Leveraging Approximation to Improve Datacenter Resource Efficiency

Neeraj Kulkarni, Feng Qi and Christina Delimitrou
Cornell University
{nsk49,fq26,delimitrou}@cornell.edu

Abstract—Cloud multi-tenancy is typically constrained to a single interactive service colocated with one or more batch, low-priority services, whose performance can be sacrificed. Approximate computing applications offer the opportunity to enable tighter colocation among multiple applications whose performance is important. We present Pliant, a lightweight cloud runtime that leverages the ability of approximate computing applications to tolerate some loss in output quality to boost the utilization of shared servers. During periods of high contention, Pliant employs incremental and interference-aware approximation to reduce interference in shared resources. We evaluate Pliant across different approximate applications, and show that it preserves QoS for all co-scheduled workloads, while incurring at most a 5% loss in output quality.

Index Terms—Super (very large) computers, Scheduling and task partitioning, Support for dynamic compilation

1 Introduction

Cloud computing has reached proliferation by offering *resource flexibility* and *cost efficiency* [3]. Cost efficiency is achieved through multi-tenancy, i.e., by co-scheduling multiple jobs on the same physical platform. Unfortunately multi-tenancy also leads to unpredictable performance due to interference [6]. When the applications that suffer from interference are high priority and interactive services, multi-tenancy is disallowed, hurting utilization, or interactive services are co-scheduled with low priority, best-effort workloads. The performance of these workloads can be sacrificed at any point to avoid performance degradation for the high priority service [7, 12, 19]. Unfortunately this limits the colocation options cloud providers have. Approximate computing offers the potential to break this utilization versus performance trade-off in shared clouds.

Approximate computing applications include workloads from several fields, such as computer vision, machine learning, analytics, and scientific applications, and have the common feature that they can tolerate some loss in output accuracy in return for improved performance and/or energy efficiency [5, 9, 14, 15, 18]. Several cloud workloads fall under this category, such as ML analytics, where achieving the highest output quality is often less important than latency and/or throughput. Exposing the knob of approximation to the cloud scheduler allows the system to sacrifice some accuracy to preserve the services' quality-of-service (QoS) constraints.

We present Pliant, a cloud runtime system that achieves both high QoS and high utilization by leveraging the ability of approximate computing applications to tolerate some loss in their output quality. Unlike prior cluster schedulers, Pliant does not directly sacrifice the performance of applications co-scheduled with interactive services. Instead, a user expresses a tolerable approximation threshold to the scheduler, and Pliant dynamically adjusts the level of approximation to the minimum amount needed to satisfy the tail latency QoS of the interactive service, without exceeding this inaccuracy threshold.

Pliant consists of a lightweight performance monitor and a dynamic compilation system. The monitor uses adaptive sampling to continuously check for QoS violations, while the compilation tool is based on DynamoRIO [2], and adjusts the degree of approximation online. When the interactive service violates its QoS, Pliant reclaims cores from the approximate computing applications, yields them to the interactive service, and adjusts the approximation degree to ensure the execution time of the approximate application does not degrade. Pliant reclaims cores incrementally to guarantee that the approximate application only sacrifices the minimum amount of accuracy needed.

We evaluate Pliant on server platforms with 44 physical (88 logical) cores, with memcached, a distributed in-memory caching service [1], as the interactive application, and a set of scientific workloads from PARSEC and SPLASH2 [4, 20] as the approximate applications. We show that Pliant is able to preserve both the tail latency QoS of the interactive service, and the nominal execution time of the approximate applications, with a 2.8% loss of output quality on average, and 5% loss in the worst case. In comparison, running the applications in precise mode results in a 2-3x increase in the tail latency of memcached, a dramatic degradation for latency-sensitive, interactive services. Finally, we explore the sensitivity of Pliant to the granularity of decisions, across load levels.

2 Related Work

We now review relevant work in interference-aware scheduling, approximate computing, and dynamic instrumentation.

Contention-aware scheduling: Sharing cloud resources results in performance degradation [6, 7, 16], and in some cases security vulnerabilities [8]. Several systems disallow colocation of interfering jobs to begin with [6, 7, 16], or partition resources to improve isolation [8, 10, 11, 13]. For example, DeepDive identifies the interference colocated VMs experience, and manages it transparently to the user [16]. Paragon [6] and Quasar [7] are cluster managers that leverage practical online data mining techniques to determine the resource requirements of incoming cloud applications, and schedule them in a way that minimizes contention. On the isolation front, Lo et al. [13] study the sensitivity of Google applications to different sources of interference, and combine hardware and software isolation techniques to preserve QoS for interactive applications. Similarly Kasture et al. implement fine-grain cache partitioning [11] and power allocation with RAPL [10] to guarantee QoS for latency-critical services. In all cases, a server hosts at most one high-priority interactive application; any remaining workloads are best effort, and their performance can be sacrificed when needed.

Dynamic recompilation: Open-source tools like DynamoRIO [2] enable online code transformations which can be used to reduce the contention the instrumented application incurs in a multi-tenant system. For example, Protean Code [12] is a co-designed compiler and runtime built over LLVM that enables code transformations with less than 1% overhead. The runtime is then used to dynamically mitigate cache pressure by disabling prefetching for low-priority applications during periods of high resource contention.

Software approximate computing techniques: Finding the approximation potential of popular application classes, and generating language constructs to express and verify approximation has generated a large amount of related work. Carbin et al. [5] present constructs for specifying acceptability properties in approximate programs. Sampson et al. [18] propose annotating data types that can be approximated, and automatically mapping such variables to approximate storage that uses low-power operations. The same authors develop ACCEPT [17], a programmer-guided compiler framework that identifies approximable code, and automatically chooses the best approximation strategies. Finally, Misailovic et.al [15] present Chisel, an optimization framework that automatically generates instructions and data that can be stored in approximate memory to improve efficiency, and compiler-level transformations that automatically generate approximate versions of applications [14].

Manuscript submitted: 27-Mar-2018. Manuscript accepted: 27-May-2018. Final manuscript received: 06-Jun-2018.

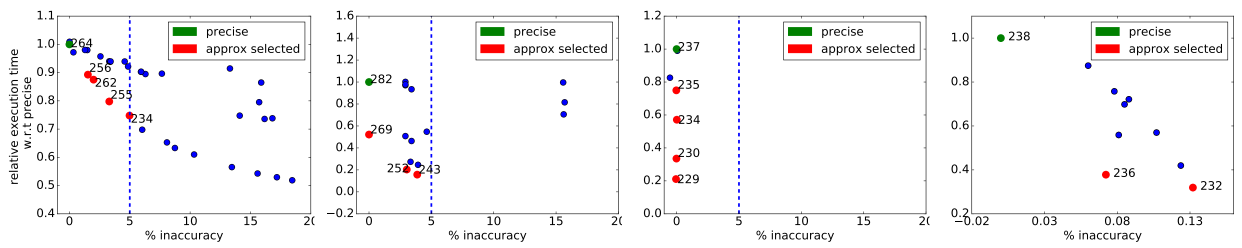

Fig. 1: Approximate design space exploration for `canneal`, `streamcluster`, `water_nqsquared`, `raytrace`.

3 Pliant

3.1 Overview

Pliant exposes approximation to the cluster scheduler, which the latter uses to ensure interactive services meet their QoS during periods of high resource contention. The interface between an application and Pliant involves expressing an interactive service's QoS, and an approximate application's nominal execution time, and tolerable quality loss. Having this information, the runtime can dynamically determine the type and degree of approximation needed for QoS to be met. Approximation can be used to restore an interactive service's performance in two ways. First, approximation can be used to directly reduce the interference generated by the approximate application. In this case, the runtime employs one or more approximation techniques that reduce contention, without hurting the execution time of the approximate workload. Second, approximation can be used as a means to improve the performance of the approximate application when its resources have been reclaimed and given to the interactive service.

Pliant consists of two components, the *performance monitor* and the *actuator*. Both components are designed to *incur minimal runtime overheads*, be *transparent to the user*, and *assign high priority to the performance of both interactive and approximate applications*.

The *performance monitor* is a lightweight tracing module that instruments the interactive applications, and continuously samples their latency (average and tail). Since QoS reflects the end-to-end latency of a service, the monitor resides at the entry point to the datacenter, and is designed to not incur any measurable performance overhead, either in terms of throughput or latency. Upon detecting a QoS violation for the interactive workload, the monitor informs Pliant, which takes action via its *actuator* module. The actuator is responsible for i) determining an appropriate approximation variant and resource allocation based on the monitored tail latency, and ii) enforcing the chosen degree of approximation. Below, we first discuss how approximation variants are chosen, and then how dynamic recompilation works in Pliant.

3.2 Approximation Design Space Exploration

We explore several approximation strategies, such as loop perforation, synchronization elision, and low precision data types.

- **Loop Perforation:** This technique omits a fraction of iterations in a loop. Typical approximate computing applications, like analytics, machine learning, and scientific workloads are iterative in nature, making loop perforation a good candidate for approximation. There are multiple ways to perforate a loop. For example, to reduce a loop by a factor p, we can a) execute only (MAX_ITER/p) consecutive iterations, or b) execute every p^{th} iteration. We can also reduce the loop by a factor $(p-1)/p$ by not executing every p^{th} iteration. We explore each of these strategies.
- **Synchronization Elision:** Synchronization constructs, like locks and barriers, can be elided. Removing locks reduces the memory traffic that acquiring locks incurs, which can be significant for highly contended locks. Apart from memory traffic, synchronization elision also benefits performance, as threads do not wait to synchronize, shortening execution time.
- **Lower Precision:** This technique replaces variables like "double" with lower precision types, such as "float" and "int", reducing memory traffic and execution time for some penalty in output quality.

Pruning the design space: We now study the relation between accuracy and execution time for approximate applications, and select approximate variants close to the pareto optimal curve. Typical applications have a large number of loops, multiple of which can be perforated. Considering all approximation possibilities makes the design space intractable, in the order of 1000s of approximation versions. We use two ways to prune this space. First, we employ an "almost" exhaustive exploration that leverages hints from the ACCEPT framework [17]. ACCEPT lists a maximum of 10 loops that can be perforated for each application. We perforate each loop by different factors and examine the resulting execution time and inaccuracy. We only preserve approximate versions with inaccuracies lower than 5%. Second, we use `gprof`, an application profiling tool, to determine which functions contribute the most to execution time. In all examined applications, there are 2-3 functions that dominate execution time. We perforate these functions to determine the potential of approximation on execution time and inaccuracy. This approach also resulted in a manageable number of favorable approximation variants, consistent with those obtained using ACCEPT. Figure 1 shows the inaccuracy-execution time for four representative applications from PARSEC and SPLASH-2. The blue dots represent the examined approximation variants, the green dot represents the precise version of the application, and the red dots represent approximation variants that reside close to the pareto-optimal frontier, and are hereafter used by Pliant. The number of selected approximate versions varies across applications. The marker labels in Fig. 1 correspond to the tail latency of an interactive service, `memcached` in this case, when it is co-scheduled with the different approximate and precise variants.

The approximate variants extracted from the design space exploration above are used to construct a single application binary. Different versions of functions that house the approximated primitives are incorporated in the original application source code, including one version that corresponds to precise execution.

3.3 Dynamic Recompilation

DynamoRIO, a dynamic recompilation tool is used to adjust an application's degree of approximation at runtime. The DynamoRIO API provides the ability to control applications at the granularity of individual instructions, as well as at a coarser granularity of functions. To avoid high instrumentation overheads, we use DynamoRIO to control applications at coarse granularity. DynamoRIO reads the program addresses of precise and approximate versions of each approximated function at the start of the program, and based on the output of the performance monitor, replaces the corresponding functions with

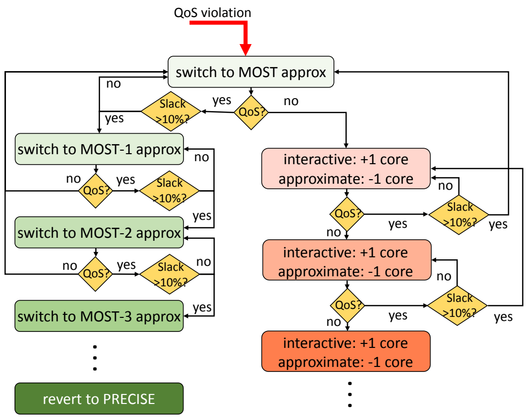

Fig. 2: Control flow of Pliant's runtime algorithm.

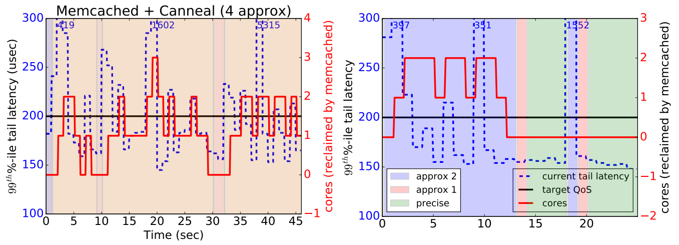

Fig. 3: Pliant's dynamic behavior when `memcached` is colocated with `canneal` (left) and `raytrace` (right). The left y-axes show memcached's tail latency over time, and the right y-axes the cores Pliant reclaims from the colocated application and yields to `memcached`.

the appropriate precise/approximate variants. When DynamoRIO is triggered (using a set of Linux signals), `drwrap_replace()` replaces the pointers to the original precise version of a function to the appropriate approximate version, using the program addresses read at start-up time. `drwrap_replace()` is also used to switch between approximate variants, or to revert back to precise execution.

Running an application over DynamoRIO can incur significant runtime overheads. We only use DynamoRIO at a coarse, function-level granularity to minimize the impact on application performance. We have quantified the overheads of dynamic recompilation and ensured they are below 8% in practice for all examined applications. In comparison, when fine-grained code transformations are necessary, there may be need for a new compiler [12].

3.4 Pliant Runtime Algorithm

Fig. 2 shows the control flow of Pliant's runtime algorithm. Initially, execution starts at precise mode, and in the event of a QoS violation Pliant switches the co-scheduled application to its most approximate version to avoid prolonged degraded performance for the interactive service. If QoS is met, Pliant examines the amount of latency slack the interactive service experiences. If slack exceeds 10% - set empirically - Pliant incrementally reverts back to less approximate versions, and potentially precise execution, to avoid unnecessarily penalizing the approximate application's output quality. If QoS is not met, Pliant additionally reclaims cores from the approximate application, one at a time until QoS is met. In that case approximation allows the co-scheduled workload to preserve its nominal execution time. If at a later point the interactive service has enough latency slack, these cores are returned to the approximate application, and its degree of approximation is reduced. The slack threshold is empirically set at 10% of QoS; lowering it results in frequent ping-ponging between approximate versions and higher overheads from DynamoRIO, while increasing it leads to resource underutilization by the interactive service. When multiple approximate applications are co-scheduled with an interactive service, approximation and core reclamation is applied via a round-robin arbiter to optimize for fairness. More sophisticated policies, including incorporating priorities between approximate applications, are deferred to future work.

4 Evaluation

4.1 Experimental Setup

Applications: We use memcached, an in-memory key-value store as the interactive service; the input load is generated using an open-loop workload generator. The QoS target of `memcached` is set at 200usec for the 99th percentile latency, consistent with prior work [7, 13]. Unless otherwise specified, we run `memcached` at high load, approximately 80-85% of saturation. We also use three workloads (fluidanimate, canneal, streamcluster) from the PARSEC benchmark suite, and three workloads (water_spatial, water_nsquared, raytrace) from the SPLASH2x suite as examples of approximate computing applications. The results are similar for the remaining workloads of both suites and are omitted due to space constraints.

Systems: We use a dual-socket, 44-physical core (88 logical core) platform, with 128GB of RAM as the server. To avoid NUMA effects, we only use one of the sockets for `memcached` and the approximate applications. `memcached` and the approximate workloads are instantiated in separate Docker containers, and pinned to different physical cores of the same socket. The containers share the 56MB last level cache (LLC), the main memory, and the NIC. An additional 6 physical cores are dedicated to network interrupts (`soft_irq`) to avoid interference with application threads. The remaining physical cores are fairly shared across the two Docker containers.

We first evaluate Pliant's dynamic behavior for two representative approximate computing applications, then show its performance and efficiency across all studied applications, and finally show the runtime's sensitivity to various configuration parameters.

4.2 Dynamic Behavior

Figures 3a and 3b show Pliant's dynamic behavior when `memcached` is colocated with `canneal` and `raytrace` respectively. Pliant uses a one second decision interval at the end of which, it makes a decision on the degree of approximation and core allocation for the two applications. In Section 4.4 we study the impact of varying the decision granularity on application performance. When a QoS violation for memcached is detected, the runtime switches the colocated application to its most approximate variant, and if that is not sufficient, it additionally reclaims physical cores and yields them to memcached, one core per decision interval. To avoid penalizing the approximate application when memcached's latency slack is high, Pliant also relinquishes cores from memcached when tail latency has $>10\%$ slack. Due to the high memory traffic `canneal` introduces, Pliant has to operate in its most approximate variant, and rely on core reclamation to meet QoS. On the other hand, `raytrace` generates high interference in some phases and almost negligible in others, allowing it to leverage both its precise operation and its multiple approximate versions over the duration of its execution.

4.3 Aggregate Results

We now evaluate Pliant for all examined applications. The decision interval is again one second. Figure 4a) shows the 99$^{th}-ile$ tail latency, execution time, and inaccuracy for the baseline system (precise), and Pliant. In the baseline, `memcached` and the precise version of the colocated application each receive a fair core allocation. In the case of Pliant, the initial allocation is fair, and the runtime adjusts it when needed online. Running in precise mode always results in severe QoS violations for `memcached`. Pliant meets `memcached`'s

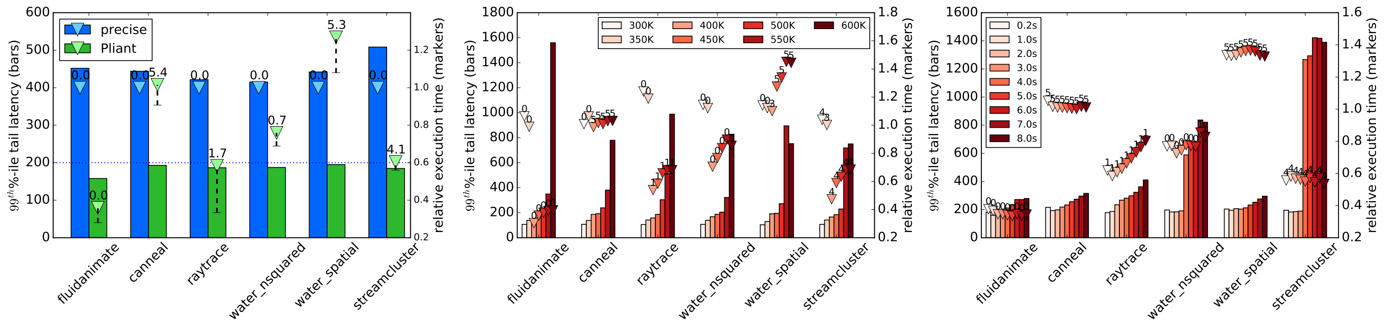

Fig. 4: **a) Comparison of Pliant against the baseline precise runtime, b) Performance of Pliant across QPS, c) Performance of Pliant for varying decision intervals across applications** The tail latency of `memcached` is shown in bars, while markers represent the execution time of approximate applications. The marker labels denote the % inaccuracy. The whiskers in a) indicate the overhead of DynamoRIO.

QoS for all colocations. Additionally, all approximate applications, except for water_spatial maintain their previous execution times, and in several cases shorten them. Their accuracy loss is also within the 5% tolerable limit. In the case of water_spatial, the selected approximate variants do not significantly reduce its execution time, and its reduced core allocation results in slightly higher execution time than in precise mode. water_spatial also experiences an unusually high instrumentation overhead from DynamoRIO (seen from the whisker in Fig 4a), which also adds to its execution time.

4.4 Sensitivity Experiments

Input load: Figure 4b shows tail latency for memcached, and execution time and inaccuracy for colocated workloads, as we vary memcached's input load (QPS). memcached's QoS is satisfied for QPS below 500K when co-located with any of the examined applications. For QPS below 350K, colocation does not impact tail latency enough to cause QoS violations, resulting in execution mostly in precise mode for the colocated workloads. As a result, there is zero accuracy loss for all applications except streamcluster (Fig 4b). For QPS above 400K, applications leverage approximation for the most part, and thus their execution time is lower. As QPS increases further, approximation alone is not enough to counteract the severely limited core allocation for the approximate applications, causing their execution time to increase.

Decision interval: Finally, we vary Pliant's decision interval (Fig. 4c). When the decision interval is too coarse (above one second), memcached experiences prolonged QoS violations until Pliant takes action. Decision intervals of 1 second or less always allow Pliant to satisfy memcached's QoS without penalizing the colocated application's execution time or accuracy. In theory, very short decision intervals can result in higher execution times for the approximate applications due to frequent switching between precise and approximate versions. In practice, due to the lightweight design of Pliant, no such degradation is observed. In fact in the case of `raytrace` there is the reverse trend; the application has higher execution time with longer decision intervals, because the runtime allows it to run with a smaller number of cores than needed for the interactive application to satisfy its QoS.

5 Conclusions

We presented Pliant, a lightweight runtime that leverages the ability of approximate applications to tolerate some loss of output quality, to preserve the QoS of co-scheduled interactive services. We demonstrated that approximation exposes a wide spectrum of operating points in terms of performance and inaccuracy, and showed that Pliant can navigate this space effectively and maintain the services' QoS, while using the lowest degree of approximation needed.

References

[1] "Distributed caching with memcached," in *Linux Journal, 2004*.

[2] "DynamoRIO: Dynamic Instrumentation Tool Platform," http://www.dynamorio.org, 2017.

[3] L. Barroso, J. Clidaras, and U. Hoelzle, *The Datacenter as a Computer*. Morgan Claypool Publishers, Synthesis Lectures on Computer Architecture, 2013.

[4] C. Bienia, S. Kumar, and et al., "The parsec benchmark suite: Characterization and architectural implications," in *Proc. of PACT*. 2008.

[5] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *Proc. of PLDI*, 2012.

[6] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in *Proc. of ASPLOS*. 2013.

[7] ——, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proc. of ASPLOS*. 2014.

[8] ——, "Bolt: I Know What You Did Last Summer... In The Cloud," in *Proc. of ASPLOS*, 2017.

[9] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.

[10] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast Analytical Power Management for Latency-Critical Systems," in *Proc. of MICRO*, 2015.

[11] H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads," in *Proc. of ASPLOS*, 2014.

[12] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars, "Protean code: Achieving near-free online code transformations for warehouse scale computers," in *Proc. of MICRO*, 2014.

[13] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. of ISCA*. 2015.

[14] S. Misailovic, "Accuracy-aware optimization of approximate programs," in *Proc. of CASES*, 2015.

[15] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *Proc. of OOPSLA*, 2014.

[16] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *Proc. of ATC*. 2013.

[17] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "Accept: A programmer-guided compiler framework for practical approximate computing," in *UW-CSE-15-01-01*. 2015.

[18] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proc. of MICRO*, 2013.

[19] L. Tang, J. Mars, and et al., "Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers," in *Proc. of ASPLOS*, 2013.

[20] S. C. Woo, M. Ohara, and et al., "The splash-2 programs: characterization and methodological considerations," in *Proc. of ISCA*. 1995.