A

Thread Tranquilizer: Dynamically Reducing Performance Variation

Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan, University of California, Riverside

To realize the performance potential of multicore systems, we must effectively manage the interactions between memory reference behaviour and the operating system policies for thread scheduling and migration decisions. We observe that these interactions lead to significant variations in the performance of a given application, from one execution to the next, even when the program input remains unchanged and no other applications are being run on the system. Our experiments with multithreaded programs, including the TATP database application, SPECjbb2005, and a subset of PARSEC and SPEC OMP programs, on a 24-core Dell PowerEdge R905 server running OpenSolaris confirms the above observation. In this work we develop Thread Tranquilizer, an automatic technique for simultaneously reducing performance variation and improving performance by dynamically choosing appropriate memory allocation and process scheduling policies. Thread Tranquilizer uses simple utilities available on modern Operating Systems for monitoring cache misses and thread context-switches and then utilizes the collected information to dynamically select appropriate memory allocation and scheduling policies. In our experiments, Thread Tranquilizer yields up to 98% (average 68%) reduction in performance variation and up to 43% (average 15%) improvement in performance over default policies of OpenSolaris. We also demonstrate that Thread Tranquilizer simultaneously reduces performance variation and improves performance of the programs on Linux. Thread Tranquilizer is easy to use as it does not require any changes to the application source code or the OS kernel.

Categories and Subject Descriptors: D.4.1 [**Process Management**]: Scheduling, Threads; D.4.8 [**Performance**]: Monitors, Measurements

General Terms: Performance, Measurement, Algorithms

Additional Key Words and Phrases: Multicore, Variation, Priority, Context-switch, Cache miss-rate

ACM Reference Format:
Pusukuri, K.K., Gupta, R., Bhuyan, L.K. 2012. Thread Tranquilizer: Dynamically Reducing Performance Variation. ACM Trans. Arch. Code Opt. V, N, Article A (January YYYY), 20 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The advent of multicore systems presents an attractive opportunity for achieving high performance on a wide range of applications. However, due to its non-uniform memory access latency (NUMA), the performance of applications on a multicore system is highly sensitive to operating system policies. The impact of operating system (OS) policies on performance makes consistently maximizing performance a significant challenge. Even after an application has been *tuned*, it may exhibit significantly different levels of performance from one execution to next. This is demonstrated by the results of the following experiment that we conducted. We executed 15 multithreaded programs, including the TATP database application, SPECjbb2005, as well as programs from PARSEC and SPEC OMP suites, on a 24-core Dell PowerEdge R905 server running

This research is supported in part by NSF grants CCF-0963996, CNS-0810906, CCF-0905509, and CSR-0912850 to the University of California, Riverside, CA; e-mail: kishore@cs.ucr.edu.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Program	Min Time (seconds)	Max Time (seconds)	SD (σ)	Min Speedup	Max Speedup	% Diff	OPT Threads	Type
swim	265.0	324.0	25.1	3.7	4.5	17.8	48	Mem
wupwise	147.2	190.3	17.4	7.3	9.5	23.2	72	Mem
equake	182.5	217.9	12.8	2.5	2.9	13.8	12	Mem
gafort	221.6	256.4	12.0	9.0	10.5	14.3	24	Mem
streamcluster	204.4	225.0	10.2	3.8	4.2	9.5	13	Mem
facesim	172.6	199.3	7.7	4.5	5.1	11.8	17	Mem
mgrid	28.6	35.7	3.4	4.5	5.6	19.6	24	CPU
canneal	93.3	102.0	3.0	4.6	5.1	9.8	43	Mem
x264	54.0	58.1	1.4	7.3	7.9	7.6	64	CPU
fluidanimate	64.4	68.5	1.4	12.2	13.0	6.2	21	Mem
bodytrack	43.4	45.2	0.7	11.0	11.5	4.3	26	CPU
swaptions	21.1	22.5	0.5	20.0	21.4	6.5	33	CPU
ferret	41.4	42.9	0.5	14.7	15.2	3.3	63	CPU
TATP	35836	45976	3358	5.2	6.6	13.6	43	Mem
SPECjbb2005	105154	121177	4741	4.1	4.7	12.8	42	Mem

Table I: Performance variation of the programs. Programs are executed with OPT threads where OPT threads is the minimum number of threads that gives best performance on our 24-core machine. Speedup is relative to the serial version of the programs. Performance of TATP is expressed in transactions per second (throughput) and speedup is relative to the single client throughput. Performance of SPECjbb2005 is expressed in 'SPECjbb2005 bops' (throughput) with OPT warehouses. Speedup of SPECjbb2005 is relative to the single warehouse (single warehouse, i.e., 35 threads) throughput. Here, % Diff is the percentage difference between Max Speedup and Min Speedup.

OpenSolaris.2009.06. Each program was executed 10 times while no other applications were being run. As shown in Table I, we observed significant variation in the performance of each program over its ten executions. The table provides the minimum and maximum execution times observed, the standard deviation (SD) for execution time, the minimum and maximum speedups achieved, and the percentage difference between Max Speedup and Min Speedup (% Diff). Table I also shows the type of the program -- Memory-intensive (Mem) or CPU-intensive (CPU). In the above experiment, the number of application threads (OPT threads) created for each application was chosen such that it maximized the observed performance on our 24-core machine.

As shown in Table I, most of the programs exhibit significant performance variation. For example, the standard deviation of the performance of streamcluster is 10.2. Since the width of one standard deviation is about 68% in a normal distribution, standard deviation of 10.2 means that there is 32% chance that the performance will lie beyond + or - 4.8% of the mean. Minimizing performance variation while simultaneously maximizing performance is clearly beneficial. Elimination of performance variation has another advantage. Many optimization techniques for improving performance or optimizing power consumption [Merkel et al. 2010; Zhuravlev et al. 2010; Dhiman et al. 2010; Pusukuri et al. 2011] on multicore machines rely on performance monitoring data. The presence of high variation in performance degrades the accuracy of the information collected and the benefits of the optimization techniques. Moreover, due to high performance variation we must use multiple runs of target programs for collecting average performance values. However, with low performance variation one may collect the same quality information via fewer runs.

To find the causes of performance variation, we conducted a performance variation study of 15 multithreaded programs. Intuitively, the cause of performance variation of an application depends upon the kind of resources the application requires. Since these 15 programs mainly exploit CPU and Main memory (not I/O), in this paper we conduct performance variation studies by considering different OS memory allocation

and scheduling policies and then identify those policies that lead to high performance and low performance variation.

An OS scheduler migrates threads from one core to another core to balance the load across the cores. However, thread migrations are expensive events as they cause a thread to pull its working set into cold caches, often at the expense of other threads [McDougall and Mauro 2006]. The impact of thread migrations on the performance of *memory-intensive* programs is significant because of their large working sets, high cache miss rates due to thread migrations, and non-uniform memory access latency. The negative impact of thread migrations on performance variation is worse when improper memory allocation policies are used. For example, memory-intensive programs experience high cache miss-rate and variation in cache miss-rate due to migrations when the default *next-touch* memory allocation policy is used.

For *CPU-intensive* programs, thread context-switches significantly impact the performance. Thread context-switches can be involuntary or voluntary. Involuntary context-switches (ICX) happen when a thread is taken off a core due to expiration of their time quantum or preemption by a higher priority thread. Voluntary context switches (VCX) are result of blocking system calls (e.g., for I/O) or when the thread fails to acquire a lock. CPU-intensive programs exhibit a higher ICX-rate in comparison to memory-intensive programs. With the default Time Share (TS) scheduling policy, priorities of threads change very frequently to adjust corresponding time-quantum for balancing load and providing fairness in scheduling. The ICX and also the variation in ICX-rate increases due to frequent changes in priorities with TS scheduling. The variation in ICX-rate significantly impacts the performance variation of CPU-intensive programs.

In summary, the *variation in cache miss-rate* is the major cause of performance variation of memory-intensive programs and the *variation in ICX-rate* is the major cause of performance variation of CPU-intensive programs. Other events such as, kernel intervention, system processes, and voluntary context-switches further increase performance variation. In this work, we focus on reducing the negative impact of thread migrations on the performance without preventing thread migrations because for a machine with large number of cores (e.g., a 24 core machine), thread migrations lead to better performance (see Section 3.2.1). Consider the model in which we bind one thread per core and thus eliminate thread migrations. This approach is effective for small number of cores but not for machines with large number of cores. From extensive experimentation on our 24 core machine we found that with one thread per core model most programs perform significantly worse in comparison to default policies that permit thread migrations.

Based upon the above observations we develop the Thread Tranquilizer framework that uses simple utilities available on modern operating systems to dynamically monitor application behavior and adapt execution to reduce performance variation. Thread Tranquilizer uses cputrack(1) utility to monitor cache miss-rate and mpstat(1) utility to monitor ICX-rate [McDougall et al. 2006]. Based on these two events, Thread Tranquilizer dynamically applies proper memory allocation and scheduling policies to achieve high performance and low variation in performance. The overhead of Thread Tranquilizer is negligible and it requires no changes to the application source code or the OS kernel.

The remainder of the paper consists of the following. The causes of performance variation are identified in section 2 through an in-depth performance analysis of several multithreaded programs on a 24-core multicore system. We show that the combination of Random or Round-Robin memory allocation and Fixed-Priority (FX) scheduling policies simultaneously reduces performance variation and improves performance of memory-intensive programs. The FX policy simultaneously reduces performance variation and improves performance of CPU-intensive programs. In section 3 we present Thread

Tranquilizer, a framework that uses simple utilities available on modern operating systems to monitor cache miss-rate and ICX-rate of a target program and, based on these events, it dynamically applies appropriate memory allocation and scheduling policies that can yield up to 98% (on average 68%) reduction in the performance variation and up to 43% (on average 15%) improvement in the performance over the default policies of OpenSolaris. On Linux, Thread Tranquilizer simultaneously reduces performance variation up to 91% and improves performance up to 53%. Related work and conclusions are given in sections 4 and 5.

2. PERFORMANCE VARIATION STUDY

The reasons for performance variation of an application depend upon the kind of resources it uses. In this work we use benchmark programs that stress mainly CPU and main memory. Therefore, we analyze OS policies that affect the usage of CPU and memory hierarchy. We study discuss how different memory allocation policies along with thread migrations affect the performance of memory-intensive programs and then study the effect of CPU scheduling policies on the performance of CPU-intensive programs. In our discussion cache miss-rate refers to the last-level cache miss-rate. Before presenting our results we describe the experimental setup used.

2.1. Experimental Setup

2.1.1. Target Machine and Benchmarks. Our experimental setup consists of a Dell PowerEdge R905 server whose configuration is shown in Table II. As we can see this machine has 24 cores. It is a ccNUMA machine and remote access takes about 2.5 times more time than local access in this machine.

Dell^{TM} PowerEdge R905:
24 Cores:
4 $\times$ 6-Core 64-bit AMD Opteron 8431 Processors (2.4 GHz);
L1 : 128 KB; Private to a core; L2 : 512 KB; Private to a core;
L3 : 6144 KB; Shared among 6 cores; Memory: 32 GB RAM;

Table II: Target Machine.

We use a total of 15 benchmark programs: eight programs are from PARSEC [Bienia et al. 2008] suite including *streamcluster, facesim, canneal, x264, fluidanimate, swaptions, ferret, and bodytrack*; five programs are from SPEC OMP [SPECOMP 2001] suite including *swim, wupwise, equake, gafort, and mgrid*; *TATP* [TATP 2009] is a database transaction application; and *SPECjbb2005* [SPECjbb 2005]. The implementations of PARSEC programs which are based upon *pthreads* are run on *native inputs*, SPEC OMP programs are run on medium input data sets, and SPECjbb2005 is run with a single JVM. TATP (a.k.a NDBB and TM-1) uses a 10000 subscriber data set of size 20 MB with a solidDB [solidDB 2010] engine. None of the applications is IO-intensive. In this work we ran each experiment 10 times and we present standard deviation results for the ten runs. We could not use some of the PARSEC and SPEC OMP programs due to the following reasons. Since Thread Tranquilizer takes around 5 seconds for one pass, the running-time of the input program should be larger than five seconds. However, in blackscholes and dedup PARSEC programs, the main thread runs for most of the time, and worker threads are run for a fraction of time. We were unable to compile raytrace and vips of PARSEC, and fma3d, art, applu, ammp of SPEC OMP on OpenSolaris. The remaining programs (galgel, apsi) of SPEC OMP do not scale beyond 6 threads.

Tuning the programs: Since we are conducting a study on a machine with 24 cores, we first examined the programs to see if their implementations require any tuning consistent with the use of large number of cores. We manually tuned the programs using

the following three techniques before using them for performance variation analysis. These tuning techniques significantly improved performance and reduced performance variation. The performance of the tuned versions of programs is considered as the baseline in the evaluation of Thread Tranquilizer.

(1) *Lock-contention and libmtmalloc*: For programs that make extensive use of heap memory, to avoid the high overhead of *malloc*, we used the *libmtmalloc* library to allow multiple threads to concurrently access to heap [Attardi and Nadgir 2003]. We also studied the impact of memory allocators libumem [Benson 2003] and libhoard [Berger et al. 2000] on performance and found that libmtmalloc is better.

(2) *TLB misses and Larger Page*: Larger page sizes for the processor's memory management unit allows more efficient use of the TLB, ultimately resulting in improved application performance [Boyd-Wickizer et al. 2010]. Larger pages minimize TLB misses, then consequently minimizes kernel intervention to serve the TLB misses, and thus, improves performance and reduces performance variation. Since our machine only supports 2MB pages along with the default 4KB pages, we used 2MB pages for tuning *memory-intensive* programs.

(3) *Balancing input load across worker threads*: In some of the applications the input load is not distributed evenly across the worker threads. In particular, when the load is not divisible by the number of threads, the extra load is assigned to a single thread. This uneven distribution of load increases performance variation. We improved the load distribution code so that the load could be more evenly spread across the worker threads. For example, consider swaptions program from PARSEC with input load of 128 swaptions. In the original code when 24 threads are used the input load of 128 swaptions is distributed as follows: five swaptions each for 23 threads; and 13 swaptions for the 24^{th} thread. We improved the input-load distribution code so that it assigns six swaptions each to eight threads and five swaptions each to the remaining 16 threads. This simple modification simultaneously reduced performance variation and improved performance.

2.1.2. Operating System. We carried out the performance variation study using OpenSolaris.2009.06 as it provides a rich set of tools to examine and understand the behavior of programs. The *memory placement optimization* feature and *chip multithreading* optimizations allow OpenSolaris to support hardware with asymmetric memory hierarchies, such as cache coherent NUMA systems and systems with chip-level multithreading and multiprocessing. To capture the distance between different CPUs and memories, a new abstraction called "locality group (lgroup)" has been introduced in OpenSolaris. Lgroups are organized into a hierarchy that represents the latency topology of the machine [McDougall and Mauro 2006].

2.2. Thread Migrations & Memory Allocation Policies

The OS scheduler migrates threads from one core to another core to balance the load across the cores. Thread migrations are expensive as they cause a thread to pull its working set into cold caches, often at the expense of other threads [McDougall and Mauro 2006]. Moreover, on NUMA machines, the negative impact of thread migrations is even higher because of variation in memory-latency. Therefore, the negative impact of thread migrations on performance of memory-intensive applications is even higher.

To understand the impact of thread migrations on our machine, we created two single threaded micro benchmarks -- one program is CPU-intensive as it executes arithmetic operations in a loop and another program is memory-intensive as it creates several large arrays using malloc and then reads and writes to them. We ran these programs for the same amount of time (nearly 16 seconds) 100 times in two different

configurations: no-migration configuration where we bind the program to only one core; and allow-migration configuration where we give the control of this program to the OS scheduler without binding it to a core so that thread migrations under default policies are possible.

We used DTrace scripts [Cantrill et al. 2004; McDougall et al. 2006] to find the number of migrations experienced by these two programs and also to measure the average time a thread takes to migrate. In Figure 1, the Table shows the average and standard deviations of the execution time and the average number of migrations per run for these programs. Using DTrace scripts we find that a thread migration takes on average around 100 μs on our machine. Therefore, the migration cost from the OS side is around 420 μs for the CPU-intensive program, but the program experiences the overhead around 8000 μs. However, the system overhead due to thread migrations with the memory-intensive program is around 330 μs, but the memory-intensive program experiences the overhead of around 459000 μs. This simple experiment clearly shows that the impact of thread migration on the performance and performance variation of memory-intensive program is much higher in comparison to CPU-intensive program.

Config.	Program	Type	Time	SD	#Mig.
1	CPU-intensive	No Binding	16379	0.04	4.2
2	CPU-intensive	Binding	16371	0.01	**0.0**
3	Memory-intensive	No Binding	16687	0.67	3.3
4	Memory-intensive	Binding	16228	0.17	**0.0**

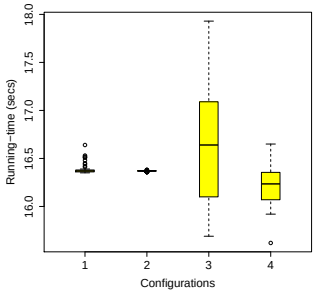


Fig. 1: The impact of thread migration on CPU-intensive and memory-intensive single threaded micro-benchmarks. The running-time (Time) is in milliseconds. Standard deviation (SD) of the running-times and the average number of thread migrations (last column) per run are presented in the table.

Figure 1 also shows that the memory-intensive program experiences significantly larger performance variation compared with the CPU-intensive program. To minimize thread migration overhead and preserve locality awareness, Opensolaris tries to migrate the threads among the cores belonging to the same chip. Actually in this experiment, the thread migrations happened among the cores of the same chip. However, when a program number of threads greater than number of cores is running, we can expect the OS to migrate threads from one chip to another chip to balance the load across the cores. This will further increase the migration cost and degrade the speedups.

2.2.1. Next-Touch (the default policy). The memory allocation policies significantly affect the impact of thread migrations on performance. The key to delivering performance on a NUMA system is to ensure that physical memory is allocated close to the threads that are expected to access it [McDougall and Mauro 2006]. The next-touch policy is based on this fact and it is the default policy on OpenSolaris. Thus, memory allocation defaults to the home lgroup of the thread performing the memory allocation. Under next-touch policy a memory-intensive thread can experience high memory latency overhead and high cache miss-rate and most importantly high variance in cache miss-rate when it is started on one core and migrated to another core which is not in its home locality group. This also leads to HyperTransport traffic which degrades performance due to high variation in memory latency of a NUMA system. Moreover, lock-contention, IO, and memory-demanding behavior cause an increase in thread migrations. The thread

migrations cause changes in thread priorities which further increases the variation in performance.

2.2.2. Random and Round-Robin Policies. While next-touch is the default memory allocation policy for private memory (heap and stack), the *random* allocation policy is the default policy for shared memory with explicit sharing when the size of shared memory is beyond the default threshold value 8MB [McDougall and Mauro 2006]. This threshold is set based on the communication characteristics of Message Passing Interface (MPI) programs [McDougall and Mauro 2006]. Therefore, it is not guaranteed that the random policy will be always applied to the shared memory for multithreaded programs that are based on pthreads and OpenMP. If the shared memory is less than 8MB, then the next-touch policy is the default also for the shared memory. More importantly, programs with huge private memory (e.g., the heap size for facesim is around 306MB) will dramatically benefit from Random/RR policies rather than the default next-touch policy. Instead of using the next-touch policy for private memory, we use Random or Round-robin (RR) policies for both the private and the shared memory for memory-intensive multithreaded programs. Table III lists these memory allocation policies [McDougall and Mauro 2006].

Policy	**Description**	**Short name**
LGRP_MEM_POLICY_NEXT	next to allocating thread's home lgroup (Next-Touch)	NEXT
LGRP_MEM_POLICY_ROUNDROBIN	round robin across all lgroups	RR
LGRP_MEM_POLICY_RANDOM	randomly across all lgroups	RANDOM

Table III: Memory Allocation Policies.

RR policy allocates a page from each leaf lgroup in round robin order. Random memory allocation just picks a random leaf lgroup to allocate memory for each page. Therefore, both RR and Random policies eventually allocate memory across all the leaf lgroups and then the threads of memory intensive workloads get a chance to reuse the data in both private and shared memory. This reduces cache-miss rate and memory latency penalty. These policies optimize for bandwidth while trying to minimize average latency for the threads accessing it throughout the NUMA system [McDougall and Mauro 2006]. They spread the memory across as many memory banks as possible, distributing the load across many memory controllers and bus interfaces, thereby preventing any single component from becoming a performance-limiting hot spot. Moreover, random placement improves the reproducibility of performance measurements by ensuring that relative locality of threads and memory remains roughly constant across multiple runs of an application [McDougall and Mauro 2006]. Therefore, RR or Random policies minimize cache miss-rate and more importantly *variation in cache miss-rate*.

2.3. Dynamic Priorities and ICX

The main goal of a modern general-purpose OS scheduler is to provide *fairness*. Since, it is not guaranteed that all the threads of a multithreaded program behave similarly at any moment (e.g., due to differences in accessing resources such as CPU, Memory, Lock data structures, Disk), we can expect the OS scheduler to make frequent changes to thread priorities to maintain an even distribution of processor resources among the threads. By default, the OS scheduler prioritizes and runs threads on a time-shared basis as implemented by the TS Scheduler Class. The adjustments in priorities are made based on the time a thread spends waiting for or consuming processor resources and the thread's time quantum varies according to its priority.

Thread priorities can change as a result of event-driven or time-interval-driven events. Event-driven changes are asynchronous in nature; they include state transitions as a result of a blocking system call, a wakeup from sleep, a preemption, etc. Preemption and expiration of the allotted time-quantum produces involuntary thread context-switches (ICX). Here changing priorities mean updating priority of threads based on their CPU usage and moving them from one priority-class queue to another priority-class queue according to their updated priority. If multiple threads have their priorities updated to the same value, the system implicitly favors the one that is updated first since it winds up being ahead in the run queue. To avoid this unfairness, the traversal of threads in the run queue starts at the list indicated by a marker. When threads in more than one list have their priorities updated, the marker is moved. Thus the order in which threads are placed in the run queue of a core the next time thread priority update function is called is altered and fairness over the long run is preserved [McDougall and Mauro 2006].

Since all the threads of an application do not behave similarly at any moment (e.g., due to their CPU usage, lock-contention time, sleep time etc.), the positions of the threads in run queues are different from one run to another run. The frequent changes in thread priorities produces *variation in ICX-rate* and thus, variation in performance. Moreover, ICX often includes lock-holder thread preemptions and we can expect increase in the frequency of lock-holder preemptions as load increases (i.e., thread count grows). More importantly, whenever a lock-holder thread is preempted, the threads that are spinning for that lock will be blocked, which in turn increases VCX-rate, and leads to poor performance under high loads [McDougall and Mauro 2006].

2.3.1. Fixed Priority Scheduling. The Fixed Priority (FX) scheduling class [McDougall and Mauro 2006] attempts to solve the issue of frequent thread ping-ponging with TS class. Threads execute on a CPU until they block on a system call, are preempted by a higher-priority thread that has become runnable, or they have used up their time quantum. The allotted time quantum varies according to the scheduling class and the priority of the thread. OS maintains time quanta for each scheduling class in an object called a dispatch table. Threads in the fixed priority class are scheduled according to the parameters in a simple fixed-priority dispatcher parameter table. The parameter table contains a global priority level and its corresponding time quantum. Once a process is at a priority level it stays at that level at a fixed time quantum. The time quantum value is only a default or starting value for processes at a particular level, as the time quantum of a fixed priority process can be changed by the user with the priocntl(1) command or the priocntl(2) system call. By providing same priority to all the threads of a multithreaded application, FX class dramatically reduces ICX, completely avoids lock-holder thread preemptions, and thus reduces performance variation. Moreover, unlike TS class, only time-driven tick processing [McDougall and Mauro 2006] is done for FX class. This reduces dispatcher locks, minimizes OS intervention, and minimizes performance variation.

Role of Time-quantum. The FX class priority range is 0 to 60, and the corresponding time-quantum range is 200 ms to 20 ms. By default, programs are run with fixed priority level zero (200 ms time-quantum) when we apply FX scheduler class. However, in order to get more benefit of using FX class, we also need to find proper time quantum value for the programs running under FX class. It is appropriate to provide small time quantum for CPU-intensive application threads as these threads heavily compete for CPU resources. In this way no thread will wait for a long time for a CPU. Since threads of a memory-intensive application do not heavily compete for CPU resources, large time-quantum is appropriate for them. Therefore by allocating appropriate time-quantum according to the recourse-usage characteristics of the target multithreaded

program, we can provide fair allocation of CPU cycles for all the threads of the program. To find appropriate time-quantum according the resource usage of applications, first we categorize the applications as memory intensive or CPU intensive. For this we chose a couple of applications from each category (a total of 4 out of 15), and ran them with varying time-quantum ranging from 5 ms to 300 ms. The 4 applications used to find the appropriate time-quanta are: streamcluster, swaptions, ferret, and bodytrack. From extensive experiments, we identified that memory intensive programs with 100 ms time-quantum give better performance and low performance variation and 20 ms time-quantum works well for CPU-intensive programs.

Therefore, we conclude that FX class with appropriate time-quantum reduces ICX-rate, more importantly reduces *variation in ICX-rate*, and thus, reduces performance variation.

2.4. Combination of Memory Allocation and Scheduling Policies

To find the appropriate configuration, we tested the 15 programs in the six configurations shown in Table IV. Figure 2 shows the running-times and cache miss-rates of facesim program (a memory-intensive program) in 10 runs with the six configurations. As shown in the boxplots of Figure 2, there is a reduction in the cache miss-rate and also reduction in the variation of cache miss-rate with the combination of Random (or RR) and FX policies. Therefore, as we can see, the reduction in the variation of cache miss-rate reduced the performance variation. This clearly shows that threads reuse the data from private memory which is spread across the nodes by the RR or Random policy. There is also significant improvement in the performance. As we expected, memory-intensive programs experience low performance variation with improved performance using Random or RR memory allocation policies.

No.	Configuration
1	(NEXT + TS)
2	(RANDOM + TS)
3	(RR + TS)
4	(NEXT + FX)
5	(RANDOM + FX)
6	(RR + FX)

Table IV: Configurations

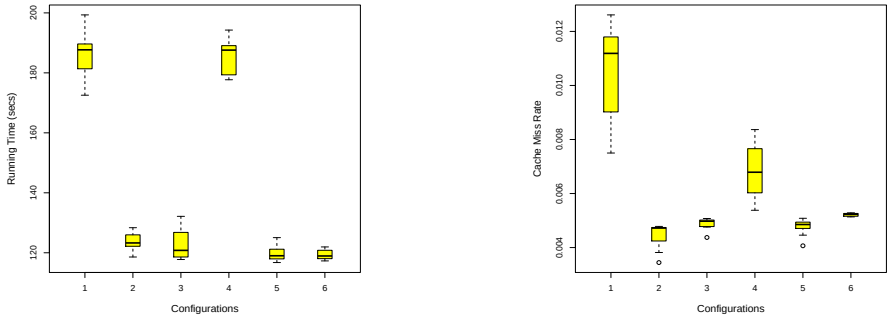


Fig. 2: Running-times and cache miss-rates of facesim (memory-intensive program) in 10 runs. Table IV lists the configurations.

Figure 3 shows the running-times and ICX-rates of mgrid for 10 runs. Mgrid is a CPU-intensive program and scales well in comparison to facesim. As shown in the boxplots of Figure 3, with FX scheduling policy the variation in the ICX-rate is reduced and thus there is a reduction in the variation of running-times. As we expected there is no significant impact of Random and RR policies on the performance of mgrid.

Figure 4 and 5 show that the combination of Random and FX policies reduces performance variation and improves performance simultaneously for memory-intensive programs. Figure 6 shows that FX scheduling policy reduces performance variation and improves performance simultaneously for CPU-intensive programs. There is no significant impact of memory allocation policies on CPU-intensive programs. FX is very

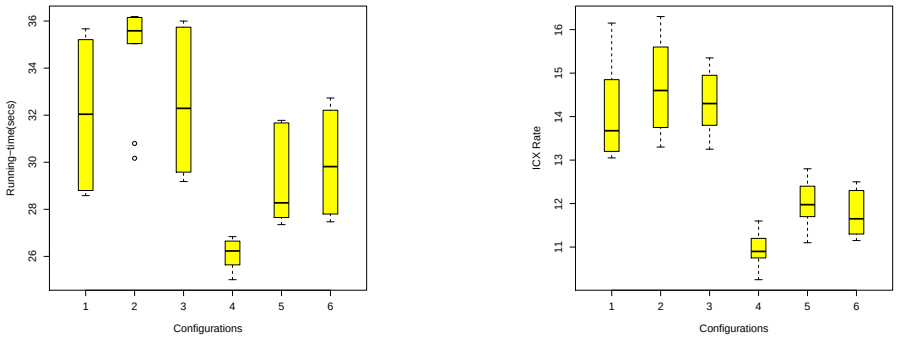


Fig. 3: Running-times and ICX Rates of mgrid (CPU-intensive program) in 10 runs. Table IV lists the configurations.

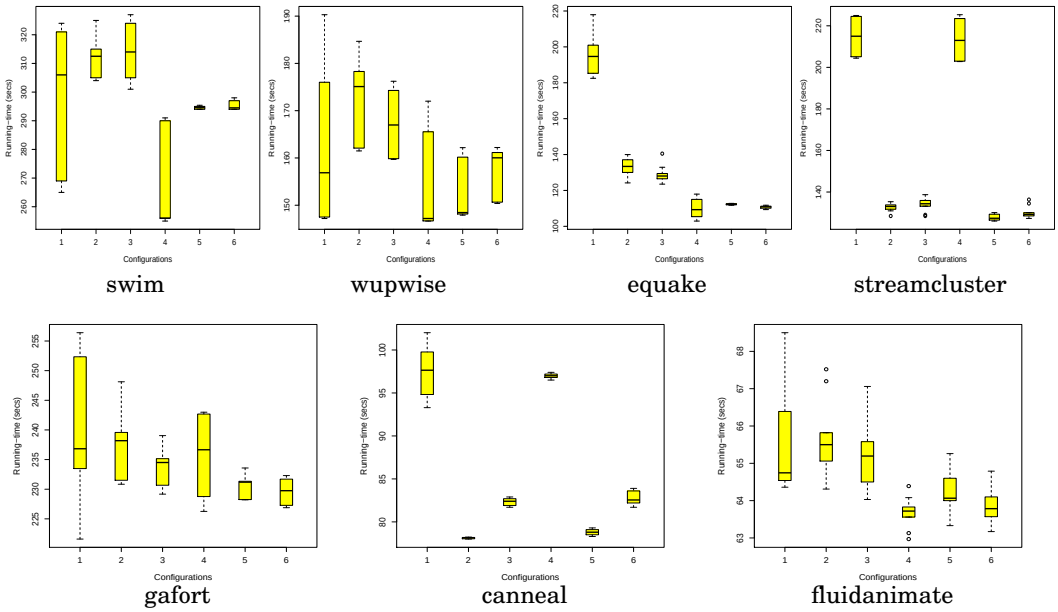


Fig. 4: Performance variation of memory-intensive programs is reduced with the combination of Random memory allocation and FX scheduling policies.

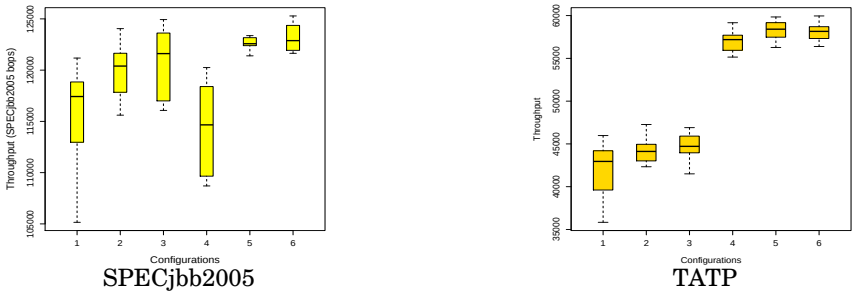


Fig. 5: Performance variation of SPECjbb2005 and TATP is reduced with the combination of Random and FX policies. Performance (**throughput**) is also improved.

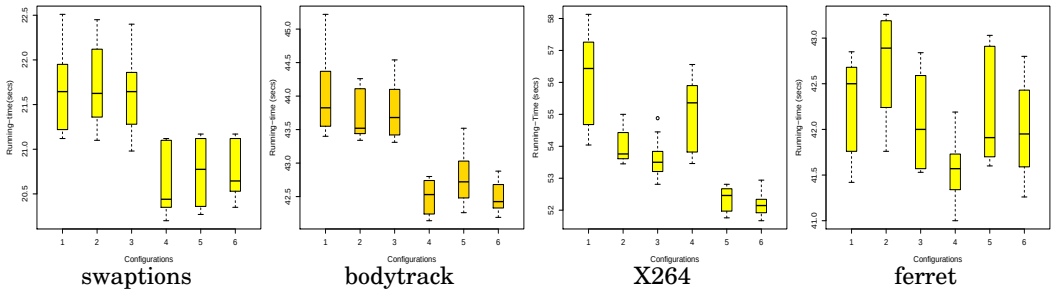


Fig. 6: Performance variation of CPU-intensive programs is reduced with FX scheduling policy. As we expected, there is no significant effect of Random or RR policies on CPU-intensive programs.

effective for programs with high lock-contention. Since swaptions, x264, and ferret are CPU-intensive have low lock-contention, FX slightly improves their performance. However, the performance variation with (FX + Next) is low compared to (TS + Next) for these three programs. Since bodytrack is a CPU-intensive and high lock-contention, the variation with (FX + Next) is significantly lower compared to (TS + Next). Moreover, among the five CPU-intensive programs, mgrid benefits significantly from FX scheduling policy. This is because the tuning techniques libmtmalloc and larger page already improved the performance and reduced the performance variation significantly for the other four CPU-intensive programs (swaptions, bodytrack, ferret, and x264). From the above experiments, it is clear that memory-intensive programs get benefit from the combination of Random memory allocation and FX scheduling policies and CPU-intensive programs significantly get benefit only with the FX scheduling policy.

Since the variation in cache miss-rate causes the performance variation of memory-intensive programs, and the variation in ICX-rate cause the performance variation of CPU-intensive programs, in the next section, we present a framework that monitors cache miss-rate and ICX-rate of the target program on line, and based on these events, it dynamically applies proper memory allocation and scheduling policies.

3. THE THREAD TRANQUILIZER FRAMEWORK

Thread Tranquilizer monitors the cache miss-rate and thread ICX-rate of a running program and based on their variation, it dynamically applies appropriate memory allocation and scheduling policies. The execution of the target program is begun with the default Next-Touch and TS policies and the program's miss-rate and ICX-rate is monitored once the worker threads have been created. Thread Tranquilizer takes a maximum of five seconds to complete one pass to select appropriate memory allocation and scheduling policies according to the phase changes of the programs. Therefore, programs with very short running times will not benefit from Thread Tranquilizer. Thus, in this work, we evaluated Thread Tranquilizer with the programs where their worker threads run for more than five seconds.

We used cputrack(1) utility to measure miss-rate and mpstat(1) utility to measure ICX-rate. One second time-interval is the minimum timeout value we could have used with the default implementation of mpstat(1) utility. However, we modified this utility to allow time intervals with millisecond resolution to measure the ICX-rate with 100 ms interval. Therefore, using cputrack(1) utility and the modified mpstat utility, we collect 10 samples of miss-rate and ICX-rate with 100 ms interval, then derive a profile data structure from the 10 samples, which contains average miss-rate, average ICX-rate, and standard deviations of miss-rate and ICX-rate.

Figure 7 shows the state-transition diagram for one pass of Thread Tranquilizer, i.e. in a time-interval of five seconds. If the average miss-rate is greater than the miss-rate

threshold, then we treat the program as a *memory-intensive* and we apply Random memory allocation policy through pmadvise(1) [McDougall and Mauro 2006] utility with proper advice options. Alternatively, we can use the kernel debugger mdb [McDougall and Mauro 2006] utility. We also apply FX policy with 100 ms time-quantum using priocntl(1) utility. To see the effectiveness of these new policies, we again collect a new profile with 10 samples of program's miss-rate and ICX-rate. Since we would like to reduce performance variation without reducing performance, we also consider the average miss-rate along with the standard deviation of miss-rate. If the average miss-rate is less than the previous average miss-rate and the standard deviation of miss-rate is less than the standard deviation of previous miss-rate, we will continue running the program with the new policies. Otherwise, we reset to the default Next-Touch memory allocation policy and apply FX scheduling policy with 100 ms time-quantum. Therefore, Next-Touch vs Random policies are decided for each allocation based on the size of the shared memory requested by the programs.

If the program is *CPU-intensive* (i.e., average miss-rate is less than the Δ miss-rate then we apply only FX scheduling policy with 20 ms time-quantum. To see the effectiveness of the FX policy, we again collect average ICX-Rate and standard deviation of ICX-rate. If the average ICX-rate is less than the previous average ICX-rate and the standard deviation of ICX-rate is less than the previous standard deviation of ICX-rate, then we keep running the target program with the FX policy until the completion of the pass. Otherwise, we reset to the default TS scheduling policy. Thread Tranquilizer uses a daemon thread to continuously monitor the target program and to deal with its phase changes. Every five seconds, a timer sends a signal and the daemon thread catches the signal and repeats the above process to effectively deal with the phase changes of the target program.

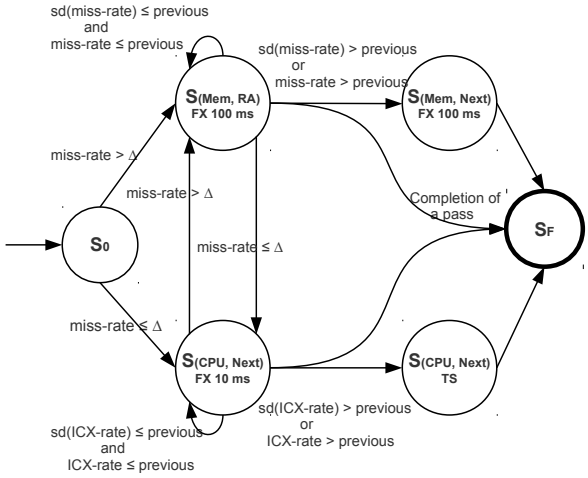
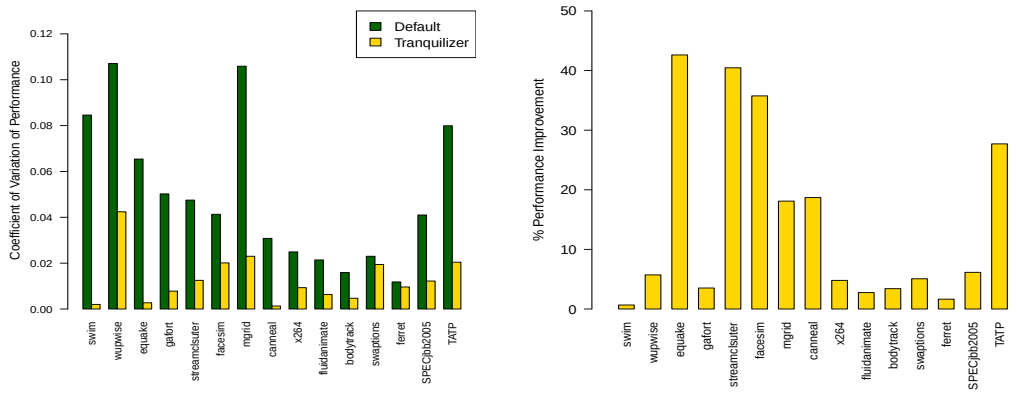


Fig. 7: State-transition diagram shows one pass of Thread Tranquilizer.

Thus, Thread Tranquilizer monitors the target program's miss-rate and ICX-rate online, according to these events it dynamically applies appropriate memory-allocation and scheduling policies, and simultaneously reduces performance variation and improves performance.

3.1. Evaluating Thread Tranquilizer

We evaluated Thread Tranquilizer with the 15 programs described that we used for the above performance variation study. Figure 8 and Table V show that performance variation (coefficient of variation) is reduced and performance is improved simultaneously. As we can see, the combination of Random and FX policies has significant impact on the performance variation of the programs-- memory-intensive programs get benefit from the combination of Random and FX policies and CPU-intensive programs with only FX scheduling policy. Though there is no significant performance improvement for the swim program, the performance variation is reduced dramatically with Thread



Performance Variation is reduced. Performance is improved.

Fig. 8: Performance variation is reduced and performance is improved with Thread Tranquilizer. The bar plot shows another view of the reduction in performance variation (*coefficient of variation*) with Thread Tranquilizer.

No.	Program	Default			Thread Tranquilizer		
		Avg. Time	Speedup	SD (σ)	Avg. Time	Speedup	SD (σ)
1	*swim*	296.6	4.0	**25.1**	294.6	4.1	**0.6**
2	*wupwise*	162.5	8.6	**17.4**	154.2	9.1	**6.5**
3	*equake*	195.7	2.7	**12.8**	112.3	4.8	**0.3**
4	*gafort*	238.9	9.7	**12.0**	230.5	10.1	**1.8**
5	*streamcluster*	214.8	4.0	**10.2**	127.9	6.7	**1.6**
6	*facesim*	186.3	4.8	**7.7**	121.7	7.3	**2.4**
7	*mgrid*	32.1	5.0	**3.4**	26.1	6.1	**0.6**
8	*canneal*	97.4	4.9	**3.0**	79.2	6.0	**0.1**
9	*x264*	56.2	7.6	**1.4**	53.5	8.0	**0.5**
10	*fluidanimate*	65.5	12.8	**1.4**	64.1	13.1	**0.4**
11	*bodytrack*	44.0	11.3	**0.7**	42.5	11.7	**0.2**
12	*swaptions*	21.7	20.8	**0.5**	21.2	21.4	**0.4**
13	*ferret*	42.3	14.9	**0.5**	41.8	15.1	**0.4**
14	*TATP*	42009	6.0	3358	58110	8.4	1186
15	*SPECjbb*	115650	4.5	4741	122762	4.8	1502

Table V: Thread Tranquilizer improves performance and reduces performance variation simultaneously by applying the combination of Random and FX policies. We used standard deviation values to allow the readers to easily map the boxplots (length of the boxplot) with the standard deviation values.

Tranquilizer. As we can see, the performance variation is reduced significantly upto 98% (on average 68%) and also performance is improved upto 43% (on average 15%) with the Framework.

3.1.1. Improves Fairness and Effectiveness Under High Loads. The combination of Random memory allocation and FX scheduling policies improves fairness in scheduling when there are more than one application running. Figure 9 shows that this combination not only reduces the performance variation of individual multithreaded programs, it also reduces the performance variation of the total running-times of the multithreaded programs in concurrent runs. As we can see from the second row of figures in Figure 9, the fluctuation in running-times of the individual programs is low across ten runs in the presence of other programs with the combination of Random and FX policies. That is, in each run, OS allocates resources fairly to all the four programs, and also the total running-time (throughput) is improved. Moreover, as we can see in Fig. 9, the combination of FX and Random policies is very effective under heavy loads. There are a total of 90 threads from the four multithreaded programs running on 24 cores, i.e., over 375% load. Similar performance improvements resulted from several experiments

of running multiple applications with Thread Tranquilizer. We are unable to present those due to lack of space. Thus, Thread Tranquilizer is also effective when there is more than one application running on the system.

3.1.2. Performance Improvements on Linux. On Linux Kernel 2.6.32 (Ubuntu 10.04 LTS server), using numactl(1) utility we applied RR memory allocation policy along with the default next-touch policy to memory-intensive programs. Linux does not support FX scheduling policy and Random policy. As shown in Figure 10, RR policy simultaneously reduces performance variation up to 91% and improves performance up to 53% of memory-intensive workloads.

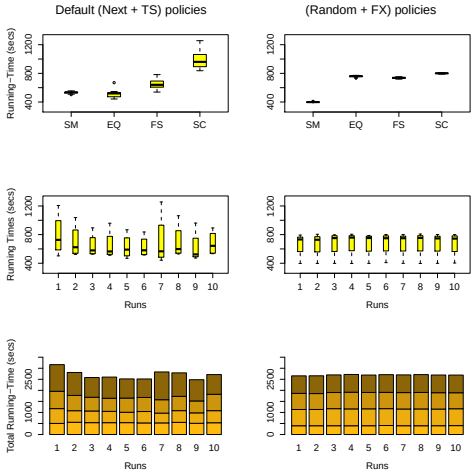
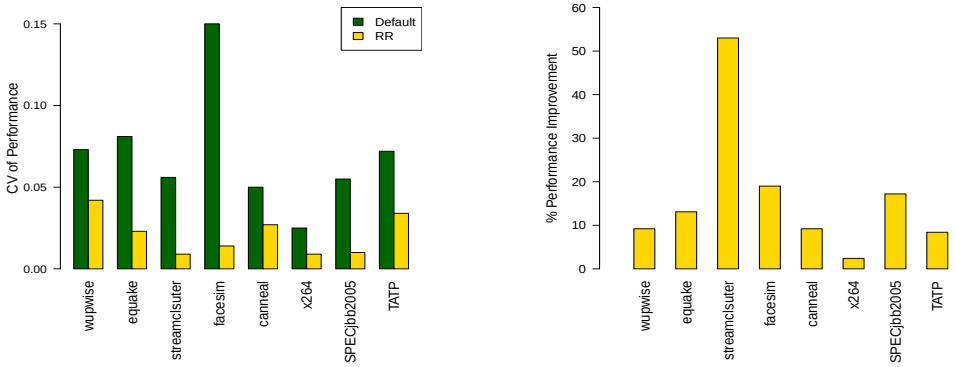


Fig. 9: Thread Tranquilizer is very effective against parallel runs of more than one application. We simultaneously ran four programs swim (SM), equake (EQ), facesim (FS), and streamcluster (SC) 10 times with the configurations of (Next + TS) and (Random + FX) policies.The figures in the first row show the running-times of individual programs in 10 runs, the figures in the second row show the running-times of the four programs in individual runs, and the figures in the third row show the total running-times of the four programs in each run. As we can see the combination of Random and FX policies provide *fairness* relative to the default policies of OpenSolaris.

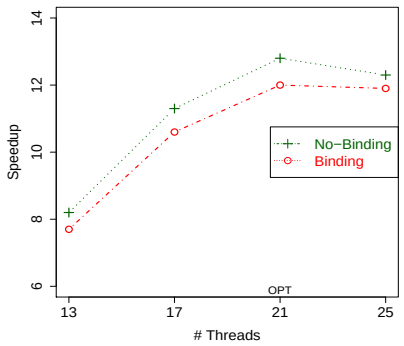
3.2. Discussion

Thread migrations, voluntary context-switches (VCX), and kernel intervention also contribute to performance variation. VCXs are generated because of the lock-contention and the IO properties of the applications. Both lock contention and I/O cause voluntary context switches (VCXs) and thus have the consequence of increasing the thread migration rate. This is because both lock contention and I/O result in *sleep to wakeup* and *run to sleep* state transitions for the threads involved. When a thread wakes up from the sleep state, the OS scheduler immediately tries to give a core to that thread by increasing the priority of the thread. If it fails to schedule the thread on the same core that it used last, it migrates the thread to another core. Moreover, the generation of VCXs is not controlled by the Operating System. However, in order to reduce lock-

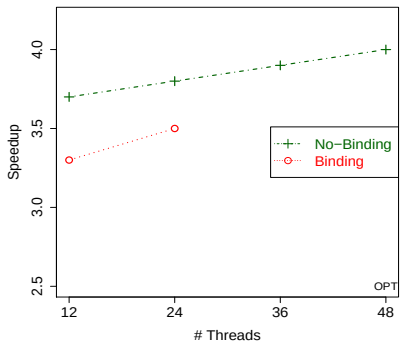


Performance variation is reduced. Performance is improved.

Fig. 10: Performance variation (CV) is reduced up to 91% and performance is improved up to 53% with RR policy on Linux.



fluidanimate.



swim.

Fig. 11: Binding degrades performance.

contention, we tuned the programs above studied with the library libmtmalloc. The library libmtamlloc minimizes lock-contention and consequently minimizes VCX rate and thread migration rate. This tuning technique improved performance of most of the 15 programs and also reduced their performance variation.

3.2.1. Binding Threads vs Performance. We can minimize thread migrations using simple utilities available on modern OS. For example, OpenSolaris provides pbind(1) utility to bind a thread/process to a core (no migrations), and psrset(1) utility to bind a thread/process to a set of cores, i.e., threads can be migrated among the cores of the core-set. We applied both these utilities on all the applicable programs, i.e., OPT threads of these programs is less than or equal to 24. However, only one of these programs benefited and that too only from processor set (psrset) utility. Moreover, OPT threads is greater than 24 for 9 programs out of the 15 programs studied (see Table I). Though pbind(1) reduces thread migrations to zero, the programs experienced significant performance loss. On a machine with few cores, binding of one thread per core may slightly improve performance; but this is not true for larger number core machines [Pusukuri et al. 2011]. As shown in Figure 11, we conducted experiments with one-thread-per-core binding model and observed that for most programs performance is significantly worse. For example, *fluidanimate* performs best with 21 threads without binding on our 24-core machine. When it is run with 21 threads with binding the performance loss is 7%. *swim* performs best with 48 threads without binding on our 24-core machine. If we use one-thread-per-core binding model (24 threads on 24 cores), then performance loss of swim is 14%. Likewise, performance losses of several programs: *ferret*, *swaptions*, *facesim* and *bodytrack* are also significant.

If the OPT threads of an application is greater than the number of cores, it is not reasonable to apply psrset(1) or pbind(1). This is because along with the application threads, there are several high-priority system processes running in the system. Since user does not have control on these system processes, OS scheduler can manage load balancing better than user initiated 'static' thread-to-core mappings. More importantly, "#threads $>$ #cores" leads to several thread-to-core mappings, and therefore exploration of these mappings to find the best one is not a simple task.

We conducted several experiments with different 'threads-to-cores' mappings (e.g., 12 threads on 12 cores set, 18 threads on 18 cores set) and identified that it is useful to use either pbind or psrset when the OPT threads is less than the number of cores. Among the six applicable programs, only one program 'streamcluster' benefits from processor sets (PSET). Figure 12 shows that the performance variation of streamcluster (with OPT Threads of 13) is dramatically reduced with processor set configuration (Next + TS + PSET), where 13 streamcluster threads run on a 12-cores set, i.e. on two NUMA nodes.

However, Thread Tranquilizer (i.e., RA + FX + PSET) configuration further reduces performance variation and improves the performance of streamcluster by around 36% over the default PSET configuration, i.e. (Next + TS + PSET). Here, Thread Tranquilizer applies PSET_RANDOM [McDougall and Mauro 2006] policy.

3.2.2. Improved Scalability. Thread Tranquilizer also improves the scalability of streamcluster. The OPT threads of streamcluster with Thread Tranquilizer is 25 threads with average running time of 105.3 secs on 24 cores which represents a performance improvement of 24.5% over the Random + FX + PSET for which OPT threads is 13. Thread Tranquilizer improves scalability of the most of the programs.

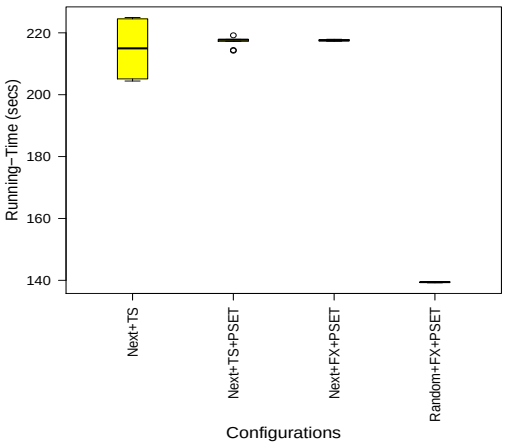


Fig. 12: Performance variation is dramatically reduced with Thread Tranquilizer.

3.2.3. Lessons Learned. Since multicore systems are rapidly taking over the computing world and NUMA architecture solves the scalability problem, we can expect many core machines in future. In order to fully exploit many core machines, modern operating system must therefore evolve with appropriate process scheduling and memory management techniques. More specifically two important challenges to be addressed:

(1) the OS scheduler should adaptively allocate resources such as number of cores according to the resource usage of whole application instead of individual threads of the application-- thread to core mapping should be done at application level instead of thread level.
(2) the OS should adaptively apply appropriate process scheduling and memory allocation policies according to the application characteristics.

Thus, in this work, we show how to reduce performance variation and improve performance simultaneously through proper choice of memory placement and process scheduling policies. We achieved this by implementing Thread Tranquilizer, a framework using simple utilities available on modern operating systems. Thread Tranquilizer monitors cache misses and thread context-switches online, and based on these events, it dynamically applies proper memory placement and scheduling policies that can yield up to 98% reduction in the performance variation and up to 43% improvement in performance over default policies of the OpenSolaris Operating System. The overhead of Thread Tranquilizer is negligible (0.07% of CPU utilization). Moreover there is no need to modify the target application source code or the OS kernel. It is also worth mentioning that since modern architectures and Operating System provide support for performance monitoring events and utilities that we used in this work, Thread Tranquilizer can be easily ported to other systems.

4. RELATED WORK

Many researchers have studied performance variability of parallel applications on large-scale parallel computers. Mraz et al., examined the effect of variance in message passing communications introduced by the AIX Operating System in parallel machines built of commodity work stations [Mraz and Ronald 1994]. They mainly considered the variance

introduced by the interrupts of the AIX Operating System and concluded that globally synchronizing the system clocks gives the best results overall as it generally caused the daemons to run in a co-scheduled fashion and did not degrade system stability. Petrini et al. [Petrini et al. 2003] showed that for large-scale parallel computers, OS noise such as periodically monitoring I/O, could cause serious performance problems. The techniques they proposed to eliminate system noise is to turn off unnecessary system daemons, moving heavyweight daemons to one node instead of spreading them across multiple nodes, etc. In [Gu et al. 2004], Gu et al., demonstrated that a significant source of noise in benchmark measurement in a Java Virtual Machine is due to code layout. Like [Kramer and Ryan 2003], Skinner et al. [Skinner and Kramer 2005] showed that variability in performance is inherently tied to contention for resources between applications and operating system. Hensley et al. [Hensley et al. 2001] minimized performance variation using cpusets on SGI Origin 3800. Most of the above works and some other works [Wright et al. 2009; Evans et al. 2003; Lofstead et al. 2010; Tabatabaee et al. 2005] used MPI based parallel applications on large-scale parallel systems and studied the performance variability caused by network and IO communications. [Gioiosa et al. 2010] also studies OS noise on the performance of MPI based NAS applications running on a quad-core machine, and it provides design and implementation of a scheduler to optimize performance by dynamically binding threads to cores and thus minimizing thread migrations.

Several other researchers investigated the interference of OS on application performance. Ferreira et al. [Ferreira et al. 2008] showed how to quantify the application performance costs due to local OS interference on a range of real-world large-scale applications using over ten thousand nodes. [Tsafrir et al. 2005] also identifies a major source of noise to be indirect overhead of periodic OS clock interrupts, that are used by all general-purpose operating systems as a means of maintaining control. Using simulations, Gupta et al. [Gupta et al. 1991] explored the trade-offs between the use of busy-waiting and blocking synchronization primitives and their interactions with the scheduling strategies. Sandholm et al. [Sandholm and Lai 2009] presented a system for allocating resources in shared data and compute clusters that improves MapReduce job scheduling by regulating user-assigned priorities to offer different service levels to jobs and users over time. In [Shen 2010], Shen provides a characterization of request behavior variations using server applications and finds that the inter-core resource sharing on multicore platforms obfuscates the request execution performance. Some studies [Constantinou et al. 2005; Sweeney and P.F. Duesterwald 2009; Becchi and Crowley 2006] investigated the impact of thread migrations and context-switches on application performance, and some other [Nataraj et al. 2007; De et al. 2007; Mann and Mittaly 2009; Seelam et al. 2010; Beckman et al. 2008] also explored the impact of OS noise on application performance.

Verghese et al. [Verghese et al. 1996] developed page migration and replication policies for ccNUMA systems and showed that migration and replication polices improved memory locality and reduced the overall stall time. [Koita et al. 2000] presents several policies for cluster-based NUMA multiprocessors that are combinations of a processor scheduling scheme and a page placement scheme and investigates the interaction between them through simulations. McCurdy et al. [McCurdy and Vetter 2010] introduced Memphis, a data-centric toolset that uses Instruction Based Sampling to help pinpoint problematic memory accesses to locate NUMA problems in some NAS parallel benchmarks. Alameldeen et al. [Alameldeen and Wood 2003] provided a methodology to compensate for variability, that combines pseudo-random perturbations, multiple simulations and standard statistical techniques; and [Hocko and Kalibera 2010] used page coloring and bin hopping page allocation algorithms to minimize performance variation. Touati et al. [Touati and Worms 2010] proposed a statistical methodology

called 'Speedup-Test' for analyzing the distribution of the observed execution times of benchmark programs and improving the reproducibility of the experimental results. While [Touati and Worms 2010] focuses on statistical methodology for enhancing performance analysis methods to improve the reproducibility of the experimental results for SPEC CPU2006 and SPEC OMP 2001, Thread Tranquilizer reduces performance variation of multithreaded workloads by applying appropriate scheduling and memory allocation policies. Pinter et al. [Pinter and Zalmanovici 2006] developed an extension to the Linux scheduler that exploits inter-task data relations to reduce data cache misses in multi-threaded applications running on SMP machines and improves performance and optimizes energy consumption. Several researchers [Merkel et al. 2010; Zhuravlev et al. 2010; Dhiman et al. 2010; Pusukuri et al. 2011; Blagodurov et al. 2011] have proposed optimization techniques based on use of hardware performance monitoring to find performance bottlenecks and these techniques dynamically improve performance and optimize power consumption of single threaded and multithreaded benchmark programs on multicore machines.

In this work we identified the reasons for performance variation of multithreaded programs. We also show how high performance and low performance variation can be simultaneously achieved through proper choice of memory management and scheduling policies without changing a single line of application code. Moreover, users can easily apply these techniques on the fly and adjust OS environment for multithreaded applications to achieve better performance predictability.

5. CONCLUSIONS

This paper provided an in-depth performance variation analysis of emerging multithreaded programs on a 24-core machine running OpenSolaris.2009.06., and identified the causes of the performance variation with precise details. We identified that the variation in cache miss-rate causes performance variation of memory-intensive programs and the variation in ICX-rate causes performance variation of CPU-intensive programs. Based on these observations, we developed a Framework that monitors cache misses and thread context-switches of a target benchmark program online, based on these events, it dynamically applies proper memory placement and scheduling policies, and simultaneously improves performance on an average by 15% and reduces performance variation on an average by 68% on OpenSolaris. On Linux, Thread Tranquilizer simultaneously reduces performance variation by up to 91% and improves performance by up to 53%. Since modern architectures and OSs provide support for performance monitoring events, our framework can be easily ported to other systems.

REFERENCES

ALAMELDEEN, A. R. AND WOOD, D. A. 2003. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. HPCA '03. IEEE Computer Society, Washington, DC, USA, 7–22.

ATTARDI, J. AND NADGIR, N. 2003. A comparison of memory allocators in multiprocessors. http://developers.sun.com/solaris/articles/multiproc/multiproc.html.

BECCHI, M. AND CROWLEY, P. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers*. CF '06. ACM, New York, NY, USA, 29–40.

BECKMAN, P., ISKRA, K., YOSHII, K., COGHLAN, S., AND NATARAJ, A. 2008. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing 11*, 3–16.

BENSON, R. 2003. Identifying memory management bugs within applications using the libumem library. http://developers.sun.com/solaris/articles/libumem_library.html.

BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS*. 117–128.

BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. PACT '08. ACM, New York, NY, USA, 72–81.

BLAGODUROV, S., ZHURAVLEV, S., DASHTI, M., AND FEDOROVA, A. 2011. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. USENIXATC'11. USENIX Association, Berkeley, CA, USA, 1–1.

BOYD-WICKIZER, S., CLEMENTS, A., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. 2010. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. Vancouver, Canada.

CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. 2004. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC '04. USENIX Association, Berkeley, CA, USA, 2–2.

CONSTANTINOU, T., SAZEIDES, Y., MICHAUD, P., FETIS, D., AND SEZNEC, A. 2005. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News 33*, 80–91.

DE, P., KOTHARI, R., AND MANN, V. 2007. Identifying sources of operating system jitter through fine-grained kernel instrumentation. *Cluster Computing, IEEE International Conference on 0*, 331–340.

DHIMAN, G., MARCHETTI, G., AND ROSING, T. 2010. vgreen: A system for energy-efficient management of virtual machines. *ACM Trans. Des. Autom. Electron. Syst. 16*, 6:1–6:27.

EVANS, J. J., HOOD, C. S., AND GROPP, W. D. 2003. Exploring the relationship between parallel application run-time variability and network performance in clusters. *Local Computer Networks, Annual IEEE Conference on 0*, 538.

FERREIRA, K. B., BRIDGES, P., AND BRIGHTWELL, R. 2008. Characterizing application sensitivity to os interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. SC '08. IEEE Press, Piscataway, NJ, USA, 19:1–19:12.

GIOIOSA, R., MCKEE, S. A., AND VALERO, M. 2010. Designing os for hpc applications: Scheduling. *Cluster Computing, IEEE International Conference on 0*, 78–87.

GU, D., VERBRUGGE, C., AND GAGNON, E. 2004. Code layout as a source of noise in jvm performance. In *In Component And Middleware Performance workshop, OOPSLA*.

GUPTA, A., TUCKER, A., AND URUSHIBARA, S. 1991. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. *SIGMETRICS Perform. Eval. Rev. 19*, 120–132.

HENSLEY, J., ALTER, R., DUFFY, D., FAHEY, M., HIGBIE, L., OPPE, T., WARD, W., BULLOCK, M., AND J.BECKLEHIMER. 2001. Minimizing runtime performance variation with cpusets on the sgi origin 3800. Tech. Rep. 01-32, ERDC MSRC. October.

HOCKO, M. AND KALIBERA, T. 2010. Reducing performance non-determinism via cache-aware page allocation strategies. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. WOSP/SIPEW '10. ACM, New York, NY, USA, 223–234.

KOITA, T., KATAYAMA, T., SAISHO, K., AND FUKUDA, A. 2000. Memory conscious scheduling for cluster-based numa multiprocessors. *J. Supercomput. 16*, 217–235.

KRAMER, W. T. C. AND RYAN, C. 2003. Performance variability of highly parallel architectures. In *Proceedings of the 2003 international conference on Computational science: PartIII*. ICCS'03. Springer-Verlag, Berlin, Heidelberg, 560–569.

LOFSTEAD, J., ZHENG, F., LIU, Q., KLASKY, S., OLDFIELD, R., KORDENBROCK, T., SCHWAN, K., AND WOLF, M. 2010. Managing variability in the io performance of petascale storage systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. IEEE Computer Society, Washington, DC, USA, 1–12.

MANN, P. D. V. AND MITTALY, U. 2009. Handling os jitter on multicore multithreaded systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. IEEE Computer Society, Washington, DC, USA, 1–12.

MCCURDY, C. AND VETTER, J. S. 2010. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *International Symposium on Performance Analysis of Systems and Software*. 87–96.

MCDOUGALL, R. AND MAURO, J. 2006. *Solaris Internals, second edition*. Prentice Hall.

MCDOUGALL, R., MAURO, J., AND GREGG, B. 2006. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Prentice Hall.

MERKEL, A., STOESS, J., AND BELLOSA, F. 2010. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*. EuroSys '10. ACM, New York, NY, USA, 153–166.

MRAZ AND RONALD. 1994. Reducing the variance of point to point transfers in the ibm 9076 parallel computer. In *Proceedings of the 1994 conference on Supercomputing*. Supercomputing '94. IEEE Computer Society Press, Los Alamitos, CA, USA, 620–629.

NATARAJ, A., MORRIS, A., MALONY, A. D., SOTTILE, M., AND BECKMAN, P. 2007. The ghost in the machine: observing the effects of kernel operation on parallel application performance. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. SC '07. ACM, New York, NY, USA, 29:1–29:12.

PETRINI, F., KERBYSON, D. J., AND PAKIN, S. 2003. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. SC '03. ACM, New York, NY, USA, 53–65.

PINTER, S. S. AND ZALMANOVICI, M. 2006. Data sharing conscious scheduling for multi-threaded applications on smp machines. In *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Dresden, Germany, August 28 - September 1, 2006, Proceedings*, W. E. Nagel, W. V. Walter, and W. Lehner, Eds. Lecture Notes in Computer Science Series, vol. 4128. Springer, 265–275.

PUSUKURI, K. K., GUPTA, R., AND BHUYAN, L. N. 2011. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *Proceedings of the 2011 International Symposium on Workload Characterization*.

PUSUKURI, K. K., VENGEROV, D., FEDOROVA, A., AND KALOGERAKI, V. 2011. Fact: a framework for adaptive contention-aware thread migrations. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*. CF '11. ACM, New York, NY, USA, 35:1–35:10.

SANDHOLM, T. AND LAI, K. 2009. Mapreduce optimization using regulated dynamic prioritization. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. SIGMETRICS '09. ACM, New York, NY, USA, 299–310.

SEELAM, S., FONG, L., TANTAWI, A., LEWARS, J., AND J. DIVIRGILIO, K. G. 2010. Extreme scale computing: Modeling the impact of system noise in multicore clustered systems. In *Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*. 1–12.

SHEN, K. 2010. Request behavior variations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. ASPLOS '10. ACM, New York, NY, USA, 103–116.

SKINNER, D. AND KRAMER, W. 2005. Understanding the causes of performance variability in hpc workloads. *IEEE Workload Characterization Symposium 0*, 137–149.

SOLIDDB. 2010. IBM soliddb 6.5 (build 2010-10-04). http://www-01.ibm.com/software/data/soliddb/soliddb/.

SPECJBB. 2005. http://www.spec.org/jbb2005.

SPECOMP. 2001. http://www.spec.org/omp.

SWEENEY, Q. T. AND P.F. DUESTERWALD, E. 2009. Understanding the cost of thread migration for multi-threaded java applications running on multicore platform. In *Proceedings of the Performance Analysis of Systems and Software*. ISPASSS '09. 123–132.

TABATABAEE, V., TIWARI, A., AND HOLLINGSWORTH, J. K. 2005. Parallel parameter tuning for applications with performance variability. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. SC '05. IEEE Computer Society, Washington, DC, USA, 57–69.

TATP. 2009. IBM telecom application transaction processing benchmark description. http://tatpbenchmark.sourceforge.net.

TOUATI, S.-A.-A. AND WORMS, J. S. B. 2010. The speed test. Tech. rep. March. http://hal.archives-ouvertes.fr/inria-00443839.

TSAFRIR, D., ETSION, Y., FEITELSON, D. G., AND KIRKPATRICK, S. 2005. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on Supercomputing*. ICS '05. ACM, New York, NY, USA, 303–312.

VERGHESE, B., DEVINE, S., GUPTA, A., AND ROSENBLUM, M. 1996. Operating system support for improving data locality on cc-numa compute servers. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. ASPLOS-VII. ACM, New York, NY, USA, 279–289.

WRIGHT, N. J., SMALLEN, S., OLSCHANOWSKY, C. M., HAYES, J., AND SNAVELY, A. 2009. Measuring and understanding variation in benchmark performance. In *Proceedings of the 2009 DoD High Performance Computing Modernization Program Users Group Conference*. HPCMP-UGC '09. IEEE Computer Society, Washington, DC, USA, 438–443.

ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. ASPLOS '10. ACM, New York, NY, USA, 129–142.