Quality Time: A Simple Online Technique for Quantifying Multicore Execution Efficiency

Anshuman Gupta
University of California, San Diego

Jack Sampson
Pennsylvania State University

Michael Bedford Taylor
University of California, San Diego

***Abstract*—In order to increase utilization, multicore processors share memory resources among an increasing number of cores. This sharing leads to memory interference, which in turn leads to a non-uniform degradation in the execution of concurrent applications, even in the presence of fairness mechanisms. Many utilities rely on application CPU Time both for measuring resource usage and inferring application progress. These utilities are therefore directly affected by the distorting effects of multicore interference on the representativeness of CPU Time as a proxy for progress. This makes reasoning about myriad properties from fairness, to QoS, to throughput optimality very difficult in consolidated environments, such as IaaS.**

We introduce the notion of Quality Time, which provides a measure of application progress analogous to CPU Time's measure of resource usage, and we propose a simple online sampling-based technique to approximate Quality Time with high accuracy. We have implemented three user-space tools called Qtime, Qtop, and Qplacer. Qtime can attach to an application to calculate its Quality Time online, Qtop is a dashboard that monitors the Quality Times of all applications on the system, and Qplacer leverages Quality Time information to find better application placements and improve overall system quality. With Quality Time, we are able to reduce the error in inferring execution efficiency from 150.3% to 25.1% in the worst case and from 30.0% to 7.5% on average. Qplacer can increase average system throughput by 3.2% when compared to static application placement.

I. Introduction

Multicore processors are already commonplace in desktop, server and mobile computing domains, and, with Moore's Law, the number of cores on commercial processors is constantly increasing. While this increase in the number of processing elements leads to increased aggregate throughput with every processor generation, it also means that there is an increasing potential for contention over uncore and off-chip resources, such as on-chip networks, coherence directories, memory controllers and DRAM.

The dynamic sharing of architectural resources among concurrent applications leads to interference that manifests itself through widely varying and unpredictable slowdowns that arise from the time-varying interaction of system components and the workload. Figure 1 shows the execution slowdown of a variety of benchmarks when they are co-scheduled with various combinations of background applications (0-3) on a 4-core 2.4 GHz Core 2 Duo machine. The slowdowns reach 2.7×, average 1.4×, and the standard deviation is very high relative to the execution time. Thus, an application's CPU Time, while still indicative of resource occupancy, has become a poor indicator of application progress, since total progress

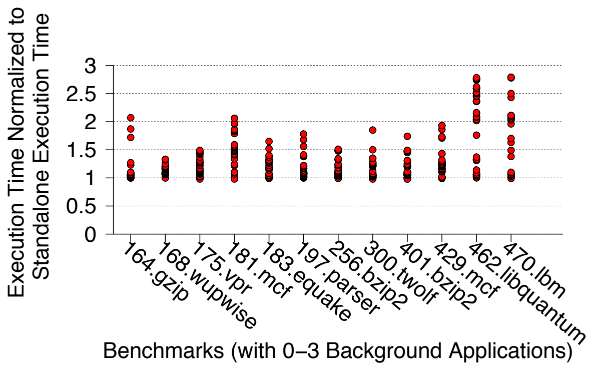


Fig. 1. ***Execution time of applications under varying co-schedules*** Despite the existence of both software and hardware mechanisms to improve fairness across concurrent applications, the range of execution times varies widely with different co-schedules due to interference.

now also depends on the net impact of the other concurrently scheduled applications. Unfortunately, many existing codes, ranging from "fair" thread scheduling heuristics, to the user commands `ps` and `top`, implicitly employ CPU Time as a proxy for application progress.

This magnitude of variation will only increase with greater core counts and degrees of resource sharing, and will result in this heuristic being less and less accurate. Going into the future, with increasing use of multicores and the advent of IaaS clouds, mechanisms that improve transparency about application progress will become increasingly useful; for instance, for improving metering and improving resource allocation to reduce interference.

We propose a metric, *Quality Time*, which improves on CPU Time as a measure of application thread progress. Quality Time is the amount of time that it would have taken the thread to reach its current execution progress had it had exclusive use of the machine. Using Quality Time, we can also compute `%Quality`, analogous to the `%CPU` field `ps` or `top` displays. Quality is reported on a per-application basis, not for the whole system, so ideally the sum of quality across applications is higher than 100%, reflecting that you are getting net throughput improvements over running a single application on the system. Intuitively, summing %Quality helps you assess the marginal throughput change of adding new applications.

In this paper, we show a simple sampling-based technique that enables low-cost online estimation of each application thread's Quality Time and `%Quality`. The technique is highly portable because it runs entirely in user-space and employs existing hardware mechanisms that are prevalent across most general purpose processors: an event counter that counts instructions issued, or alternatively, a counter that counts L1 data cache accesses. The technique allows measurement accuracy and overhead to be traded off, with typical settings of $< 1\%$ overhead yielding accurate estimations. We develop a user-space library, Qlib, that facilitates the measurement of Quality Time by concurrent applications, and introduce three tools using Qlib: Qtime, Qtop, and Qplacer. Qtime and Qtop are analogous to the Unix utilities time and top, respectively, and Qplacer uses Quality Time information to discover preferable affinities among co-scheduled applications and exploit them through remapping. All three of these applications run in user-space and require only that an environment variable be set to enable our QLib library; no recompilation is required.

The contributions of this paper include:

- **The QLib library for estimating Quality Time** We develop a library that coordinates the sampling of hardware counters to estimate Quality Time within 7.5%. On publication, we will release the source for both QLib and the three utilities that rely on it.
- **Efficient user-space estimation of Quality Time** We demonstrate that by merely setting an environment variable to pre-load QLib when an application runs, we can estimate Quality Time for unmodified binaries entirely in user-space.
- **Qtime** We have developed Qtime, a utility that provides real-time, online estimation of a given application's Quality Time for profiling and metering purposes. Compared to using CPU-time to estimate progress, Qtime drops worst case errors from 150.3% to 25.1%, and reduces average case error from 30.0% to 7.5% with an overhead of less than 1%.
- **Qtop** We have developed Qtop, a dashboard utility that provides visibility of execution quality across an entire system.
- **Qplacer** We have developed Qplacer, a user-space affinity mapping tool that actively monitors inter-application interference using Quality Time metrics, and moves application threads among cores to achieve 3.2% higher average throughput over a random scheduling, and provides up to a 105.0% maximum throughput improvement.

The remainder of the paper proceeds as follows. Section II describes our insights and methods for inferring Quality Time from hardware counters. Section III describes the implementation of QLib and the Qtime utility, and Sections IV and V describe our implementations of Qtop and Qplacer, respectively. Section VI showcases our results for Qtime accuracy and Qplacer throughput improvements. Section VII reviews related work, and Section VIII concludes.

II. Architectural Interference and Performance Measurement

In modern processors, interference is difficult to predict and manage because of three factors. First, microarchitectural resources, such as cache occupancy and bandwidth, are often shared in a free-for-all fashion and resource allocation decisions occur at very fine temporal granularity. Since main memory is vastly slower than on-chip memory, every memory access from any thread could have potentially detrimental effects on the execution of its cohorts if servicing the memory request forces the eviction of a more useful cache line. Second, applications running on out-of-order superscalars differ greatly in both their demand for resources and in their sensitivity to not having their demands met. Third, threads frequently transition through execution phases [1], which results in many possible phase combinations when different threads are run together. Thus, the space of all possible combinations of co-scheduled resource demands is large, frequently changing, and effects depend on interactions of address streams at run-time.

In the rest of this section, we explore the challenges in providing an interference-agnostic metric for application progress. We begin by examining the limitations of CPU Time, discuss alternative proxy metrics, and investigate how best to map from these new metrics to notions of application progress and execution quality in the presence of interference.

CPU Time As a Proxy for Application Thread Progress Implicitly, CPU Time is computed by the OS using either a timer circuit that asserts a periodic interrupt, or a cycle counter, which counts the number of clock cycles that have elapsed. Relying upon these hardware mechanisms for measuring thread progress works poorly in a multicore environment because these measures are oblivious to any external effects from other threads that may slow execution. In the context-switched uniprocessor domain, it was sensible to rely on CPU Time as a metric for thread progress. With interference, we are driven to cast about for potential hardware mechanisms that might be more effective substitutes.

Alternative Proxies Hardware metrics that are more closely tied to program progress are more promising. Of particular interest are hardware event counters, which count events in a processor's execution, and have become increasingly ubiquitous in modern processors. Such event counters provide a plausible proxy for execution progress because they are highly correlated to real-program progress. For example, in a simple loop accessing an array, the number of L1 data cache accesses, or even simply the number of instructions executed, correlate directly with the number of iterations/progress through the loop. Among the many event counters that are often available, most architectures provide one or more that are not particularly sensitive to interference, i.e. both isolated and concurrent runs generate similar event counts, making their measurement more strongly tied to program fundamentals than the current execution environment.

Deriving Time From Measuring Events However, there is one key challenge in using counters that are essentially counting program properties – normal, interference-free values may vary between different programs or different inputs, leaving us with no sense of the expected value for that run. In order to convert from "event" units to "time" units, we need to estimate the expected rate that those events occur at in interference-free execution. Because of program phases, event rates are likely to change with time even within a single thread, so static conversion ratios are insufficient. Simply reading the hardware counters in concurrent mode alone is insufficient as standalone performance estimate is required as well to measure slowdown.

We employ a sampling based approach in order to discern the expected event rate. To estimate an application's Quality Time, we repeatedly measure event counter statistics over both a very short sample period, where other application threads have been temporarily suspended, and then over a much longer execution phase, wherein it is co-scheduled. The short sample period is used to establish the event rate under interference-free execution. Then, we use that rate to convert the total number of events into a interference-free time value, Quality Time.

Quality-Time focuses on single-threaded multi-programmed workloads, which are present in most IaaS environments. For systems where multiple multithreaded applications are running, it would work poorly to stop individual threads in an application because they might synchronize with each other. Instead, we could compute a Ganged Quality-Time for each application, by simultaneously sampling, suspending, and resuming all the threads in that application.

Choosing a Sampling-Compatible Counter Using this sampling-based approach to sample the interference-free event rate relies upon the hypothesis that the sampled event-rate of a small execution period of the application can be used to extrapolate the behavior of a much larger period of execution. The choice of event counter greatly affects the quality of that extrapolation. Ideally, we would have an event that is 1) high frequency, to avoid aliasing errors due to integer precision counters; 2) oblivious to interference, in the sense that the number of counted events in a fixed sequence of instructions should not vary when other programs are run and 3) low variability in measured event-rates (e.g. events/progress) within the current program phase.

Every architecture provides a number of hardware event counters for insight and debugging purposes. To choose among the many such options, we examined the events exposed by the PAPI [2] library over all the benchmarks from Figure 1, each running interference-free, to select the events with minimal extrapolation error. We used the cycles/event ratio present in a single interval of 1 ms to predict the cycles for the next 99 ms using the event count for those 99 intervals (1% sampling). As shown in Figure 2, the extrapolation error varies widely depending on the event selected. L1 data cache access (L1-DCA) and instructions committed (TOT-INS) ended up being among the best metrics, because they were strong on all three requirements for events. First, they have low aliasing error compared to low frequency events such as L2 cache misses (L2-TCM). Second, they are interference-oblivious, unlike for instance, L2 cache misses. Third, and finally, they have relatively low variability (with L1-DCA having the lowest) across programs. For the remainder of the paper, we will focus on L1 data cache access and instructions committed as the two hardware events to track.

Choosing a Sampling Interval We explored the impact of sampling overhead and duration on Quality Time estimation accuracy. Ideally, the sampling duration should be long enough to *absorb* micro-variations in the application execution. At the same time, increasing the sampling duration increases the disruption to other threads during sampling, and, given a fixed sampling overhead, may negatively impact the dynamism of our predictions and our coverage of different application phases. Figure 3 shows the results of our sweep over sampling duration and frequency for the L1DCA event using the same experimental setup as in Figure 2. The accuracy increases with higher sampling rates, as expected, but this also leads to a higher overhead. On the other hand, the impact of sampling duration was not as direct, since different applications have different phase durations and behave differently for varying sampling durations. For this paper, we use a 1 ms sampling duration.

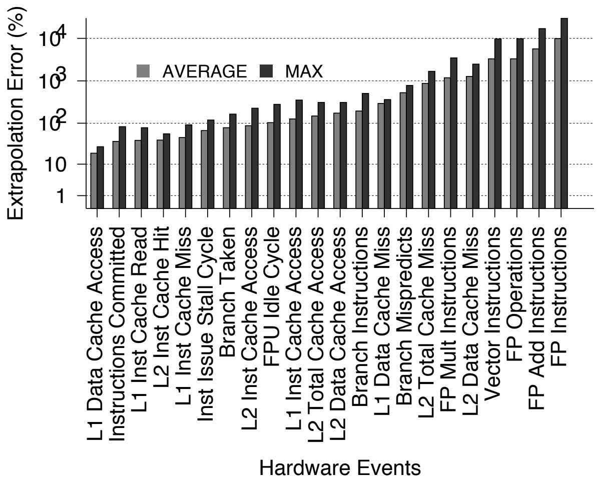


Fig. 2. ***Choosing hardware events that have low extrapolation error.*** Quality Time estimation samples a small region of execution and acquires an event rate that is then applied to estimate the behavior of a larger region of code. Event counters that generate good event rates have low aliasing errors, are interference-agnostic, and have low variability. The top counters were *Instructions Committed* and *L1 Data Cache Access*, and are used for the remainder of the paper.

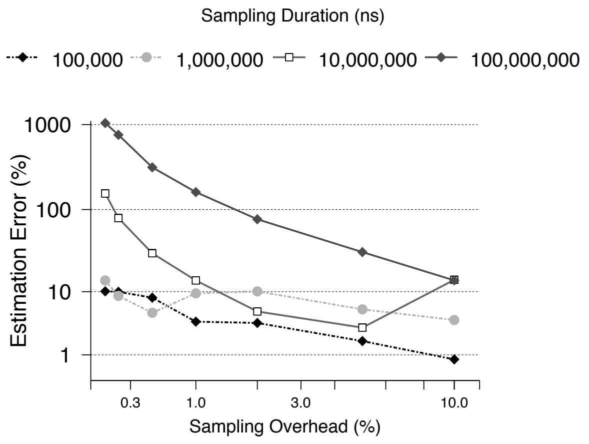


Fig. 3. ***Quality Time estimation accuracy increases with increasing sampling overhead.*** We can estimate Quality Time with less than 10% inaccuracy using sampling with an overhead of 1%. In this paper, we employ 1 millisecond sampling durations.

III. QLIB AND QTIME: MEASURING ONLINE APPLICATION EXECUTION QUALITY

There are two paths toward implementing counter-based approaches for estimating Quality Time. Namely, the approach can be implemented either in the kernel or in user-space. The former would provide a system-wide view of the effects of interference and allows for the greatest variety of responses to the incoming data. A user-space approach is also desirable because any user can portably reason about the quality of execution of their jobs on any system they may find themselves executing those jobs on. However, since a user-space approach can only control that user's applications, there is the potential for lost accuracy due to interference from other concurrent users.

In this section, we describe the design of Qtime, a user-space tool for estimating an application's Quality Time, and Qlib, the user-space library it relies on. Qlib requires measurements of hardware event counters to indicate application progress when running alone as well as concurrently. It uses the PAPI library [2] to provide access to the hardware event counters and read them out periodically. The PAPI library itself relies on the perfCtr module in linux kernel whose lighter-weight access to counters reduces complexity and overheads; moreover, PAPI is generally useful because it improves portability, and swaps in/out counters if the OS schedules in background threads. Due to PAPI's virtualization of these hardware event counters, Qtime can continue to get meaningful statistics even in the face of context switches.

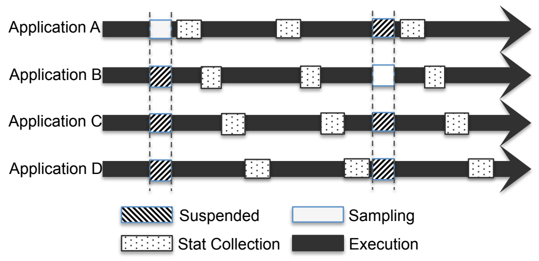


Fig. 4. ***Qtime framework.*** Qtime uses a sampling-based framework implemented in a dynamically linked library called QLib. Through this library, the applications regularly collect hardware event counters, either in standalone mode or concurrent mode, to estimate the application progress and execution quality online. To collect samples in standalone mode, all other applications are suspended. This synchronization between applications is managed through a shared memory region, which is managed by QLib.

In our user-space implementation, we allow concurrent applications to simultaneously estimate their Quality Time. Each invocation of Qtime attaches to a single application, whose Quality Time will then be measured. Each application must collect its event statistics both in isolated and concurrent execution. These measurements occur throughout execution to provide robustness against application phase-changes. Since we use hardware event counters to track progress, applications periodically read the current hardware event counters for their execution, as shown in Figure 4. Since hardware counters are usually shared among all concurrent applications on a processor, there is some orchestration required to get accurate measurements during concurrent execution. In addition to the sampling phases, wherein other applications are suspended, the applications also time-multiplex the collection of event counts during concurrent (non-sampling) periods. This is clearly seen in Figure 4 in the pattern of stat-collection periods between the two indicated sampling periods. Overall stat counts for the entire concurrent execution period are extrapolated linearly.

To orchestrate sampling among multiple applications, we create a library called *Qlib*. Rather than use a centralized controller, Qlib allows each application to measure Quality Time using its own resources. This reduces unnecessary context switches and provides scalability through decentralization.

The QLib library has to be linked to the application binary. To eschew recompilation, we take advantage of *LD_PRELOAD* to intercept *__libc_start_main* and link in *QLib*, a library to handle the communications among the applications, PAPI calls, and a *Shared Memory Region* holding Quality Time statistics, as shown in Figure 6. QLib interfaces with PAPI and configures the hardware event counters. It also sets up an overflow controller that tells PAPI to read out the hardware event counters every time the *PAPI_TOT_CYCLES* counter exceeds a given threshold. During these overflows, the statistics will be written (WRSHM) into the shared memory region shown in Figure 7. QLib writes the collected event counts during sampling phases to a different location in shared memory than the non-sampled statistics. These isolated application statistics are collected for use as representative samples. QLib also sets up the signal handlers for beginning the sampling phases, and an exit handler such that when the application exits, it frees up associated entries in the shared memory region. It also sets up a SIGALRM handler.

During both the sampling and execution phases, the applications periodically send event statistics, CPU-time, and the currently calculated Quality Time to a shared memory region, or SMR, for communicating this information, as shown in Figure 7. SMR is divided into shared memory region Entries, or SMREs, where each SMRE can store an application's PID, Quality Time, CPU-time, instructions committed and L1 cache accesses. At the beginning of execution, every application allocates a SMRE. The application periodically updates the SMREs with its execution statistics. The statistics reported are cumulative, so they are overwritten and there is no need of a read-modify-write. When an application exits, its SMRE is freed and can be allocated to other applications.

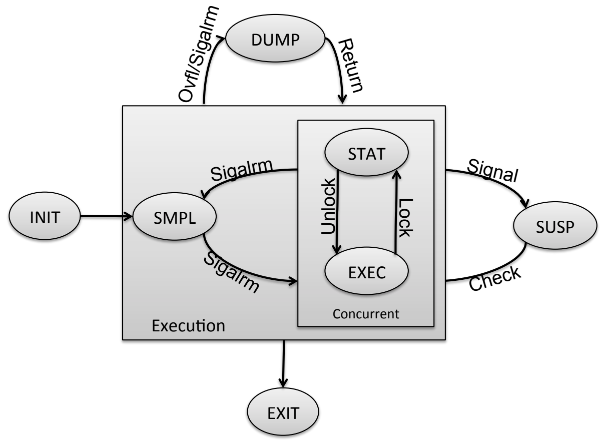


Fig. 5. ***State Machine for application execution when running Qtime tool.*** An application, when using Qtime tool to estimate its Quality Time, starts in the sampling state (SMPL) where it tries to collect event counters in standalone mode by signaling other applications to suspend (SUSP). Samples are collected (DUMP) in the counter overflow handler. After some predetermined time, the application resumes execution (EXEC) in concurrent mode and periodically collects event samples in the concurrent mode (STAT).

Figure 5 describes the state machine in QLib that controls the sampling and statistic collection behaviors for each thread. We use SIGALRM to periodically schedule when the application will be ready to get samples in standalone mode or concurrent mode, and when it should stop collecting samples. Once an application decides to sample an application, it sends a *real-time signal* to all other applications. On receiving the signal, all the other applications enter a suspended state and periodically read a shared memory region to check if the sampling state is over. Meanwhile the sampling application initiates the hardware event counters and collects the counters on periodic counter overflows. After a predetermined sampling duration, the application stops collecting samples, writes a value to the shared memory region that indicates that the sampling is over, and resumes execution. When the suspended applications next read the shared memory region, they resume their execution as well.

Qtime allows suspension overhead and Quality Time estimation accuracy to be traded off. In our experiments, we employed 1% sampling intervals in this paper. Thus, in an N core system with 1% sampling per application, the cost of suspending co-runners during sampling to be (N-1)% (e.g. 3% on a quad-core system). However, the application-level overheads can be a lot less than this (as low as 0%), because an application can have significant speedup when it has the machine all to itself.

The applications also collect samples during the concurrent execution. In the SIGALRM handler, if it needs to start collecting samples, it tries to obtain a lock placed in the shared memory region to ensure that no other application is currently using the hardware counters. On obtaining the lock, the application initializes the hardware counters, records event counts, and calculates the Quality Time. Lock contention is rare because each thread is implicitly assigned a time slot (think time-division multiplexing) when it tries to acquire the lock. A thread contends for the QLib lock with its co-runners for its first sampling period, but on receiving the lock, its sampling cycle is placed out-of-phase with co-runners leading to negligible contention in subsequent sampling periods. The Quality Time is also updated in the shared memory region every time the PAPI_TOT_CYCLES counter overflows.

Before the application begins standalone sampling, it dumps its current statistics in the shared memory region, changes the mode to sampling mode and resets the hardware event counters. Then it dumps the statistics in the shared memory region on overflows until it finishes standalone sampling. At this point, it again dumps the statistics, resets the hardware counters, changes to the EXEC state, and resumes execution. During the EXEC state, the application is not collecting any samples and, since it temporarily suspends the event gathering, it doesn't register any event overflows. As a result, during the EXEC state the Quality Time estimation is done inside the SIGALRM handler. Qtime records the ratio of application's Quality Time progress versus the CPU-time progress during the STAT state. Qtime then uses this ratio during the EXEC state to update the application Quality Time based on its CPU-time progress.

Since the hardware counters are read on cycle overflow, while the state transitions happen inside the SIGALRM handler, which is triggered on time, the state transitions and statistics collection are not synchronized. Thus we can get dirty data across these transitions. To avoid that, every time the application transitions from a statistics collecting state, it reads the counters, dumps the data into the shared memory region, and resets the hardware counters before starting the next state.

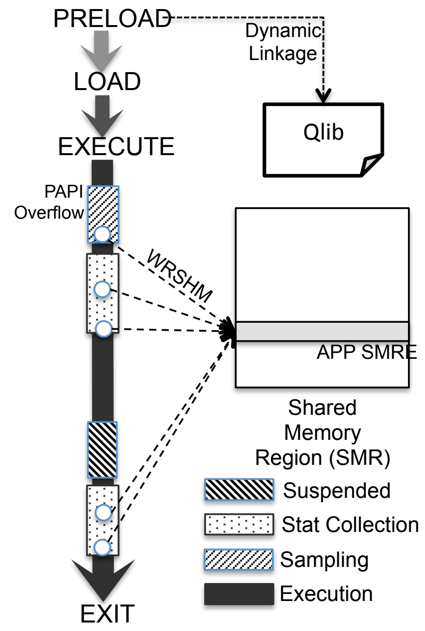


Fig. 6. ***Application Execution.*** Qtime preloads QLib at the launch of an application, which sets up the application sampling framework. During execution the statistics are written to the SMR when PAPI is triggered by an event counter overflow. Applications can suspend other applications to collect samples for standalone execution by sending a signal to other applications. The applications read the PIDs from the SMR for this purpose. On receiving the signal the application remains suspended, and periodically checks the SMR to see if the application has finished sampling, after which it resumes execution. In concurrent mode, applications can collect samples after obtaining a stat collection lock present in the SMR.

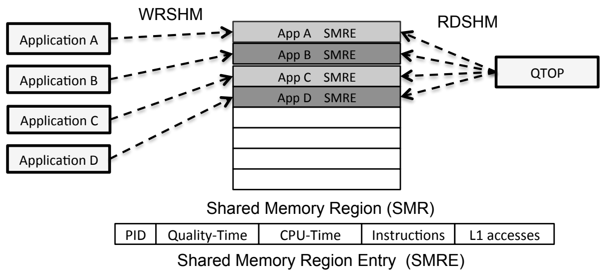
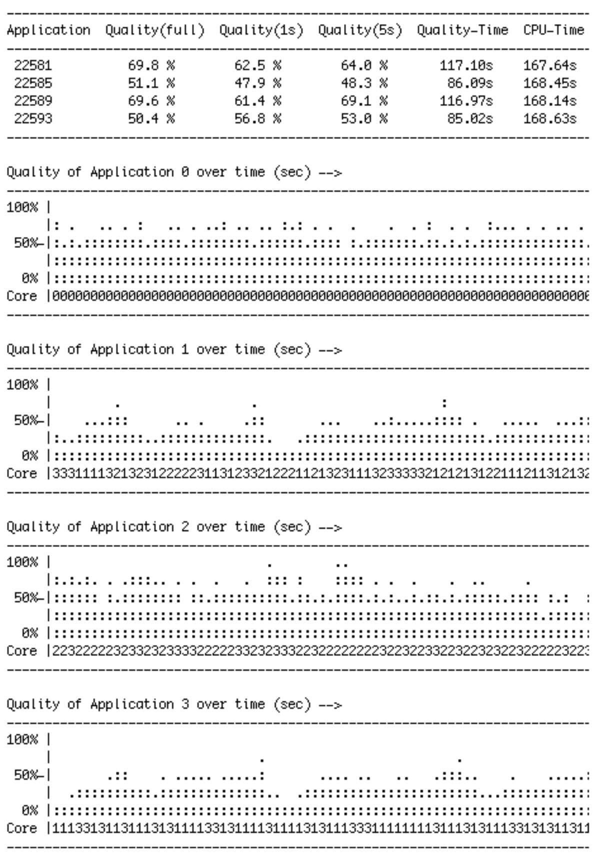


Fig. 7. ***Shared Memory Region.*** Applications write their Quality Time, CPU-times as well as execution statistics into a Shared Memory Region (SMR) where they can be read (RDSHM) by other applications for synchronization as well as management purposes. Each application is allocated a Shared Memory Region Entry (SMRE), which contains application PID, Quality Time, CPU-time, and other hardware statistics. The shared memory region also contain the variables indicating currently sampling application as well as the application currently collecting execution statistics.

Taken as a whole, Qtime provides an efficient and accurate tool that allows an application to see its own Quality Time. This is already highly useful for purposes such as profiling, but in the next section, we will show how collecting Quality Times for all currently running applications is useful for managing overall system quality.

IV. QTOP: A DASHBOARD FOR MONITORING EXECUTION QUALITIES OF OTHER APPLICATIONS

In addition to Qtime, we also implemented Qtop, a dashboard which continuously tracks application qualities, and provides monitoring and controlling facility for the overall system quality. Applications run with Qtime dump application statistics including PID, Quality Time, and CPU-time in the shared memory region to communicate with other applications. Qtop periodically reads this shared memory region for the execution statistics and maintains a history of the application qualities over time.



```
--------------------------------------------------------------------------------
Application  Quality(full)  Quality(1s)  Quality(5s)  Quality-Time  CPU-Time
--------------------------------------------------------------------------------
 22581          69.8 %         62.5 %        64.0 %       117.10s    167.64s
 22585          51.1 %         47.9 %        48.3 %        86.09s    168.45s
 22589          69.6 %         61.4 %        69.1 %       116.97s    168.14s
 22593          50.4 %         56.8 %        53.0 %        85.02s    168.63s
--------------------------------------------------------------------------------

Quality of Application 0 over time (sec) -->
--------------------------------------------------------------------------------
100% |
     |: .   .. . :   .. . ..: .. .. :.: . .  .    .  . :  . .  :... . . .. .
 50%-|:.:.:::::::.::::.::::::::.:::::.:::: :.:::::::.::.:.:.::::::::::::::::.
     |::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
  0% |::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Core |000000000000000000000000000000000000000000000000000000000000000000000000
--------------------------------------------------------------------------------

Quality of Application 1 over time (sec) -->
--------------------------------------------------------------------------------
100% |
     |        .                .                          :
 50%-|    ...:::      .. .     .::       ...    ..:.....:::: .   .....  ...::
     |:..:::::::::..::::::::::::::.   .::::::::::::::::::::::::::.:::::::::::
  0% |::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Core |333111113213231222223113123321222112132311132333332121213122111211312132
--------------------------------------------------------------------------------

Quality of Application 2 over time (sec) -->
--------------------------------------------------------------------------------
100% |                            .        ..
     |:.:.:. . .:::.. . .  .   .  ::: :    :::: . .  .    .  ..     .
 50%-|:::::: :.:::::::: ::.:::::::::::.::.:.::::.:.:..:.::.:.:::::.:::: :.:  :
     |:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::.::::::::
  0% |::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Core |223222222323323233322222332323332232222222232232233223223232232222232223
--------------------------------------------------------------------------------

Quality of Application 3 over time (sec) -->
--------------------------------------------------------------------------------
100% |
     |                               .                          .
 50%-|       .::    . ..... .....:        .... ..   ..   .::..    .    .....:
     |   .:::::::::.:::::::::::::::..   .:::::::::::::::::::::::...::::::::::
  0% |::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Core |111331311311113131111331311113111131311133311111111311131311133131311311
--------------------------------------------------------------------------------
```

Fig. 8. ***Qtop monitors the online quality of applications using the Qtime tool.*** Qtop periodically reads the information written to the shared memory region by all applications running the Qtime tool. It provides an estimate of the overall as well as the current quality of application execution, for example during last 1 second and the last 5 seconds as shown here. The history of application quality is shown in a live curve where each dot represents a 10% quality during that second. Underneath, the core on which the application was running at that point is also shown. These visual quality profiles can provide valuable insights into application interactions and phase behaviors.

The Qtop dashboard presents a compact visualization of the system execution quality at present as well as over the past. Qtop creates a live display of the quality of applications executing using the Qtime tool, as shown in Figure 8. It displays not just an application's Quality Time and CPU-time, but also takes a ratio of these two time metrics to calculate the application's execution quality over recent execution, and displays it in ascii at a resolution of 10% execution quality. It also summarizes the application quality over the entire execution as well as customizable periods, such as last 1 second or last 5 seconds, and displays them in the application summary. Finally, it also shows the cores on which the application was executing for each update.

Qtop can be readily used for detecting a lack of overall quality in the system as well as the offending workload. Similarly, it could be used to detect whether the system is underutilized and can be further consolidated. Qtop can monitor the entire system with very low overhead (∼1% core overhead) comparable to common linux monitoring tools such as *top*.

V. QPLACER: A QUALITY TIME BASED AFFINITY MAPPING TOOL

The quality of application execution can be significantly affected by architectural resource contention, and different schedules of applications to cores will result in different levels of contention. Since Qtime enables online monitoring of application qualities, it provides an opportunity to alter system configurations on-the-go and quickly react to changing application phases. This level of dynamic reactivity is very difficult when monitoring or profiling is done offline or on remote machines. We create a user-space tool, Qplacer, that attempts to improve system throughput by suggesting application placements that will improve execution quality. Qplacer is a user-space tool and does not require root access to run.

While Qplacer does not interfere with OS application scheduling in terms of which applications are currently scheduled, it attempts to increase system quality by discovering better application placements. For this purpose Qplacer uses simulated annealing, which ensures that even if the system converges to a locally optimal placement, it continues to probabilistically try other configurations and provide robustness against dynamically changing application phases and interactions. It also prevents frequent placement changes by forcing the system to stay in the current configuration for extended periods. Possibly other algorithms like genetic programming could be appropriate as well.

Qplacer uses the following simulated annealing model:

- *States, S*: Represented by unique configurations in the system. Two configurations are not unique if all applications are homogeneous co-locations in the two configurations. For example, on our evaluation machine, there are three possible unique configurations, i.e. "ab,cd", "ac,bd", and "ad,cb".
- *Energy, E(S)*: Each configuration has an associated energy. Qplacer records the *Weighted-Quality* of each application in each configuration, and uses the sum of these Weighted-Qualities of all applications as the energy of the configuration. Every interval, the Weighted-Quality of all applications are calculated by taking the sum of their existing Weighted-Quality, weighted down by a damping constant, and the current Quality of these applications, weighted down by one minus the damping constant. The initial Weighted-Quality of each application is a 100%.
- *Temperature, T*: The system temperature determines its entropy. Qplacer is less willing to change soon after a configuration switch. So it uses the natural logarithm of the time (in milliseconds) since last switch as the system temperature.
- *Switch Probabilities, P*: Finally, Qplacer determines the probability to switch from state S to S' as:

$$P(S, S', T) = e^{\beta\times(E(S')-E(S))+T} \qquad (1)$$

Configuration of System 1

OS	CentOS release 5.8 (Final), Linux 2.6.39.4
Processor	Quad-core Intel Xeon X3220, 2.40GHz, 2 x 4MB shared L2 cache
Memory	1066MHz FSB, 6GB DDR3

Configuration of System 2

OS	CentOS release 5.8 (Final), Linux 2.6.29.6
Processor	2-Socket 6-core AMD Opteron 2427, 2.40GHz, 6MB shared L3 cache
Memory	2400MHz HT, 64GB DDR3

TABLE I. SYSTEM CONFIGURATIONS USED IN OUR EVALUATION.

Application	Suite	Description
164.gzip	SPEC2000	File compression
168.wupwise	SPEC2000	Quantum chromodynamics
175.vpr	SPEC2000	Place and route CAD tool
181.mcf	SPEC2000	Vehicle scheduling algorithm
183.equake	SPEC2000	Seismic wave propagation
197.parser	SPEC2000	Word processing
256.bzip2	SPEC2000	File compression
300.twolf	SPEC2000	Computer aided design
401.bzip2	SPEC2006	File compression
429.mcf	SPEC2006	Vehicle scheduling algorithm
462.libquantum	SPEC2006	Quantum computing
470.lbm	SPEC2006	Computation fluid dynamics

TABLE II. BENCHMARKS USED IN OUR EVALUATION.

β is a convergence constant. The probabilities are normalized so that the sum of all switch probabilities is 1.0.

Every interval, Qplacer estimates the switching probabilities, and then generates a random number to determine the next state of the system. The Qtop monitoring tool displays the application swaps made. While there are overheads associated with swapping configurations, our results indicate that the swaps occur infrequently enough, and the programs converge to beneficial schedules rapidly enough that affinity control can provide benefits in the common case. This is particularly promising compared to static profiling approaches, as it means that Qplacer or similar approaches will provide benefits even for previously unseen programs. Thus, Qplacer will be particularly useful for expanding IaaS and cloud computing domains wherein arbitrary user computations may be offloaded to consolidated servers for execution.

VI. EVALUATION

We now describe the evaluation of our user space tools: Qtime, which approximates application Quality Times, and Qplacer, which uses simulated annealing to improve application placements.

A. Evaluation Methodology

We implemented a user-space tool, Qtime, to suspend and sample applications, and calculate Quality Time using statistics collected from applications. The other user-space tool Qplacer is a tool that monitors application qualities and improve application placement using simulated annealing. We run our experiments on the setup described in Table I. We use the PAPI library [2] version 5.0.1.0 to collect application execution statistics. We use sampling periods of 1 millisecond and sampling periods of 1% compared to concurrent execution periods. We use 0.25 as the damping constant for simulated annealing, and 2.00 as the convergence constant. We use the PAPI overflow threshold as 1 million cycles.

We use benchmarks from SPEC2000 [3] and SPEC2006 [4] for our evaluation. We describe the benchmark characteristics in Table II. For each workload, we run all the benchmarks in a loop until all the applications have finished at least once and measure the Quality Time for all the applications simultaneously. The applications are sampled periodically (once every 100ms) to manage phase changes. The experiments are run for the full application duration; since SPEC benchmarks with *ref* inputs typically run for multiple minutes and phases changes can happen at sub-millisecond scale, the experimental runs did see frequent phase changes and Qtime was indeed able to adapt to them.

Evaluating our Quality Time scheme and comparing alternatives requires us to replay each scheme in the face of potentially variable machine behavior. Thus, we run both our baseline isolated executions and each benchmark tuples multiple times to better cover the scope of real program behaviors. We report the evaluation results for Qtime accuracy based on 12 standalone runs for the 12 benchmarks on both Systems, along with 78 runs for all possible pairs of 12 benchmarks with 1 instance each for both System 1 and System 2, 78 runs with 2 instances each for system 1, and 78 runs with 3 instances each for System 2. We also run every experiment thrice to adapt for system noise and report their arithmetic means. Therefore, we report the Qtime results based on 1008 runs overall. On the other hand for Qplacer evaluation, we report results based on all 78 possible pairings of the 12 benchmarks with 2 instances each on System 1. We run 3 experiments with Qplacer and 3 without Qplacer for each benchmark pair and report the arithmetic means to compensate for system noise. Therefore, Qplacer results are reported based on 468 runs overall. For comparing total system throughput we compare the sum of execution times for all the applications' first runs. For comparing the accuracy of instantaneous estimates, we use the simplifying assumption that, for these benchmarks, an equivalent number of committed instructions implies an equivalent amount of application progress. This simplifying assumption is borne out by the minimal variance in total committed instructions across runs for SPEC benchmarks.

B. Qtime's Quality Time Estimation Results

We ran Qtime with a sampling period of 1% compared to the concurrent execution period and a sampling duration of 1 millisecond using the L1-DCA and TOT-INS events. We observe that while the existing method of using CPU time to track application progress has an error of 29.5% on average (arithmetic mean) for the two systems described in Table I. The errors on these two systems evaluated are 36.9% and 22.2% on average and 150.3% and 115.6% at most, respectively. Our initial simple sampling-based technique is able to reduce this error for almost all the workloads, as shown in Figure 9, to 10.8% on average when using L1 data cache access and 16.4% when using instructions committed, as shown in Figure 9. However, for some workloads, the error goes up significantly when using our sampling technique. This is due to the amplification effects of sampling: If our

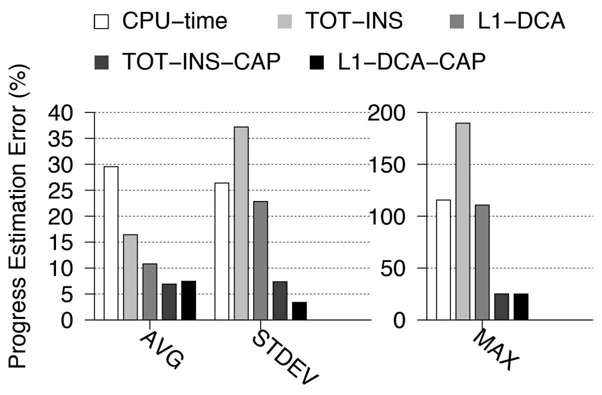


Fig. 9. ***Quality Time can be accurately and efficiently estimated by our technique.*** Sampling-based Quality Time estimation has an accuracy of 10.8% on average, when using L1-DCA. However, the maximum error is 110.7% with L1-DCA and 189.6% with TOT-INS. We are able to reduce the max error to 25.1% (and 7.5% on average) when using L1-DCA by capping the Quality Time each interval.

"representative" sample is actually in a different phase than the execution we use it to predict, then our back-calculation of time can be erroneous. In particular, if we sample during a low event-frequency phase, and predict for a high event-frequency phase, our estimation of time elapsed can be implausibly high. As a result, the maximum error is still very high, 110.7% and 189.6% for L1 data cache access and instructions committed, respectively, as shown in Figure 9.

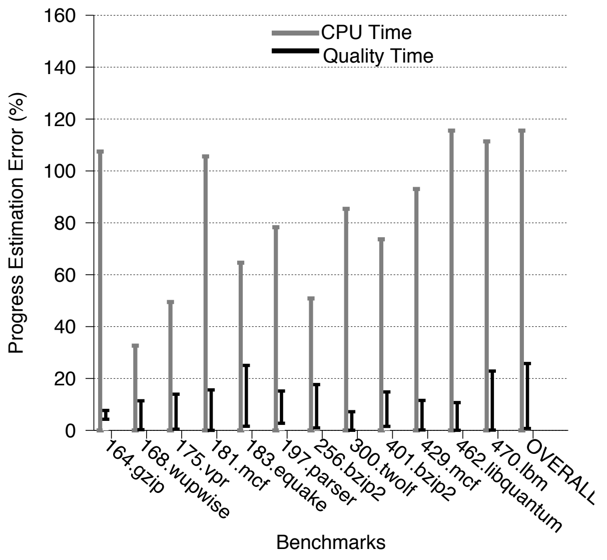


Fig. 10. ***Qtime's Quality Time estimation, by benchmark.*** We show the ranges of error for both CPU-Time and Qtime's L1-DCA-based Quality Time estimation. Our technique improves the accuracy and reduces the variability in approximating application progress in the presence of interference.

Fortunately, we can refine our technique by applying *capping* to our Quality Time estimation to improve our results. We know that for every interval, the Quality Time cannot be less than zero. Also, ignoring the rare case of speedups when sharing resources, the Quality Time can be assumed to be less than or equal to the CPU time. We apply these two limits every intervals to bound our Quality Time estimation. As shown in Figure 9, capping leads to better Quality Time estimation, especially when using L1 data cache accesses. When using L1 data cache access, capping reduces the maximum error to 25.1% and arithmetic mean of absolute errors to 7.5% (8.2% and 6.7%, respectively, on the two systems evaluated) with a standard deviation of 3.4%, as shown in Figure 9. The instantaneous errors cancelled each other out and as a result the absolute error in Quality-Time remained stable. This implies that the longer the application runs the smaller the %Quality error. Figure 10 shows the final results by benchmark.

We get similar Qtime accuracy results on systems with Hyper-threading. This was made possible due to PAPI's virtualization and Intel's hyper-threaded event counter support [5]. A full evaluation of Qtime/HT has been left as future work.

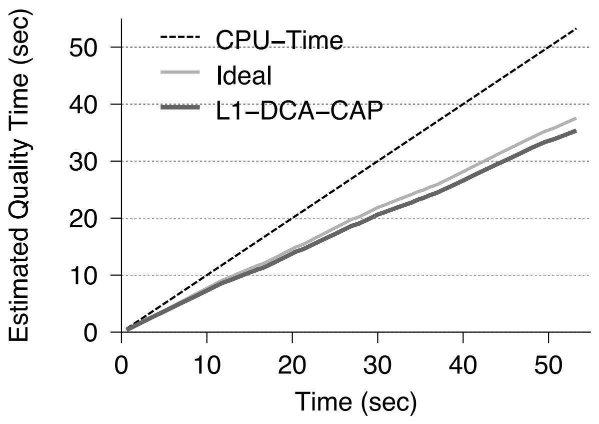


Fig. 11. ***Instantaneous tracking of Quality Time.*** We plot the calculated and ideal Quality Time for 401.bzip2 as a function of time. We show that our technique can provide accurate instantaneous estimations of Quality Time as well as accurate holistic estimations, with the estimated Quality Time closely tracking the ideal.

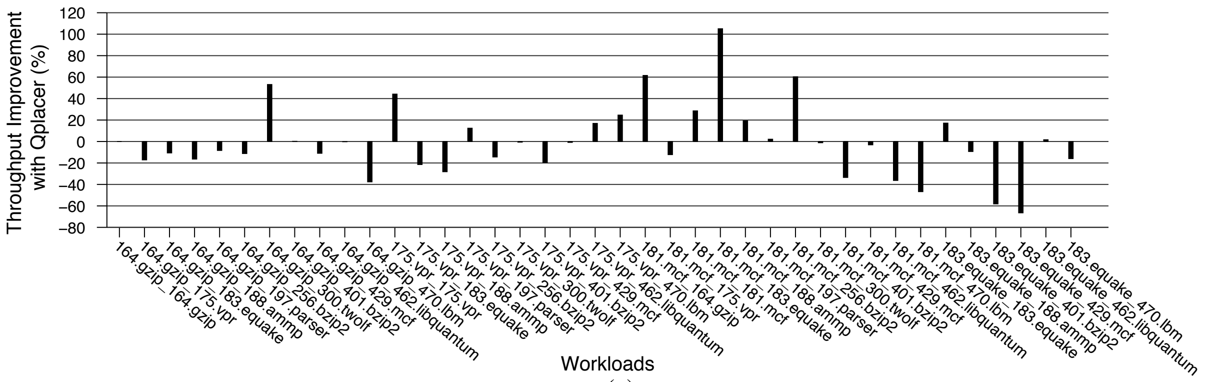
Figure 11 shows an example of how our technique can provid consistently accurate estimations over the course of execution, as well as accurate summary statistics. Figure 11 shows that, for the entire course of the execution of 401.bzip2, the estimation of progress tracks very closely with the ideal. While whole-execution accuracy is sufficient for use cases such as IaaS metering, fine-grained accuracy is necessary for using Quality Time for scheduling, resource allocation, or other dynamic decisions.

Getting sampling to work in a dynamic system without excessive extrapolation error is indeed challenging. A key result of our paper, as seen here, is that our approach, a combination of frequent sampling, judicious choice of hardware events, and capping, i.e. bound error in one direction, allows the system to infer much more information about execution quality than is possible from CPU-time alone. Combined with CPU-time, Qtime allows for a more complete picture of execution properties under contention.

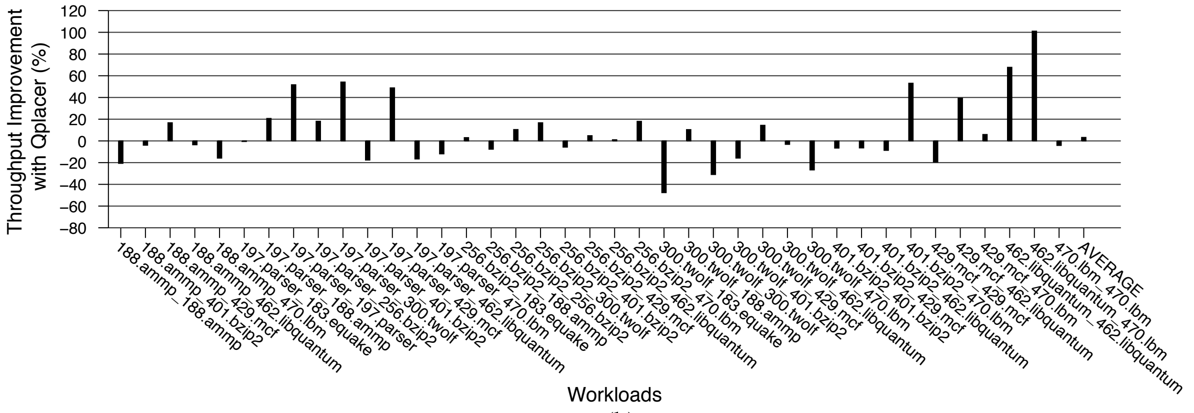
C. Throughput Improvement with Qplacer

Since Quality Time can accurately estimate the impact of interference on an application, making Quality Time information available to a scheduler can be useful in dynamically discovering which application pairings result in the least conflicting schedules.

We evaluated our affinity scheduler's ability to produce good co-schedules on the Quad-core Intel Xeon evaluation



(a)



(b)

Fig. 12. ***Qplacer can improve throughput by using Quality Time for placement.*** Qplacer can use the online Quality Time estimates for the applications, which changes over time due to application phases, and dynamically determines application placement using simulated annealing. In our evaluation on System 1, described in Table I, this leads to an average throughput improvement of 3.2% over 78 runs (all possible pairs of 12 benchmarks; two instances of each benchmark launched) and a maximum throughput improvement of 105.0%. The improvements should be even larger for systems with higher heterogeneity.

system (System 1 in Table I) by scheduling two copies each of two applications on the multi-chip module with two processors, each with two cores. The system works for more applications than number of cores as well. We run one application per core to match the behavior of typical clusters with batch schedulers and commercial IaaS systems, such as Amazon EC2. We compare our tool's dynamic scheduling against the average throughput over the 3 possible distinct static scheduling configurations. Figure 12, where each bar represents the % increase in total throughput for 4-application workloads when using Qplacer compared to the average throughput reported by 2-possible static application placements, shows that for most sets of applications, choosing to dynamically reschedule based on Quality Time indications of interference is beneficial for overall throughput. In cases where there was little potential benefit between the best and average schedules, the technique sometimes decreased throughput due to the errors in estimation of the Quality Time leading to a misguided choice of application placement. The migration and sampling overheads were small and did not significantly impact the overall throughput. Improving our Quality Time estimation and heuristic to avoid these low benefit cases will be a future effort.

VII. Related Work

Architectural Interference Typical commercial multicore processors share resources among concurrent threads, and resource sharing leads to interference between applications, as described by Tang et al. [6]. Govindan et al. [7] show that even with the use of hypervisors the unpredictability in slowdowns is very high. Stillwell et al. [8] also examined the performance impact of resource sharing in servers at the system level. For these resource-sharing processors, it is important to precisely estimate the performance of applications in order to improve resource accounting and utilization, as shown by Armbrust et al. [9]. For resource-sharing commercial systems, it is important to accurately estimate the progress of applications and exercise control over it in order to maintain performance guarantees and improve resource utilization, as also pointed out by Buttazzo [10], since even state-of-the-art resource management schemes, such as the ones proposed by Gohner et al. [11] and Elmroth et al. [12], do not account for application slowdowns due to sharing of processor resources.

The emergence of manycore computing in the server space, punctuated by the arrival of Tilera's Tile Gx100 [13] and Intel's 48-core SCC [14], offers higher density and energy-efficiency. However, these benefits are only realizable if interference is more carefully controlled, as these manycore processors heavily rely on shared resources. Quality Time provides a performance abstraction for interference-free execution. Such abstractions are useful for utilities such as kernel-schedulers. We are also witnessing an emergence of multicores in other ecosystems such as embedded computing, such as the smartphones. These systems are under an even bigger pressure to share resources due to limited area budgets, and this can lead to difficulties in existing resource management techniques for multiprogrammed embedded systems, such as the ones proposed by Lipari et al. [15], Bernat et al. [16], and Beccari et al. [17], reducing the effectiveness of techniques such as resource reservation [18] and proportional resource sharing [19] for real-time systems.

Quality Time provides a performance abstraction for interference-free execution. Such abstractions can be useful for high-order decisions such as resource management and progress tracking, as suggested by Zhang et al. [20], in manycore systems.

Resource Isolation Performance isolation has been proposed as a means to reduce resource interference. Verghese et al. [21] proposed mechanisms for performance isolation for resources such as I/O bandwidth and storage, while Banga et al. [22] suggested resource containers to isolate and account for system-level resource usage. However, since typical manycore architectures rely on shared processor resources, this performance isolation (and not just resource isolation [23]) should be extended to the micro-architectural levels to account for application slowdowns due to sharing of processor resources.

Performance Estimation Performance estimation has been studied in existing literature for different objectives. For example, Eyerman et al. [24] used a mechanistic performance modeling to create CPI-stacks, which can be used to determine performance bottleneck in systems. These models however require some knowledge of the microarchitecture as well as some offline regression. Lee et al. [25] use offline regression to estimate performance and power consumption of applications. However, our performance estimation technique is online and requires no knowledge of the underlying microarchitecture.

Research has also been done on creating application progress estimates in hardware. TimeCube [26] tracks application progress using an analytical performance estimation model similar to the one proposed by Solihin et al. [27]. These models are able to model minute architectural details such as tracking prefetches, measure memory bandwidth constraints, and cache intricacies such as dirty lines, via mechanisms like those proposed by Kaseridis et al. [28]. They can model off-chip architectural resources that affect application performance, such as the details of DRAM DDR protocol and bank buffer behaviors. These mechanisms can even obviate the need for standalone execution by using shadow structures, for example shadow cache techniques that have been proposed for associative caches, such as by Zhou et al. [29].

However, while it is faster to accumulate execution statistics in hardware and the performance estimation of mechanisms designed alongside the architectures can produce very high accuracy due to their intimate knowledge of the architectural details, no commodity system currently implements these mechanisms. Thus, these techniques are not germane for the many systems that use existing multicore processors, which are already facing architectural interference problems. The techniques to mitigate the interference problem on these systems need to be implemented in software. Qtime focuses on portability and using a purely online approach (no off-line analysis). Although on the surface this precludes inclusion of micro-architectural knowledge, some micro-architectural details (e.g., cache sizes) are relatively easy to read from the system and could be used to potentially improve Qtime.

Performance Analysis Several performance analysis tools and techniques have been proposed previously, such as VTune [30] and Cilkview [31]. However, these techniques analyze application performance in isolation, and usually drive analysis of offline systems, such as by London et al. [32]. Zagha et al. [33] propose an offline performance analysis using hardware counters for specifically MIPS R10K, whereas our technique can be used on any platform. Compared to the context-sensitive technique proposed by Ammons et al. [34], our technique provides less information, which is sufficient for our purpose; as a result, the tool is efficient enough to be used online.

Kambadur et al. [35] propose using remote machines to analyze profile data obtained from live datacenter applications using Google Wide Profiler [36]. This technique has a very low overhead; however, since they collect profiles on live applications, but process them on remote machines, the round-times are too large to make phase-sensitive analysis/scheduling. Moreover, they do not have standalone performance estimates, which is very useful in providing bounded QoS to applications in certain settings, such as IaaS or embedded-systems.

Resource Management Several hardware techniques have been proposed to manage resources inside processor itself, such as profiling based allocation schemes proposed by Liu et al. [37] and Suh et al. [38]. Bitirgen et al. [39] proposed simultaneous cache and bandwidth allocation using machine learning. These management schemes are tuned for varying purposes; for example, Hsu et al. [40] tune their cache allocation algorithm to maximize different metrics such as fairness and throughput; and Guo et al. [41] allocate cache partitions based on QoS provided by choosing between strict, elastic, and opportunistic schemes. However, these policies are better managed at the software level, because of the possible changes in the system requirements.

In software, co-scheduling can reduce pressure on the resources and increase performance when sharing scarce resources between multiple applications. Amongst previous works, Cazorla et al. [42], Jiang et al. [43], El-Moursy et al. [44] and Snavely et al. [45] discuss mechanisms for application scheduling. Federova et al. [46] examined OS-level scheduling to optimize CMT (multithread CMPs) performance. However, due to increasing heterogeneity within the processors as well as across different processors, application placement is becoming increasingly important. Qtop uses simulated annealing to affect application placement while leaving the scheduling decisions to the kernel.

VIII. Conclusion

We observe that, even though multicore processors are being used in all forms of computing, many software utilities continue to use CPU time as the proxy for application progress – which can be misleading when architectural resources are shared among cores. On the two multicore processors evaluated, we show these distortions to be 30.0% on average and up to 150.3% in the worst case. We introduce the notion of Quality Time, a simple quantification for application execution efficiency, which can be used by software utilities as a proxy for application progress. Quality Time can be estimated efficiently and accurately with user-space utilities, without recompilation. We implement three tools, Qtime, Qtop, and Qplacer, which generate, monitor, and use Quality Time, respectively, to reduce the error in tracking application progress to 7.5% on average and 25.1% in the worst-case, and increase throughput by 3.2% on average and 105.0% in the best case.

ACKNOWLEDGEMENT

This work was partially supported by NSF Awards 06483880, 0846152, 0811794, 1018850, 0811794, and 1228992, and by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES

[1] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS*, 2002.

[2] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," 2009.

[3] "SPEC CPU 2000 benchmark specifications," 2000, SPEC2000 Benchmark Release.

[4] "SPEC CPU 2006," 2006. [Online]. Available: http://www.spec.org/cpu2006/

[5] "Implement performance monitoring for hyper-threading technology," *Intel Developer Forum*. [Online]. Available: http://goo.gl/aYq4Xs

[6] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *ISCA*, 2011.

[7] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *SOCC*, 2011.

[8] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation using virtual clusters," in *International Symposium on Cluster Computing and the Grid*, 2009.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, April 2010.

[10] G. Buttazzo, "Research trends in real-time computing for embedded systems," *SIGBED Rev.*, 2006.

[11] M. Gohner, M. Waldburger, F. Gubler, G. Rodosek, and B. Stiller, "An accounting model for dynamic virtual organizations," in *International Symposium on Cluster Computing and the Grid, 2007.*

[12] E. Elmroth, F. G. Marquez, D. Henriksson, and D. P. Ferrera, "Accounting and billing for federated cloud infrastructures," in *International Conference on Grid and Cooperative Computing*, 2009.

[13] R. Schooler, "The processor: Many-core for embedded and cloud computing," in *Workshop on High Performance Embedded Computing*, 2010.

[14] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, S. Borkar, V. De, R. C. D. Wijngaart, and T. Mattson, "A 48-Core IA-32 Message-Passing Processor with DVFS in 45 nm CMOS," in *ISSCC*, 2010.

[15] G. Lipari and S. K. Baruah, "Efficient scheduling of real-time multi-task applications in dynamic systems," in *Real Time Technology and Applications Symposium*, 2000.

[16] G. Bernat and A. Burns, "Multiple servers and capacity sharing for implementing flexible scheduling," *Real-Time Syst.*, Jan. 2002.

[17] G. Beccari, S. Caselli, and F. Zanichelli, "A technique for adaptive scheduling of soft real-time tasks," *Real-Time Syst.*, Jul. 2005.

[18] L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *Real-Time Syst.*, Jul. 2004.

[19] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Real-Time Systems Symposium*, 1996.

[20] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen, "Processor hardware counter statistics as a first-class system resource," in *Workshop on Hot Topics in Operating Systems*, 2007.

[21] B. Verghese, A. Gupta, and M. Rosenblum, "Performance isolation: sharing and isolation in shared-memory multiprocessors," in *ASPLOS*, 1998.

[22] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: a new facility for resource management in server systems," in *OSDI*, 1999.

[23] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," in *ISCA*, 2007.

[24] S. Eyerman, K. Hoste, and L. Eeckhout, "Mechanistic-empirical processor performance modeling for constructing cpi stacks on real hardware," in *ISPASS*, 2011.

[25] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," *SIGARCH Comput. Archit. News*, 2006.

[26] A. Gupta, J. Sampson, and M. B. Taylor, "Timecube: A manycore embedded processor with interference-agnostic progress tracking," in *SAMOS*, 2013.

[27] Y. Solihin, V. Lam, and J. Torrellas, "Scal-tool: pinpointing and quantifying scalability bottlenecks in dsm multiprocessors," in *SC*, 1999.

[28] D. Kaseridis, J. Stuecheli, J. Chen, and L. K. John, "A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems." in *HPCA*, 2010.

[29] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *ASPLOS*, 2004.

[30] J. Reinders, "Vtune performance analyzer essentials," in *Intel Press*, 2005.

[31] Y. He, C. E. Leiserson, and W. M. Leiserson, "The cilkview scalability analyzer," in *SPAA*, 2010.

[32] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer, "End-user tools for application performance analysis using hardware counters," in *PDCS*, 2001.

[33] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance analysis using the mips r10000 performance counters," in *SC*, 1996.

[34] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *PLDI*, 1997.

[35] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *SC*, 2012.

[36] 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.

[37] C. Liu et al., "Organizing the last line of defense before hitting the memory wall for CMPs," in *HPCA*, 2004.

[38] G. E. Suh, S. Devadas, and L. Rudolph, "Analytical cache models with applications to cache partitioning." in *HPCA*, 2002.

[39] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *MICRO*, 2008.

[40] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource," in *PACT*, 2006.

[41] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A framework for providing quality of service in chip multi-processors," in *MICRO*, 2007.

[42] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero, "Architectural support for real-time task scheduling in smt processors," in *CASES*, 2005.

[43] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *PACT*, 2008.

[44] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas, "Compatible phase co-scheduling on a cmp of multi-threaded processors," in *IPDPS*, 2006.

[45] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *ASPLOS*, 2000.

[46] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, "Performance of multithreaded chip multiprocessors and implications for operating system design," in *USENIX Technical Conference*, 2005.