cache and branch MPKIs in the following sections.

VI. Top-down Analysis

Top-Down characterization is a hierarchical organization of event-based metrics that identifies the dominant performance bottlenecks in an application [42]. Its aim is to show, on average, how well the CPU's pipeline is being utilized while running an application.

Top-Down divides the processor pipeline into slots which can either be filled with instructions or are empty. Empty slots can be attributed to various bottlenecks within the pipeline. The ratio of empty slots with respect to the total number of available slots is calculated, which determines how much each bottleneck affects processor performance.

We use the Top-Down methodology in addition to examining raw metrics such as cache MPKI values as we want to measure their impact on actual processor performance. For example, a D-cache miss typically only decreases performance when it results in a full window stall in the reorder buffer. Only looking at the D-cache MPKI provides an incomplete picture

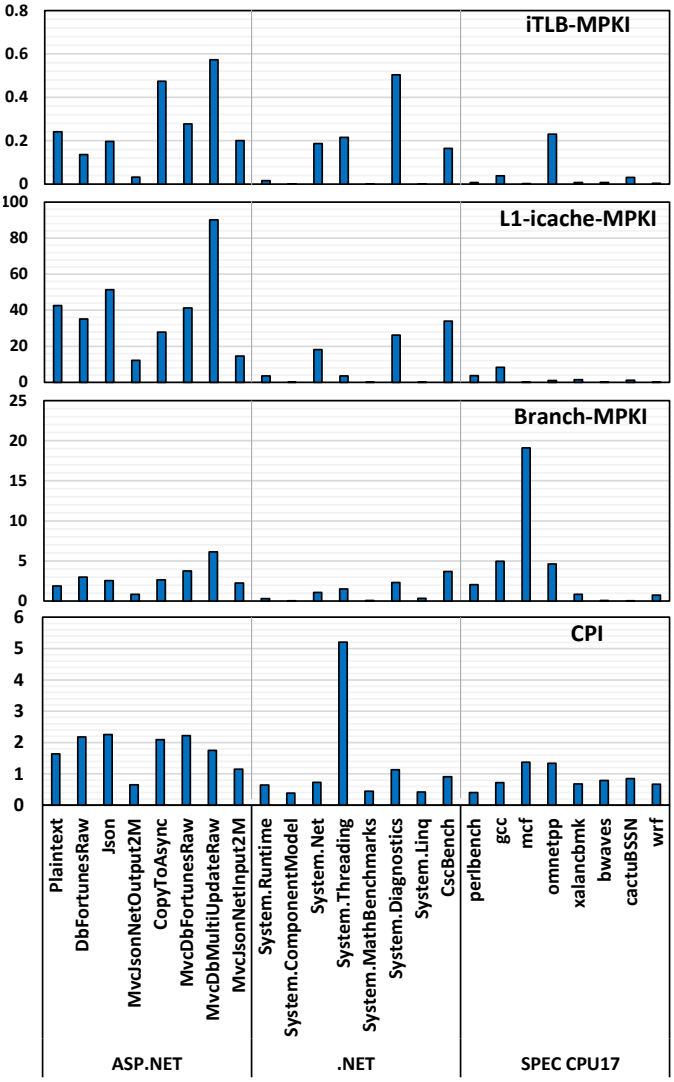


Fig. 8: Performance counter comparisons on x86-64.

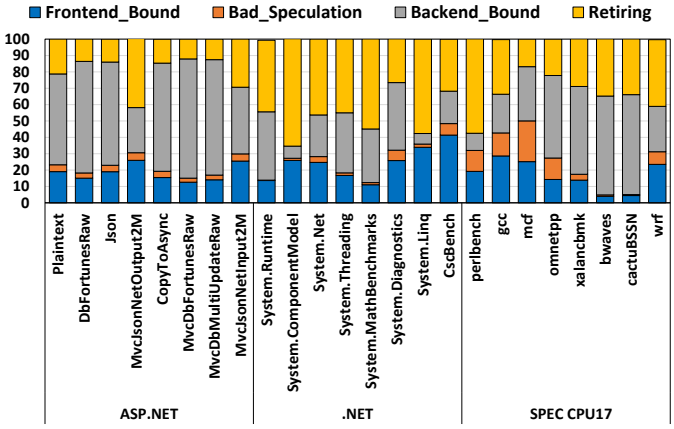


Fig. 9: Basic Top-Down profile for all benchmarks. Bars show the percentage of pipeline slots allocated to each bottleneck, broadly divided into the categories in the legend. Retiring indicates no bottlenecks.

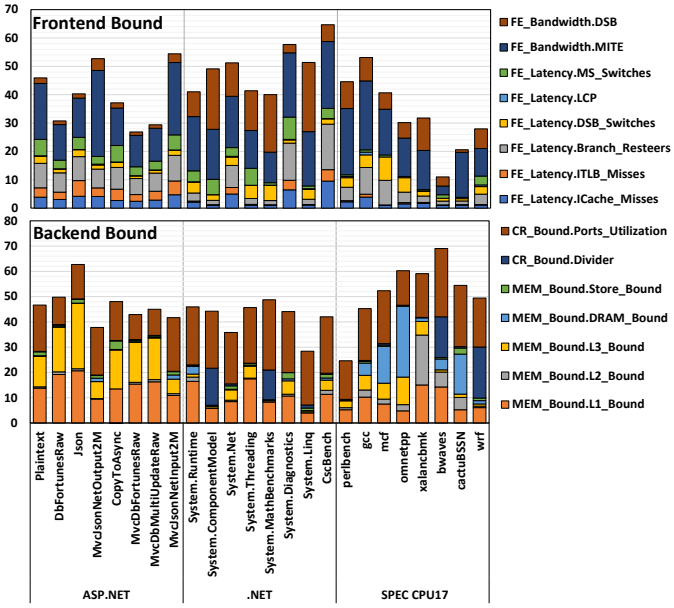


Fig. 10: Breakdown of empty pipeline slots in the Frontend and Backend. Bars show the distribution of empty slots attributed to the bottlenecks listed in the legend. FE = Frontend, CR = Core, MEM = Memory.

of its impact on performance. Top-Down uses performance counters available on Intel processors that measure empty pipeline slots and stall cycles at various pipeline structures, thus giving us a better picture of the impact of such raw metrics on performance.

We use *toplev*, which is part of the *pmu-tools* repository [15], for parsing performance counter values and obtaining a detailed Top-Down profile for each benchmark. The tools outline various bottlenecks; we explain what each metric represents in the following discussion.

A. Basic Top-Down Analysis

The observations from the basic breakdown in Figure 9 can be summarized as follows:

- ASP.NET is significantly backend bound, more so than the other benchmarks. This is in line with the perception that datacenter applications tend to be backend bound.
- Neither .NET nor ASP.NET have a significant bad-speculation component which is caused due to mispredictions-related pipeline flushes. The SPEC suite has a wider spread of branch frequency, branch MPKIs, and control flow complexity, and hence have a broader spread of related stalls than ASP.NET and .NET programs. In contrast to SPEC, kernel and the JIT compiler also contribute significantly to the branches in ASP.NET and .NET apart from the user code.
- Some .NET and ASP.NET applications have a significant frontend bound component which we explore in detail in the next section.

B. In-depth Analysis of Pipeline Bottlenecks

1) Frontend Bound: Figure 10 (top) shows a detailed breakdown of percentage of empty pipeline slots in the frontend. Note that percentages of less than 5% can be inaccurate due to

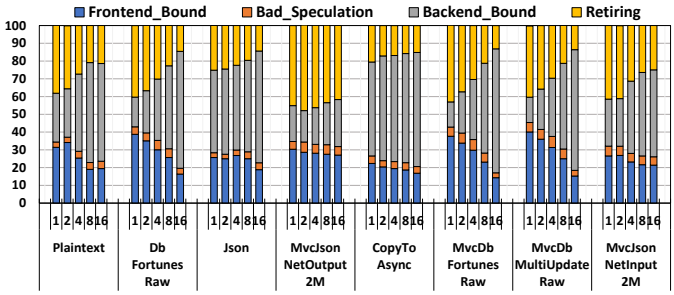


Fig. 11: Top-Down profile for ASP.NET applications running on 1, 2, 4, 8, 16 cores.

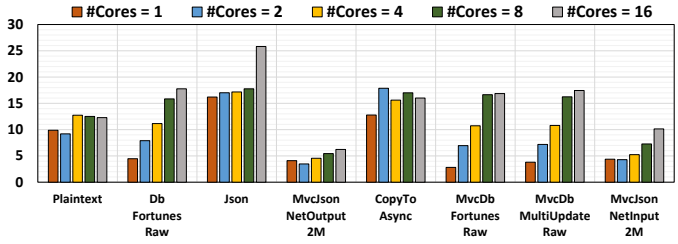


Fig. 12: Percentage of L3-Bound stalls for ASP.NET applications running on 1, 2, 4, 8, 16 cores.

measurement errors as stated by the tool. A significant fraction of the these come from DSB (Decode Stream Buffer) and MITE (legacy decoder) bandwidth. This is primarily attributed to less than peak fetch bandwidth from either the DSB due to structural constraints, or the standard pipeline due to lack of enough decoders. Another major reason for lost pipeline slots in both categories is packet breaks: the inability of the frontend to process multiple instructions of a certain kind in a single cycle. For example, packet breaks due to more than one taken branch prevent the frontend from filling all the fetch slots.

The frontend latency-bound metrics cover pipeline slots that are empty due to frontend stalls. The main sources of these stalls are BTB misses (branch re-steers), I-TLB misses, and I-cache misses. All three are large for most .NET and ASP.NET benchmarks, due to the large code base associated with these applications and frequent JIT events. While the I-cache MPKI in these benchmarks is high, a lot of the stalls due to them are hidden by backend stalls (e.g., due to memory accesses) which reduces their impact on performance. High (Microcode sequencer) MS-switches indicates a large number of microcoded instructions that result in empty pipeline slots since it takes multiple cycles to access the microcode ROM. These are likely due to the CLR code since they are consistent across most ASP.NET and .NET benchmarks.

2) Backend Bound: Figure 10 (bottom) shows a detailed breakdown of empty pipeline slots in the backend. The most interesting observation here is that the ASP.NET programs are L3 bound, which is the Last Level Cache (LLC) in the system. This means that a large fraction of empty slots in the pipeline are due to LLC misses. However, we observed that the per-core LLC-MPKI for the ASP.NET benchmarks is actually lower compared to the rest. This indicates that the stalls associated with the LLC are likely because of increased access latencies. To investigate this further, we studied how the bottlenecks in ASP.NET scale with core count.

Figure 11 shows that as the number of cores increases, most benchmarks are more backend bound. Upon further analysis, we found that this is primarily due to an increase in L3-bound stalls, as shown in Figure 12, while the per-core LLC-MPKI remains relatively stable. This supports our earlier claim that ASP.NET benchmarks see increased LLC access latencies. Since the access latency increases as the application scales, this can be attributed to contention at the ports of individual LLC slices. ASP.NET runs on a large number of cores and therefore ends up generating a significant amount of on-chip traffic. Thus the increased LLC latency could also be a result of contention in the Network-on-Chip (NoC).

One other interesting observation is that a significant percentage of empty slots are due to D-cache latency (˜4 cycles for modern L1 D-caches) in ASP.NET and select .NET benchmarks. This indicates there are a large number of simultaneous requests to the D-cache that are hits which saturate the D-cache bandwidth in these applications. SPEC CPU17 is more DRAM bound. Since the .NET and ASP.NET applications do not have a large working set (all under 500MiB), they do not exercise memory as much as SPEC CPU17, which can have large working sets (up to 16GB for some applications).

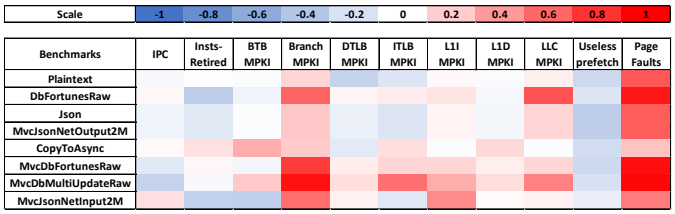
Port utilization (Figure 10, bottom) represents the percentage of empty pipeline slots due to micro-ops not being issued to execution ports at peak bandwidth when micro-ops are present in the reservation stations. This therefore includes stalls due to lack of intrinsic Instruction Level Parallelism (ILP) within the program, which are not caused by microarchitecture constraints but are captured in this metric. Applications that use the core divider extensively have a small percentage of empty slots allocated to them since the divider functional unit is normally non-pipelined and takes multiple cycles.

VII. Analysing the Managed Runtime

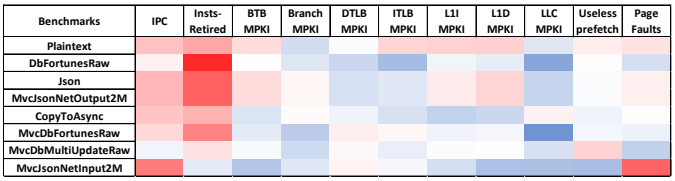
To gain insights into how the managed runtime affects application performance, we collected traces for several runtime events using *LTTng* [5] for the ASP.NET benchmarks. Since most of the .NET microbenchmarks were extremely short, obtaining useful traces for them was difficult. Hence we only performed basic GC studies on these benchmarks.

A. Correlation of Hardware Performance Counters with Runtime Events for ASP.NET

The runtime event traces for ASP.NET were collected in the form of samples over the period of execution of each benchmark along with corresponding samples for performance counters. Each sample was associated with a timestamp with a sampling interval of 1 millisecond. We then calculated the correlation coefficient (Pearson's correlation) between the runtime event samples and performance counters. Figures 13a and 13b summarize some of the data for the ASP.NET benchmarks. The correlation coefficient by itself does not indicate whether runtime events cause the change in the performance counters.



(a) JIT-start events.



(b) GC invocations.

Fig. 13: Correlation of JIT-start events and GC invocations in ASP.NET with a few performance counter values.

We manually examined the sample timestamps and confirmed that changes in the performance counter values were observed after changes in the JIT and GC event samples. The delay between the runtime events and the change in performance counters ranged from 10 microseconds to 5 milliseconds and was consistent across multiple runs. We therefore surmise that the observed runtime events were primarily responsible for the changes in the performance counters .

For the JIT correlation studies, we increased the heap size to maximum. This was done to reduce GC events in the program and hence their effect on the performance counters. Likewise, for the GC studies we used a small heap size to increase the number of GC events in the program and highlight their effect on the performance counters.

1) JIT events: The positive correlation coefficients observed between JIT-start events and branch MPKI, LLC-MPKI and page faults indicate that JIT events cause an increase, 5%-20%, in these metrics. We also observed a minor increase ($\sim$5%) in the L1 I-cache MPKI. These can be attributed to cold starts. After JIT compilation, code pages are assigned new addresses. These new pages always miss in caches as traditional prefetchers do not issue requests beyond the page boundary, causing an increase in cache MPKI and page faults. (ASP.NET has $\sim$300$\times$ as many page faults as SPEC.)

Note that we also see a negative correlation with respect to the number of useless prefetches which indicates that data within the JITed pages is actually prefetchable. However the large increase in LLC MPKI suggests that the prefetches are not aggressive enough.

Branch MPKI is also affected by cold starts. The branch predictor state is stored in tables indexed by the program counter (PC). Since JITing a code page changes the branch addresses, the predictor state is lost even if the control flow behavior of those branches is unchanged. This results in a low prediction accuracy immediately after a code page is JITed as additional time is needed to retrain the branch predictor. This effect is not limited to branch predictors as a number of micro-architecture structures use PC-indexed meta-data to improve performance. While the effect on branch prediction was prominent for ASP.NET, larger real-world applications may exhibit similar issues with other structures such as PC-based prefetchers.

While the runtime JIT compiler contributes to some of the above misses when invoked, the sampled performance counters indicate that a large proportion of the increase in the performance counter values was due to the generated JITed code pages.

We find that JIT compilation has many trade-offs associated with the micro-architecture. It is challenging to solve the above problems only in the runtime or the compiler as this would require detailed knowledge of the underlying micro-architecture. Further, additional code added to the software without ISA changes can also increase code size, further exacerbating the problem. Cross-stack solutions that involve hardware changes are therefore important to address these issues. For example, hooks in the ISA can be used by software to provide meta-data regarding JITed code pages to the hardware. This can help improve prefetching for these pages. The meta-data can also be used to either preserve or transform the microarchitectural state of the machine (such as branch predictor tables) related to these pages to reduce the effect of cold starts.

2) Garbage Collection events: Figure 13b shows that GC events improve LLC MPKI, perhaps counter-intuitively. This is likely due to better locality in caches after dead objects are removed and data is rearranged (heap compaction). While GC events also results in a large amount of data movement, the benefits of rearranging data surprisingly leads to an overall decrease in the LLC MPKI (of $\sim$8%). We see an increase in instruction footprint due to GC events - an overhead that is already well explored, but the overall performance (IPC) is positively correlated with GC events. Since the ASP.NET programs we use do not have very large working sets (all under 500MiB), the GC is not invoked often. The benefits of rearranging data in the caches out-weights the overhead of the additional code executed, and hence we see that IPC improves after GC events. While GC overheads may be larger in real-world applications with big working sets, rearrangement of data in caches after GC events still improves cache locality.

Hardware acceleration of GC is therefore useful not only because it helps alleviate the potentially large overheads of GC (as shown in prior work [31], [33]) but also because it can improve cache performance. In fact, even limited GC acceleration in hardware can potentially reap the benefits of greater locality as it does not incur the overhead of frequent GC events as compared to previously proposed software-only solutions [23], [24].

This provides an interesting insight into memory management which is currently performed exclusively in software. The hardware has detailed information about the cache hierarchy: cache sizes, replacement policies etc. If a part of the memory management (GC in managed languages) is offloaded to the hardware, it can potentially improve performance through aggressive dead block elimination and better mapping of data to cache blocks. While this idea is applicable to all programs

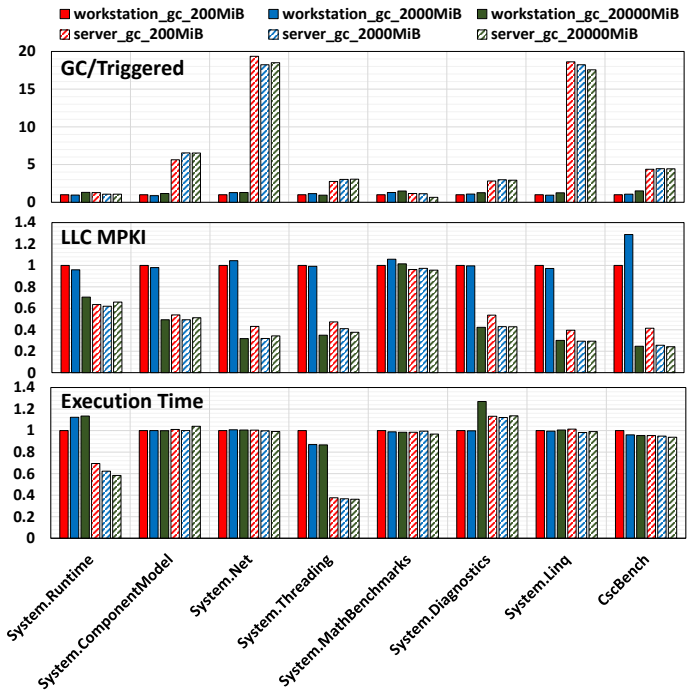


Fig. 14: Comparison between different GCs. Data is normalized to workstation_gc_200MiB.

include those without explicit garbage collection, managed languages are prime candidates for such optimizations since the meta-data and software interfaces required for offloading memory management to the hardware are much more readily available in these frameworks. For instance, the concept of GC generations used by managed language to separate short-lived and long-lived objects can be leveraged to place data efficiently in the memory hierarchy.

B. GC Analysis for .NET Microbenchmarks

To understand the effect of GC on the .NET microbenchmarks, we used aggregate performance counter and runtime event values. The .NET kernel provides two Garbage Collection mechanisms, workstation GC and server GC. Workstation GC is designed for client apps. It runs as a user thread and competes with applications running on the core for CPU time. Server GC runs on multiple dedicated threads at high priority and is more resource intensive. Server GC is designed for datacenter applications that require high throughput and scalability and has larger overheads [6].

In our experiments, we run the .NET microbenchmarks with these two GCs and three different maximum heap sizes, 200MiB, $2,000$MiB, and $20,000$MiB to gain insights into how GC aggressiveness affects the microarchitecture. System.Collections cannot be complied with workstation GC and 200MiB maximum heap size due to an OutOfMemory Exception. System.Text, System.Collections, and System.Tests, cannot be compiled with server GC and 200MiB maximum heap size as server GC requires a larger minimum memory size for these applications.

GC/Triggered, LLC-MPKI and execution time are the three metrics that are affected the most under different GC settings and are shown in Figure 14. Since server GC is more aggressive, the GC is triggered $6.18\times$ more often than when using workstation GC on an average. Server GC achieves a $0.59\times$ reduction in LLC-MPKI compared to workstation GC. Applications running with server GC run $1.14\times$ faster than those using workstation GC indicating that the benefit of reduced cache MPKI outweighs the overhead of executing additional GC code for most of the benchmarks as they do not have a large working set. Few workloads like System.MathBenchmarks which exhibit very little cache activity perform worse with server GC due to the additional overhead. These results are in line with those we observed for ASP.NET.

VIII. Conclusion

Through our analysis of .NET benchmarks, we uncovered many micro-architectural bottlenecks in these applications as outlined in §VI, many of which have not been explored before in the context of managed language workloads. We showed that they are significantly different from the standard SPEC CPU17 suite in frontend, backend and CPI behavior. Hence, we argue that they should be included in architecture studies, particularly when designing the new Arm server processors.

In §VII, we provided several interesting insights about the impact of runtime events on the core micro-architecture. They reveal key opportunities for performance optimization of managed languages in hardware, some of which we summarize below:

- Improving the performance of the networking stack to speed up datacenter applications.
- Better management of meta-data in frontend structures such as the I-TLB, BTB and I-cache for applications with large and diverse code footprints.
- Data placement strategies in LLC slices to reduce contention at the NoC.
- Aggressive prefetching and software-driven hardware transformations for structures in the microarchitecture to make them more JIT-friendly.
- Offloading a part of Garbage Collection to hardware for improved cache performance while keeping the overhead of memory management low.

A key theme in many of these opportunities is hardware solutions that are assisted by meta-data provided by the software. This can take the form of new instructions in the ISA to manipulate already present hardware structures or programming accelerators at runtime to perform specific tasks. Managed runtimes already provide a large amount of meta-data that can be leveraged to implement such hardware solutions without needing to involve the programmer. The .NET CLR, for instance, contains meta-data about live objects and their references which can be communicated to the hardware.

Performance optimizations that span the stack are gaining popularity as it is becoming increasingly difficult to improve CPU performance only through technology improvements. Managed language workloads are becoming increasingly important, especially in datacenters, and provide fertile ground for cross-stack optimizations. Research in architecture and

microarchitecture optimization for managed languages is likely to grow further in importance with growing popularity of managed languages. Further studies of more diverse .NET programs and full-system analysis are needed.

IX. Acknowledgment

We thank Andy Ayers, Sébastien Ros, David Levinthal, Shyam Murthy, Tanvir Ahmed Khan, and anonymous reviewers for insightful discussions and comments about the work.

References

[1] "Azure Cosmos DB documentation". https://docs.microsoft.com/en-us/azure/cosmos-db.

[2] "Azure SQL Database documentation". https://docs.microsoft.com/en-us/azure/azure-sql/database/connect-query-dotnet-core.

[3] "Bing.com runs on .NET Core 2.1!". https://devblogs.microsoft.com/dotnet/bing-com-runs-on-net-core-2-1.

[4] "Linux perf tool.". https://perf.wiki.kernel.org/index.php/Main_Page.

[5] "LTTng.". (https://lttng.org/).

[6] "Microsoft .NET Garbage Collection Document". https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/rks.

[7] ".NET applications on Azure". https://azure.microsoft.com/en-us/develop/net.

[8] ".NET Core 3.1". https://dotnet.microsoft.com/download/dotnet-core/3.1.

[9] ".NET Core is 'Most Loved' Framework in Stack Overflow Survey". https://visualstudiomagazine.com/articles/2019/04/09/so-survey.aspx.

[10] "Reason Why .Net Framework is the Most desirable Framework in 2020!". https://tinyurl.com/y2ecyetx.

[11] "SPEC CPU2017". https://www.spec.org/cpu2017.

[12] "SPECjvm2008". https://spec.org/jvm2008/.

[13] "State of the Developer Nation 19th Edition - Q3 2020". https://www.developereconomics.com/resources/reports/state-of-the-developer-nation-q3-20201.

[14] "TechEmpower Web Framework Benchmarks.". (https://www.techempower.com/benchmarks/).

[15] "Toplev tool.". (https://github.com/andikleen/pmu-tools/wiki/toplev-manual).

[16] "What are "SPECspeed" and "SPECrate" metrics?". https://www.spec.org/cpu2017/Docs/overview.html#metrics.

[17] "What is ASP.NET?". https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet.

[18] "What is .NET?". https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet.

[19] ".NET Performance". https://github.com/dotnet/performance, commit c86ef708.

[20] "ASP.NET Benchmarks". https://github.com/aspnet/Benchmarks, commit fa417157.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[22] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 169–190, 2006.

[23] Trishul M. Chilimbi and James R. Larus. "Using Generational Garbage Collection to Implement Cache-Conscious Data Placement". In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, page 37–48, 1998.

[24] Trishul M Chilimbi and James R Larus. "Data Structure Partitioning with Garbage Collection to Optimize Cache Utilization", November 20 2001. US Patent 6,321,240.

[25] Andy Clark. "Stack Overflow Migrate Architecture from .NET Framework to .NET Core". https://www.infoq.com/news/2020/04/Stack-Overflow-New-Architecture/.

[26] Mathieu Desnoyers and Michel Dagenais. "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux". *OLS (Ottawa Linux Symposium)*, 01 2006.

[27] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. "Predicting Inter-thread Cache Contention on a Chip Multi-processor Architecture". In *11th International Symposium on High-Performance Computer Architecture*, pages 340–351, 2005.

[28] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware". In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 37–48, 2012.

[29] Robert Gawlik and Thorsten Holz. "SoK: Make JIT-Spray Great Again". In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, August 2018. USENIX Association.

[30] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C Steely Jr, and Joel Emer. "Achieving Non-inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies". In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–162. IEEE, 2010.

[31] José A. Joao, Onur Mutlu, and Yale N. Patt. "Flexible Reference-Counting-Based Hardware Acceleration for Garbage Collection". In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 418–428, 2009.

[32] A. Limaye and T. Adegbija. "A Workload Characterization of the SPEC CPU2017 Benchmark Suite". In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158, 2018.

[33] M. Maas, K. Asanović, and J. Kubiatowicz. "A Hardware Accelerator for Tracing Garbage Collection". In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 138–151, 2018.

[34] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. "Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?". In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282. IEEE, 2018.

[35] Aashish Phansalkar, Ajay Joshi, and Lizy K John. "Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite". In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 412–423, 2007.

[36] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, et al. "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–47, 2019.

[37] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. "MLPerf Inference Benchmark". In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020.

[38] Mitsuhisa Sato. "The Supercomputer "Fugaku" and Arm-SVE Enabled A64FX Processor for Energy-efficiency and Sustained Application Performance". In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 1–5. IEEE, 2020.

[39] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. "Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine". In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, page 657–676, 2011.

[40] Livio Soares and Michael Stumm. "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls". In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 33–46, 2010.

[41] Svante Wold, Kim Esbensen, and Paul Geladi. "Principal Component Analysis". *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[42] A. Yasin. "A Top-Down Method for Performance Analysis and Counters Architecture". In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.