

Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring

Kishore Kumar Pusukuri, Rajiv Gupta, Laxmi N. Bhuyan

Department of Computer Science and Engineering

University of California, Riverside

Riverside, USA 92521

kishore@cs.ucr.edu, gupta@cs.ucr.edu, bhuyan@cs.ucr.edu

Abstract—It is often assumed that to maximize the performance of a multithreaded application, the number of threads created should equal the number of cores. While this may be true for systems with four or eight cores, this is not true for systems with larger number of cores. Our experiments with PARSEC programs on a 24-core machine demonstrate this. Therefore, dynamically determining the appropriate number of threads for a multithreaded application is an important unsolved problem. In this paper we develop a simple technique for dynamically determining appropriate number of threads without recompiling the application or using complex compilation techniques or modifying Operating System policies. We first present a scalability study of eight programs from PARSEC conducted on a 24 core Dell PowerEdge R905 server running OpenSolaris.2009.06 for numbers of threads ranging from a few threads to 128 threads. Our study shows that not only does the maximum speedup achieved by these programs vary widely (from 3.6x to 21.9x), the number of threads that produce maximum speedups also vary widely (from 16 to 63 threads). By understanding the overall speedup behavior of these programs we identify the critical Operating System level factors that explain why the speedups vary with the number of threads. As an application of these observations, we develop a framework called “Thread Reinforcer” that dynamically monitors program’s execution to search for the number of threads that are likely to yield best speedups. Thread Reinforcer identifies optimal or near optimal number of threads for most of the PARSEC programs studied and as well as for SPEC OMP and PBZIP2 programs.

I. Introduction

With the widespread availability of multicore systems a great deal of interest has arisen in developing techniques for delivering performance on multicore systems. Towards this end many studies are being conducted to study the performance of multithreaded workloads such as PARSEC on small scale multicore machines [1]–[3], [5], [24]. Since performance of a multithreaded application depends upon the number of threads used to run on a multi-core system, finding appropriate number of threads for getting best performance is very important. Using few threads leads to under utilization of system resources and using too many threads degrades application performance because of lock-contention and contention of shared resources. One simple off-line method is to run the application with different number of threads and choose the appropriate number of threads that gives best performance. However, this is time-consuming, does not work if the number of threads is input dependent, does not adapt to the system’s dynamic behavior, and therefore is not a practical solution. It is often assumed

that to maximize performance the number of threads should equal the number of cores [1]–[3], [5], [24], [28]. While this is true for systems with 4 cores or 8 cores, it is not true for systems with larger number of cores (see Table III).

On a machine with few cores, binding [12], [24], [28] (one-thread-per-core model) with $\#threads == \#cores$ may slightly improve performance; but this is not true for machines with larger number cores. As shown in Figure 1, we conducted experiments with one-thread-per-core binding model and observed that for most programs performance is significantly worse. For example, *swaptions* performs best with 32 threads without binding on our 24-core machine. When it is run with 24 threads without binding the performance loss is 9% and with binding the performance loss is 17%. Likewise, *ferret* performs best with 63 threads without binding on our 24-core machine. If we use one-thread-per-core binding model, then performance loss of *ferret* is 54%. Performance losses of *facesim* and *bodytrack* are also significant. More significantly, memory-intensive and high lock-contention programs experience severe performance degradation with binding on large number core machines. The problem with binding is that bounded thread may not get to run promptly. Thus, dynamically finding a suitable number of threads for a multi-threaded application to optimize system resources in a multi-core environment is an important open problem.

The existing dynamic compilation techniques [6] for finding appropriate number of threads are quite complex. In [6] authors noted that the Operating System (OS) and hardware likely cannot infer enough information about the application to make effective choices such as determining how many number of threads an application should leverage. However, since the modern OSs have a rich set of tools available to examine and understand the programs, using these tools, we present a simple framework that dynamically finds the suitable number of threads by observing OS level factors and show that the number of threads suggested by the algorithm achieves near optimal speedup. Our approach does not require recompilation of the application or modifications to OS policies.

To understand the complex relationship between number of threads, number of available cores, and the resulting speedups for multithreaded programs on machines with larger number of cores, we first conduct a performance study of eight PARSEC programs on a 24 core Dell PowerEdge R905 machine running OpenSolaris.2009.06. We study the performance of these

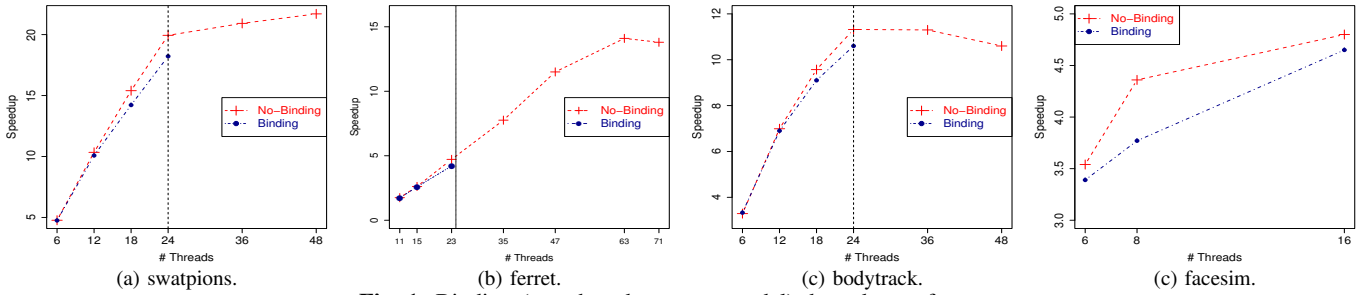


Fig. 1: Binding (one-thread-per-core model) degrades performance.

programs for different numbers of threads ranging from a few threads to 128 threads. Based on the scalability analysis of these programs, we identify the OS level factors that explain why an application has the best performance with only a certain number of threads. Based on the observations, we develop a framework called ‘Thread Reinforcer’ to dynamically monitor program’s execution and OS level factors for finding optimum number of threads that are expected to yield the best speedup. While Thread Reinforcer is developed based on the observations of PARSEC programs, it is also tested against different multithreaded programs SPEC OMP and PBZIP2 programs. The overhead of Thread Reinforcer is very low and it can be easily ported to any modern operating systems. The key contributions of this work are as follows:

- Our study of PARSEC shows that not only does the maximum speedup achieved by these programs vary widely (from 3.6x for *canneal* to 21.9x for *swaptions*), the number of threads that produce maximum speedups also vary widely (from 16 threads for *facesim* to 63 threads for *ferret*). We observe that for five out of eight programs the maximum speedup results from creating more threads than the number of cores.
- While degree of parallelism, availability of additional cores, and degree of lock contention were found to play an important role in limiting performance, we also identified additional factors. When the number of threads is less than the number of cores the context switch rate plays an important role. When the number of threads is greater than the number of cores thread migrations performed by the OS can limit the speedups.
- Based upon the above observations, we develop a framework called ‘Thread Reinforcer’ for determining the appropriate number of threads to create for an application. The framework monitors the above factors at runtime and uses the observations to guide the search for determining the appropriate number of threads. The numbers of threads determined by this algorithm are near optimal for most of the applications from PARSEC, SPEC OMP, and PBZIP.

The remainder of this paper is organized as follows. Section II presents the experimental setup, the speedups for eight PARSEC programs for varying number of threads, and studies the causes that limit the speedups. Section III presents Thread Reinforcer framework for automatically finding number of threads. Related work and conclusions are given in Sections 4 and 5.

TABLE I: Target Machine and OS.

Dell™ PowerEdge R905:
24 Cores:
4 × 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;
Operating System: OpenSolaris 2009.06

TABLE II: PARSEC Programs Used: n is a command line argument that determines the number of threads created.

Program	Number of Threads Created
<i>swaptions</i>	(main + workers) $1 + n$
<i>ferret</i>	(main + 6-stage pipeline) $1 + (1 + n + n + n + n + 1)$
<i>bodytrack</i>	(3-stage pipeline) $1 + n + 1$
<i>blackscholes</i>	(main + workers) $1 + n$
<i>canneal</i>	(main + workers) $1 + n$
<i>fluidanimate</i>	(main + workers) $1 + n$
<i>facesim</i>	(main + workers) $1 + (n-1)$
<i>streamcluster</i>	(main + workers) $1 + n$

II. Scalability Study and Analysis

A. Experimental Setup

Target Machine and Operating System: Our experimental setup consists of a Dell PowerEdge R905 server whose configuration is shown in Table I. As we can see this machine has 24 cores. We carried out this study using OpenSolaris operating systems as there is a rich set of tools available to examine and understand the behavior of programs running under OpenSolaris. We ran each experiment 10 times and present average results for the ten runs.

Applications Used: In this study we use eight programs from the PARSEC suite – we could not use some of the PARSEC programs because we were unable to get them to compile and run under OpenSolaris. The eight programs used in this work are described in Table II. For each program the number of threads used is also described. Each program takes a command line argument n and then uses it in determining how many threads to create. As we can see from the table, *ferret* and *bodytrack* are pipelined into six and three stages. While the first and last stages consist of one thread each, the intervening stages have n threads each. The rest of the programs consist of a single *main* thread and multiple number of *worker* threads whose number is determined based upon the value of n . By varying the value of n we can run each application using different number of threads. The implementations are based upon *pthread*s and *native inputs* are used in all our experiments. The maximum number of threads was limited to 128 in this work as this was more than sufficient to study the full range of program’s behavior on the 24 core Dell PowerEdge R905 server. We also evaluated the framework against seven SPEC OMP programs and PBZIP2 program.

B. Tuning the Implementation of Applications.

Previous studies of PARSEC have been carried out for machine configurations with a small number of cores (2, 4, or 8). It has been observed that the performance of these programs scales well for a small number of cores. However, since we are conducting a study which considers the scalability of these application programs for larger number of cores, we first examined the programs to see if their implementations require any tuning consistent with the use of a larger number of cores. Our study of the applications revealed two main issues that required tuning of implementations. First, for programs that make extensive use of heap memory, to avoid the high overhead of *malloc* [36], we used the *libmtmalloc* library to allow multiple threads to concurrently access to heap. Second, in some applications where the input load is not evenly distributed across worker threads, we improved the load distribution code.

By tuning the implementations in the above fashion, the performance for seven out of eight applications considered was improved. In some cases the improvements are small (*ferret*, *blackscholes*, *streamcluster*, and *bodytrack*), moderate improvement was observed in case of *fluidanimate*, and very high improvement was observed for *swaptions*. The improvement in *swaptions* can be explained as follows. We observed dramatic reduction in locking events when we switch from *malloc* to *mtmalloc* in a run where 24 worker threads are used. In the original *swaptions* worker thread code the input load of 128 *swaptions* is distributed across 24 threads as follows: five *swaptions* each are given to 23 threads; and 13 *swaptions* are assigned to the 24th thread. This is because the code first assigns equal load to all threads and all remaining load to the last thread. When the number of threads is large, this causes load imbalance. To remove this imbalance, we modified the code such that it assigns six *swaptions* each to eight threads and five *swaptions* each to the remaining 16 threads. This is because instead of assigning the extra load to one thread, we distribute it across multiple threads.

C. Performance for Varying Number of Threads

We ran each program for varying number of threads and collected the speedups observed. Each program was run ten times and speedups were averaged over these runs. Table III shows the maximum speedup (Max Speedup) for each program on the 24-core machine along with the minimum number of

TABLE III: Maximum speedups observed and corresponding number of threads for PARSEC programs on the 24-core machine. Programs were run from a minimum of 4 threads to a maximum of 128 threads.

Program	Tuned Version		Original	
	Max Speedup	OPT Threads	Max Speedup	OPT Threads
<i>swaptions</i>	21.9	33	3.6	7
<i>ferret</i>	14.1	63	13.7	63
<i>bodytrack</i>	11.4	26	11.1	26
<i>blackscholes</i>	4.9	33	4.7	33
<i>canneal</i>	3.6	41	no change	
<i>fluidanimate</i>	12.7	21	12	65
<i>facesim</i>	4.9	16	4.6	16
<i>streamcluster</i>	4.2	17	4.0	17

threads (called OPT Threads) that produced this speedup. The data is provided for both the *tuned version* of the program and the *original version* of the program. As we can see, tuning resulted in improved performance for several programs. In the rest of the paper we will only consider the tuned versions of the program.

As we can see from Table III, not only does the maximum speedup achieved by these programs vary widely (from 3.6x for *canneal* to 21.9x for *swaptions*), the number of threads that produce maximum speedups also varies widely (from 16 threads for *facesim* to 63 threads for *ferret*). Moreover, for the first five programs the maximum speedup results from creating more threads than the number of cores, i.e. OPT-Threads is greater than 24. For the other three programs OPT-Threads is less than the number of cores.

The above observation that the value of OPT-Threads varies widely is significant – it tells us that the choice of number of threads that are created is an important one. Experiments in prior studies involving PARSEC [2], [4], [24] were performed for configurations with a small number of cores (4 and 8). In these studies the number of threads was typically set to equal the number of cores as this typically provided the best performance. However, the same approach cannot be taken when machines with larger number of cores are being used. In other words, we must select appropriate number of threads to maximize the speedups obtained.

To observe how the speedup varies with the number of threads we plot the speedups for all our experiments in Figure 2. The graph on the left shows the speedups for programs for which OPT-Threads is greater than 24 and the graph on the right shows the speedups for the programs for which OPT-

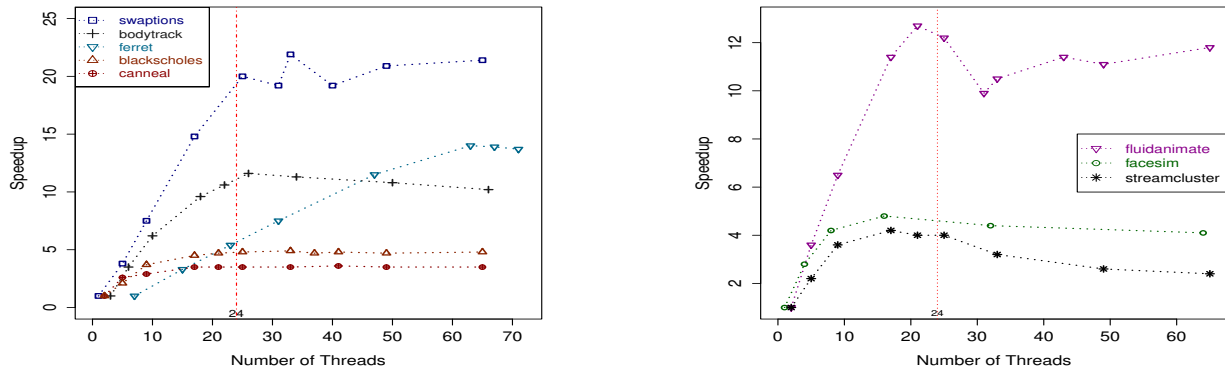


Fig. 2: Speedup behavior of PARSEC workloads for varying number of threads: The graph on the left shows the behavior of applications where maximum speedup was observed for *Number of Threads* > *Number of Cores* = 24; and The graph on the right shows the behavior of applications where maximum speedup was observed for *Number of Threads* < *Number of Cores* = 24.

Threads is less than 24. The programs with OPT-Threads greater than 24 exhibit different behaviors. The speedups for *swaptions* and *ferret* scale well with the number of threads with maximum speedups resulting from use of 33 and 63 threads respectively. While *bodytrack* provides substantial speedups, once maximum speedup of 11.6 is achieved with 26 threads, the speedups starts to fall gradually as more threads are added. The speedups of *blackscholes* and *cannal* increase very slowly with the number of threads due to lack of parallelism in these programs. For programs with OPT-Threads less than 24, once the number of threads reaches OPT-Threads, speedups fall as additional threads are created. This behavior is the result of lock contention that increases with the number of threads.

D. Factors Determining Scalability

In this section we present additional data collected with the aim of understanding the factors that lead to the observed speedup behaviors presented in Figure 2. Using the *prstat* [7] utility, we studied the following main components of the execution times for threads in each application.

- 1) **User:** The percentage of time a thread spends in user mode.
- 2) **System:** The percentage of time a thread spends in processing the following system events: system calls, system traps, text page faults, and data page faults.
- 3) **Lock-contention:** The percentage of time a thread spends waiting for user locks, condition-variables etc.
- 4) **Latency:** The percentage of time a thread spends waiting for a CPU. In other words, although the thread is ready to run, it is not scheduled on any core.

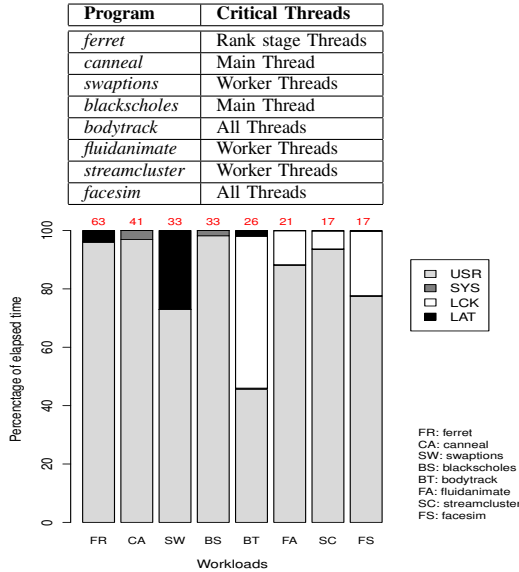


Fig. 3: Breakdown of elapsed time of critical threads.

We studied the above times for all threads to see if changes in these times would explain the changes in speedups observed by varying number of threads. Although we examined the data for all threads, it quickly became apparent that in many programs not all threads were critical to the overall speedup. We identified the *critical threads* and studied them in greater detail. The critical threads for each application are listed in the table below. Figure 3 provides the breakdown of the time of critical threads in the above four categories – this data is for the OPT-Threads run and is the average across all critical threads.

As we can see, in some programs lock-contention (LCK) plays a critical role, in others the threads spend significant time waiting for a CPU as latency (LAT) is high, and the system time (SYS) is the highest for *cannal* and *blackscholes*.

In the remainder of this section we analyze the above times for each of the programs in greater detail to study their relationship with speedup variations that are observed when number of threads is varied. We further identify the program characteristics that are the causes for the observed speedup variations.

1) OPT-Threads > Number of Cores

Scalable Performance. As we can see from the graph on the left in Figure 2, for three programs (*swaptions*, *bodytrack*, and *ferret*) in this category, the speedups scale quite well. As the number of threads is varied from a few threads to around 24, which is the number of cores, the speedup increases linearly with the number of threads. However, once the number of threads is increased further, the three programs exhibit different trends as described below:

- (Erratic) *swaptions*: Although the speedup for *swaptions* can be significantly increased -- from 20 for 25 threads to 21.9 for 33 threads -- its trend is erratic. Sometimes the addition of more threads increases the speedup while at other times an increase in number of threads reduces the speedup.
- (Steady Decline) *bodytrack*: The speedup for *bodytrack* decreases as the number of threads is increased beyond 26 threads. The decline in speedup is quite steady.
- (Continued Increase) *ferret*: The speedup for *ferret* continues to increase linearly. In fact the linear increase in speedup is observed from the minimum number of 6 threads all the way up till 63 threads. Interestingly no change in behavior is observed when the number of threads is increased from less than the number of cores to more than the number of cores.

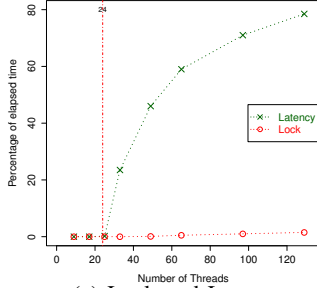
Next we trace the differing behaviors back to specific characteristics of these programs.

swaptions: First let us consider the erratic behavior of speedups observed in *swaptions*. We first examined the lock contention and latency information. As shown in Figure 4(a), the lock contention (LOCK) is very low and remains very low throughout and the latency (LAT) increases steadily which shows that the additional threads created are ready to run but are simply waiting for a CPU (core) to become available. This keeps the execution time to be the same. Therefore we need to look elsewhere for an explanation. Upon further analysis we found that the speedup behavior is correlated to the *thread migration rate*. As we can see from Figure 4(b), when the migration rate goes up, the speedup goes down and vice versa – the migration rate was measured using the *mpstat* [7] utility. Migrations are expensive events as they cause a thread to pull its working set into cold caches, often at the expense of other threads [7]. Thus, the speedup behavior is a direct consequence of changes in thread migration rate.

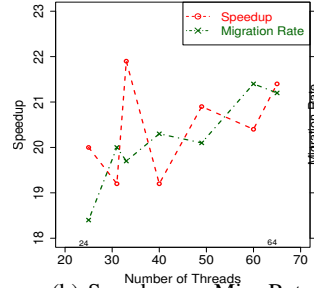
The OS scheduler plays a significant role here as it is responsible for making migration decisions. When a thread makes a transition from sleep state to a ready-to-run state, if

TABLE IV: Behavior of *ferret*.

Total Threads	n	Load (1)			Segment (n)		Extract (n)		Vector (n)		Rank (n)		Out (1)		Speedup
		USR	SYS	LOCK	USR	LOCK	USR	LOCK	USR	LOCK	USR	LOCK	USR	LOCK	
15	3	22	4	74	8	92	1	99	44	56	100	0	0.5	99.3	3.3
31	7	44	7.8	48	6.7	93	1	99	43	57	100	0	0.6	99	7.5
47	11	56	11.3	32	5.4	95	1	99	40	60	100	0	0.7	99	11.5
55	13	64	14	19	5	95	1	99	44	56	98	0	0.7	99	12.5
63	15	79	20	0	5	95	1	99	43	57	96	0	0.7	99	14.1
71	17	77	20	0	5	95	1	99	37	63	80	16	0.7	99	13.8
87	21	78	17	0	4	96	1	99	28	72	65	33	0.4	99.3	13.7
103	25	75	17	0	3	97	1	99	24	76	53.5	45	0.4	99.3	13.4
119	29	74	17	0	3	97	1	99	20	80	46	52.5	0.4	99.4	13.2
127	31	70	20	0	3	97	1	99	19	81	40	59	0.4	99.4	13.1



(a) Lock and Latency



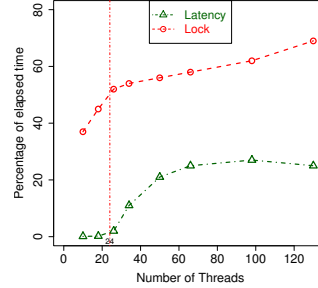
(b) Speedup vs. Mig. Rate.

Fig. 4: *swaptions*: Cause of Erratic Speedup Changes.

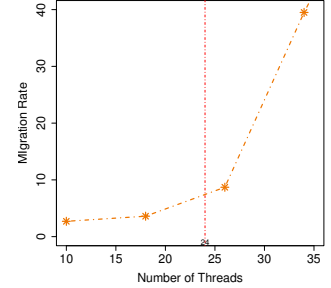
the core on which it last ran is not available, the thread is likely to be migrated to another available core. In general, one would expect more migrations as the number of threads increases beyond the number of cores. However, if the number of threads is divisible by the number of cores, then the likelihood of migrations is less compared to when this is not the case. In the former case, the OS scheduler can allocate equal number of threads to each core, balancing the load, and thus reducing the need for migrations. Thus we conclude that *variations in degree of load balancing* across cores causes corresponding variations in thread migration rate and hence the observed speedups. For example, in Figure 4(b), the thread migration rate for 48 threads on 24 cores is lower than thread migration rate for 40 threads on 24 cores. Moreover, we can expect low thread migration rate when the input load (128 swaptions) is perfectly divisible by the number of threads (e.g., 16, 32, 64 etc.).

bodytrack: Next let us consider the steady decline in speedup observed for *bodytrack*. Figure 5(a) shows that although the latency (LAT) rises as more threads are created, so does the lock contention (LOCK) which is significant for *bodytrack*. In addition, *bodytrack* is an I/O intensive benchmark where I/O is performed by all the threads. We observed that this program produces around 350 *ioctl()* calls per second. Both lock contention and I/O 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, if it fails to schedule the thread on the same core that it used last, it migrates the thread to another core. As we can see from Figure 5(b), the thread migration rate for *bodytrack* rises with the number of threads which causes a steady decline in its speedup.

ferret: The behavior of this program is interesting as the speedup for it increases linearly starting from 6 threads to all the way up to 63 threads even though only 24 cores are



(a) Lock and Latency



(b) Migration Rate.

Fig. 5: *bodytrack*: Cause of Decline in Speedup.

available. To understand this behavior we need to examine the program in greater detail. The program is divided into six pipeline stages – the results of processing in one stage are passed on to the next stage. The stages are: Load, Segment, Extract, Vector, Rank, and Out. The first and last stage have a single thread but each of the intermediate stages are a pool of n threads. Between each pair of consecutive stages a queue is provided through which results are communicated and locking is used to control queues accesses.

The reason for the observed behavior is as follows. The *Rank* stage performs most of the work and thus the speedup of the application is determined by the *Rank* stage. Moreover the other stages perform relatively little work and thus their threads together use only a fraction of the compute power of the available cores. Thus, as long as cores are not sufficiently utilized, more speedup can be obtained by creating additional threads for the *Rank* stage. The maximum speedup of 14.1 for *ferret* was observed when the total number of threads created was 63 which actually corresponds to 15 threads for *Rank* stage. That is, the linear rise in speedup is observed from 1 thread to 15 threads for the *Rank* stage which is well under the total of 24 cores available – the remaining cores are sufficient to satisfy the needs of all other threads.

The justification of the above reasoning can be found in the data presented in Table IV where we show the average percentage of USR and LOCK times for all stages and SYS time for only *Load* stage because all other times are quite small. The threads belonging to *Segment*, *Extract*, and *Out* stages perform very little work and mostly spend their time waiting for results to become available in their incoming queues. While the *Load* and *Vector* stages do perform significant amount of work, they nevertheless perform less work than the *Rank* stage. The performance of the *Rank* stage determines the overall speedup – adding additional threads to the *Rank* stage continues to yield additional speedups as long as this stage does not experience lock contention. Once lock contention times start to rise (starting at $n = 17$), the speedup begins to fall.

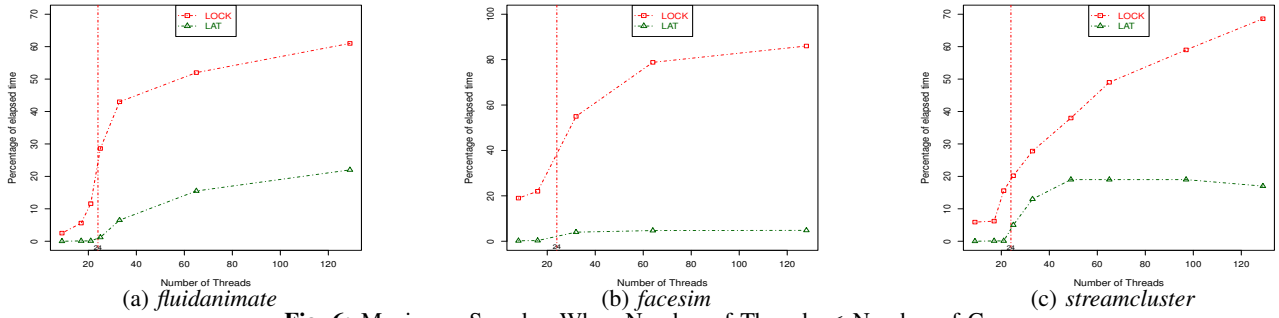


Fig. 6: Maximum Speedup When Number of Threads < Number of Cores.

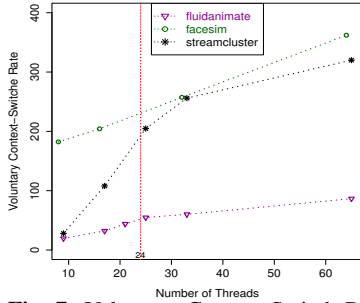


Fig. 7: Voluntary Context Switch Rate.

To further confirm our observations above we ran an experiment in which we increased the number of threads in the *Rank* stage and lowered the number of threads in other intermediate stages. We found that the configuration with (1, 10, 10, 10, 16, 1) threads gave a speedup of 13.9 and when we changed the configuration to (1, 16, 16, 16, 16, 1) threads the speedup remained the same. This further confirms the importance of the *Rank* stage.

Performance Does Not Scale. (*blackscholes* and *cannal*) Although the maximum speedups of these programs (4.9 and 3.6) are observed when 32 and 40 worker threads are created, the speedups of both these programs increase very little beyond 16 worker threads. This is because most of the work is performed by the main thread and the overall CPU utilization becomes low. The main thread takes up 85% and 70% of the time for *blackscholes* and *cannal* respectively. During rest of the time the parallelized part of the program is executed by worker threads. The impact of parallelization of this limited part on the overall speedup diminishes with increasing number of threads.

2) OPT-Threads < Number of Cores

The three programs where maximum speedup was achieved using fewer threads than number of cores are *fluidanimate*, *facesim*, and *streamcluster*. In these programs the key factor that limits performance is lock contention. Figure 6 shows that the time due to lock contention (LOCK) dramatically increases with number of threads while the latency (LAT) shows modest or no increase. The maximum speedups are observed at 21 threads for *fluidanimate*, 16 threads for *facesim*, and 17 threads for *streamcluster*.

When the number of threads is less than the number of cores, the load balancing task of the OS scheduler becomes simple and thread migrations become rare. Thus, unlike *swaptions* and *bodytrack* where maximum speedups were observed for greater than 24 threads, thread migration rate does not play any role in

TABLE V: Voluntary vs. Involuntary Context Switches.

Program	VCX (%)	ICX (%)
<i>fluidanimate</i>	84	16
<i>facesim</i>	97	3
<i>streamcluster</i>	94	6
<i>swaptions</i>	11	89
<i>ferret</i>	13	87

the performance of the three programs considered in this section. However, the increased lock contention leads to slowdowns because of increased *context switch rate*. We can divide context-switches into two types: involuntary context-switches (ICX) and voluntary context-switches (VCX). Involuntary context-switches happen when threads are involuntarily taken off a core (e.g., due to expiration of their time quantum). Voluntary context-switches occur when a thread performs a blocking system call (e.g., for I/O) or when it fails to acquire a lock. In such cases a thread voluntarily releases the core using the *yield()* system call before going to sleep using *lwp_park()* system call. Therefore as more threads are created and lock contention increases, VCX context switch rate rises as shown in Figure 7. It is also worth noting that most of the context switches performed by the three programs are in the VCX category. We measured the VCX and ICX data using the *prstat* utility. Table V shows that the percentage of VCX ranges from 84% to 97% for the three programs considered here. In contrast, the VCX represents only 11% and 13% of context switches for *swaptions* and *ferret*.

Since the speedup behavior of an application correlates with variations in LOCK, MIGR_RATE, VCX_RATE, and CPU_UTIL, in the next section we develop a framework for automatically determining the number of threads by runtime monitoring of the above characteristics.

III. The Thread Reinforcer Framework

The applications considered allow the user to control the number of threads created using the command line argument *n* in Table II. Since our experiments show that the number of threads that yield peak performance varies greatly from one program to another, the selection of *n* places an added burden on the user. Therefore, in this section, we develop a framework for automatically selecting the number of threads.

The framework we propose runs the application in two steps. In the first step the application is run multiple times for short durations of time during which its behavior is monitored and based upon runtime observations Thread Reinforcer searches for the appropriate number of threads. Once this number is found, in the second step, the application is fully reexecuted with the number of threads determined in the first step. We

TABLE VI: Factors considered wrt to the number of threads.

Factor	≤ 24 Threads	> 24 Threads
LOCK	Yes	Yes
VCX_RATE	Yes	-
MIGR_RATE	-	Yes
CPU_UTIL	Yes	Yes

have to rerun the applications for short durations because the applications are written such that they do not support varying of number of threads online. Thus, Thread Reinforcer does not consider phase changes of the target program. However, out of the 16 programs tested, only the *ammp* program shows two significantly different phases and its first phase dominates the execution. Therefore Thread Reinforcer works well also for the *ammp* program.

Each time an application is to be executed on a new input, Thread Reinforcer is used to determine the appropriate number of threads for that input. This is done in order to handle applications whose runtime behavior is input dependant and thus the optimal number of threads may vary across inputs. Our goal is twofold: to find the appropriate number of threads and to do so quickly so as to minimize runtime overhead. The applications we have considered take from tens of seconds to a few hundred seconds to execute in the OPT-Threads configuration. Therefore, we aim to design Thread Reinforcer so that the times it takes to search for appropriate number of threads is only a few seconds. This ensures that the benefits of the algorithm outweigh the runtime overhead of using it.

Thread Reinforcer searches for appropriate number of threads in the range of T_{min} and T_{max} threads as follows. It runs the application for increasing number of threads for short time durations. Each successive run contains either T_{step} or $T_{step}/2$ additional threads. The decision of whether or not to run the program for higher number of threads and whether to increase the number of threads by T_{step} or $T_{step}/2$, is based upon changes in profiles observed over the past two runs. The profile consists of four components: *LOCK* (lock contention), *MIGR_RATE* (thread migration rate), *VCX_RATE* (voluntary context switch rate), and *CPU_UTIL* (processor utilization). The values of each of these measures are characterized as either *low* or *high* based upon set *thresholds* for these parameters. Our algorithm not only examines the current values of above profiles, it also examines how rapidly they are changing. The changes of these values over the past two runs are denoted as $\Delta LOCK$, $\Delta MIGR_RATE$, ΔVCX_RATE , and ΔCPU_UTIL . The changes are also characterized as *low* and *high* to indicate whether the change is gradual or rapid. At any point in the most penultimate run represents the current best solution of our algorithm and the last run is compared with the previous run to see if it should be viewed as an improvement over the penultimate run. If it is considered to be an improvement, then the last run becomes our current best solution. Based upon the strength of improvement, we run the program with T_{step} or $T_{step}/2$ additional threads. The above process continues as long as improvement is observed. Eventually Thread Reinforcer terminates if no improvement or degradation is observed, or we have already reached the maximum number of threads T_{max} .

Table VI identifies the components that play an important role when the number of threads is no more than the number of cores (i.e., 24) versus when the number of threads is greater

than the number of cores. The lock contention is an important factor which must be considered throughout. However, for ≤ 24 threads the *VCX_RATE* is important while for > 24 threads the *MIGR_RATE* is important to consider. In general, the limit of parallelism for a program may reach at any time. Thus *CPU_UTIL* is an important factor to consider throughout. The above observations are a direct consequence of our observations made during the study presented earlier.

Figure 8 presents Thread Reinforcer in detail. Thread Reinforcer is initiated by calling *FindN()* and when it terminates it returns the value of command like parameter n that is closest to the number of threads that are expected to give the best performance. *FindN()* is iterative – it checks for termination by calling *Terminate()* and if termination conditions are not met, it calls *ComputeNextT()* to find out the number of threads that must be used in the next run. Consider the code for *Terminate()*. It first checks if processor utilization has increased from the penultimate run to the last run. If this is not the case then the algorithm terminates otherwise the lock contention is examined for termination. If lock contention is high then termination occurs if one of the following is true: lock contention has increased significantly; number of threads is no more than the number of cores and voluntary context switch rate has sharply increased; or number of threads is greater than the number of cores and thread migration rate has sharply increased. Finally, if the above termination condition is also not met we do not terminate the algorithm unless we have already reached the upper limit for number of threads. Before iterating another step, the number of additional threads to be created is determined. *ComputeNextT()* does this task – if the overheads of locking, context switches, or migration rate increase slowly then T_{step} additional threads are created; otherwise $T_{step}/2$ additional threads are created.

We implemented Thread Reinforcer to evaluate its effectiveness in finding appropriate number of threads and study its runtime overhead. Before experimentation, we needed to select the various thresholds used by Thread Reinforcer. To guide the selection of thresholds we used three of the eight programs: *fluidanimate*, *facesim*, and *blackscholes*. We ran these selected programs on *small* inputs: for *fluidanimate* and *blackscholes* we used the *simlarge* input and for *facesim* we used the *simsmall* input. We studied the profiles of the programs and identified the threshold values for *LOCK*, *MIGR_RATE*, *VCX_RATE*, *CPU_UTIL* as follows. The threshold values were chosen such that after reaching the threshold value, the value of the profile characteristic became more sensitive to the number of threads and showed a rapid increase. There are two types of threshold values: absolute thresholds and Δ thresholds. The Δ threshold indicates how rapidly the corresponding characteristic is changing. For *LOCK* and *VCX_RATE* both thresholds are used by our algorithm. For *MIGR_RATE* and *CPU_UTIL* only Δ threshold is used. It should be noted that the three programs that were chosen to help in selection of thresholds collectively cover all four of the profile characteristics: for *fluidanimate* both *LOCK* and *MIGR_RATE* are important; for *facesim* *VCX_RATE* is important; and for *blackscholes* *CPU_UTIL* is important.

TABLE VII: Algorithm vs. Optimal (PARSEC programs).

Program	Number of Threads		Speedups	
	Algorithm	Optimal	Algorithm	Optimal
<i>facesim</i>	16	16	4.9	4.9
<i>bodytrack</i>	26	26	11.4	11.4
<i>swaptions</i>	33	33	21.9	21.9
<i>ferret</i>	63	63	14.1	14.1
<i>fluidanimate</i>	25	21	12.2	12.7
<i>streamcluster</i>	25	17	4.0	4.2
<i>canneal</i>	9	41	2.9	3.6
<i>blackscholes</i>	9	33	3.7	4.9

TABLE X: Algorithm vs. Optimal (Other programs).

Program	Number of Threads		Speedups	
	Algorithm	Optimal	Algorithm	Optimal
<i>ammp</i>	24	24	11.8	11.8
<i>art</i>	32	32	8.8	8.8
<i>fma3d</i>	16	20	5.5	5.7
<i>gafort</i>	64	48	9.7	9.8
<i>mgrid</i>	16	16	5.0	5.0
<i>swim</i>	32	24	3.9	4.0
<i>wupwise</i>	24	24	8.6	8.6
<i>pbzip2</i>	24	28	6.7	6.9

In our experiments we ran all eight PARSEC programs considered in this work and all the programs were run on *native inputs* – note that the thresholds were selected by running only three programs using small inputs. The number of threads was varied in the range of 8 (T_{min}) to 72 (T_{max}) threads and T_{step} was set to 8 threads. The time interval for which an application was profiled in each iteration of our algorithm was set to 100 milliseconds beyond the initial input reading phase of each application. The profiling utilities *prstat* and *mpstat* by default use a 1 second interval, i.e. this is the minimum timeout value we could have used with the default implementation. To minimize the runtime overhead of our algorithm we wanted to use smaller time intervals. Therefore we modified these utilities to allow time intervals with millisecond resolution.

Table VII presents the number of threads found by Thread Reinforcer and compares it with the OPT-Threads number that was reported earlier in the paper. The corresponding speedups for these number of threads are also reported. From the results we can see that for the first four programs (*facesim*, *bodytrack*, *swaptions*, *ferret*) the number of threads found by our algorithm is exactly the same as OPT-Threads. For the next two programs, *fluidanimate* and *streamcluster*, the numbers are close as they differ by $T_{step}/2 (= 4)$ and $T_{step} (= 8)$ respectively. The loss in speedups due to this suboptimal choice of the number of threads is quite small. For the last two programs, *canneal* and *blackscholes*, the number of threads Thread Reinforcer selects is much smaller than OPT-Threads.

Table VIII shows the conditions which caused the termination of Thread Reinforcer. For six of these programs lock contention, migration rate, and voluntary context switch rate play an important role. For the other two programs the programs terminate because improvement in the CPU_UTIL is small and hence Thread Reinforcer terminates assuming that there is no more parallelism in the application. The termination condition for *canneal* and *blackscholes* explains why the number of threads selected by our algorithm differs greatly from the OPT-Threads value. The speedup of these programs rises very slowly and thus the change in CPU_UTIL is quite low. For the threshold setting we have used Thread Reinforcer simply concludes that there is no longer any need to add threads as

TABLE VIII: Termination Conditions for PARSEC Programs.

Program	Condition
<i>facesim</i>	VCX_RATE
<i>bodytrack</i>	MIGR_RATE
<i>swaptions</i>	LOCK
<i>ferret</i>	LOCK
<i>fluidanimate</i>	LOCK
<i>streamcluster</i>	MIGR_RATE
<i>canneal</i>	CPU_UTIL
<i>blackscholes</i>	CPU_UTIL

TABLE XI: Termination Conditions for Other Programs.

Program	Condition
<i>ammp</i>	MIGR_RATE
<i>art</i>	LOCK
<i>fma3d</i>	LOCK
<i>gafort</i>	CPU_UTIL
<i>mgrid</i>	VCX_RATE
<i>swim</i>	CPU_UTIL
<i>wupwise</i>	CPU_UTIL
<i>pbzip2</i>	CPU_UTIL

TABLE IX: Search Overhead (seconds) for PARSEC programs.

Program	T_{search}	$T_{parallel}$	Percentage
<i>canneal</i>	0.5	131	0.4%
<i>facesim</i>	1.1	186	0.6%
<i>blackscholes</i>	0.5	85	0.6%
<i>streamcluster</i>	3.2	226	1.4%
<i>fluidanimate</i>	1.5	69	2.2%
<i>ferret</i>	1.3	41.9	3.1%
<i>bodytrack</i>	1.6	43.8	3.7%
<i>swaptions</i>	0.9	21.3	4.2%

TABLE XII: Search Overhead (seconds) for Other programs.

Program	T_{search}	$T_{parallel}$	Percentage
<i>ammp</i>	0.9	267.1	0.3%
<i>art</i>	1.3	62.8	2.1%
<i>fma3d</i>	0.7	23	3.0%
<i>gafort</i>	1.6	238.9	0.7%
<i>mgrid</i>	0.7	32.1	2.2%
<i>swim</i>	1.3	302.4	0.4%
<i>wupwise</i>	1.2	162.5	0.7%
<i>pbzip2</i>	1.1	201.3	0.6%

there is no more parallelism to exploit.

Finally we consider the search overhead of Thread Reinforcer for PARSEC programs. Table IX shows the times for the search and parallel execution for each of the programs. As we can see from the table, the search times vary from 0.5 seconds to 3.2 seconds while the parallel execution times of the programs range from 21.9 seconds to 226 seconds. The final column shows the search time as a percentage of parallel execution time for each program. The programs are listed in increasing order of this percentage value. For the first three programs this percentage is extremely small – ranging from 0.4% to 0.6%. For the remaining programs these percentages are between 1.4% and 4.2%. Thus, the runtime overhead of our algorithm is quite small. Therefore it can be used to select the number of threads when an application is run on a new input.

A. Thread Reinforcer Against Other Programs

Since Thread Reinforcer uses the thresholds of PARSEC programs, we would like to see how Thread Reinforcer works for programs other than PARSEC. For this, we tested Thread Reinforcer against seven SPEC OMP programs and PBZIP2 program, a total of eight other programs. Table X lists the programs, and also presents the number of threads found by Thread Reinforcer and compares it with the OPT-Threads number.

Table XI shows the conditions which caused the termination of Thread Reinforcer. For four of these programs CPU utilization (no more parallelism to exploit), and for other four programs lock contention, migration rate, and voluntary context switch rate play an important role. Table XII shows that the search overhead is very low compared to the parallel execution-time of the programs. Therefore, Thread Reinforcer can be used to find the optimum number of threads of multithreaded applications, moreover this experiment shows that the thresholds are broadly applicable.

B. Limitations

Initialization Period: An important limitation of the current implementation of Thread Reinforcer is that it works well for applications that have short initialization period (i.e., creation of worker threads early in the execution is observed). Since the current implementation of the applications studied do not allow

us to assign arbitrary number of threads at run-time, we need to rerun the applications for finding the optimum number of threads. Therefore, if the initialization period of an application is too long then the search overhead increases and benefits of

- *Convert()* converts number of threads into command line param. value;
- Profile $P = \langle CPU_UTIL, LOCK, VCX_RATE, MIGR_RATE \rangle$;
- (T_{best}, N_{best}) is the current best solution; P_{best} is its profile;
- (T_{try}, N_{try}) is the next solution tried; P_{try} is its profile;
- $\Delta P_{field} = P_{try}.field - P_{best}.field$;
- *low* returns true/false if $P_{try}.field$ or ΔP_{field} is low/not low;
- *high* returns true/false if $P_{try}.field$ or ΔP_{field} is high/not high;
- T_{step} is increments in which number of threads is increased;
- T_{min}/T_{max} is minimum/maximum number of threads allowed;

```

FindN() {
    Tbest ← Tmin; Nbest ← Convert(Tbest);
    Pbest ← Collect profile for 100 milliseconds run with parameter Nbest.

    Ttry ← Tmin + Tstep; Ntry ← Convert(Ttry);
    Ptry ← Collect profile for 100 milliseconds run with parameter Ntry.

    loop
        if Terminate(Pbest, Ptry) = true then
            return(Nbest)
        else
            Tbest ← Ttry; Nbest ← Ntry; Pbest ← Ptry
            Ttry ← ComputeNextT(Pbest, Ptry);
            Ntry ← Convert(Ttry);
            Ptry ← Collect profile for 100 milliseconds run with parameter Ntry.
        endif
    endloop
}

ComputeNextT(Pbest, Ptry) {
    if Ttry ≤ NumCores then
        if low(Ptry.LOCK) or low(Ptry.ΔVCX_RATE) or
           (high(Ptry.LOCK) and low(Ptry.ΔLOCK))
        then
            ΔT = Tstep
        else
            ΔT = Tstep/2
        endif
    else – Ttry > NumCores
        if low(Ptry.LOCK) or low(Ptry.ΔMIGR_RATE) then
            ΔT = Tstep
        else
            ΔT = Tstep/2
        endif
    endif
    return( minimum(Ttry + ΔT, Tmax) )
}

Terminate(Pbest, Ptry) {
    – terminate if no more parallelism was found
    if low(Ptry.ΔCPU_UTIL) then return(true) endif

    – terminate for high lock contention, VCX rate, and migration rate
    if high(Ptry.LOCK) then
        if high(Ptry.ΔLOCK) or
           Ttry ≤ NumCores and high(Ptry.ΔVCX_RATE) or
           Ttry > NumCores and high(Ptry.ΔMIGR_RATE)
        then
            return(true)
        endif
    endif

    – terminate if no more threads can be created
    if Ttry = Tmax then
        Tbest ← Ttry; Nbest ← Ntry; return(true)
    endif

    – otherwise do not terminate
    return(false)
}

```

Fig. 8: *FindN()* returns the best value for command line parameter, N_{best} , which corresponds to the appropriate number of threads determined for running the application. It is an iterative algorithm that calls *Terminate* to see if it is time to terminate the search with current value of N_{best} or whether to increase the number of threads to the number returned by *ComputeNextT()*.

Thread Reinforcer will decline. However, the implementation can be adapted such that we rerun the application from the starting point of the worker threads invocation.

Phase Changes: Thread Reinforcer does not consider phase changes of the target program. However, out of 16 programs tested, only *ammp* program shows two significantly different phases and its first phase dominates the whole execution. Thus, Thread Reinforcer works well for it also.

Cache Sharing: Since the focus of our work is on the effect of OS level factors on the scalability of multithreaded programs, other factors such as architectural factors including cache sharing is outside the scope of this work.

IV. Related Work

Dynamically finding a suitable number of threads for a multi-threaded application to optimize performance in a multi-core environment is an important problem. While this issue has been studied in context of quad-core and 8-core systems, it has not been studied for systems with larger number of cores. When number of cores is small it is often recommended, that number of threads created equal the number of cores. In contrast our work demonstrates that on a 24 core system many of the PARSEC programs require much more than 24 threads to maximize speedups (e.g., *ferret* requires 63 threads to get the highest speedup).

Controlling Number of Threads. In [6], Lee et al. show how to adjust number of threads in an application dynamically to optimize system efficiency. They develop a run-time system called “Thread Tailor” which uses dynamic compilation to combine threads based on the communication patterns between them in order to minimize synchronization overhead and contention of shared resources (e.g., caches). They achieve performance improvements for three PARSEC programs on quad-core and 8-core systems. However, they used a baseline of number of threads equals the number of cores (4 or 8) for performance comparisons and they did not present the optimal number of threads resulting from their technique.

To improve performance and optimize power consumption for OpenMP based multi-threaded workloads, Suleman et al. [8], proposed a framework that dynamically controls number of threads using runtime information such as memory bandwidth and synchronization. They show that there is no benefit of using larger number of threads than the number of cores. Similarly, Nieplocha et al. [9] demonstrate that some applications saturate shared resources as few as 8 threads on an 8-core Sun Niagara processor. Curtis-Maury et al., [28] predict efficient energy-concurrency levels for parallel regions of multithreaded programs using machine learning techniques. Thus once again the above works considered small number of cores and used one-thread-per-core binding model.

Jung et al. [10], presented performance estimation models and techniques for generating adaptive code for quad-core SMT multiprocessor architectures. The adaptive execution techniques determine an optimal number of threads using dynamic feedback and run-time decision runs. Similarly, Kunal et al. [20] proposed an adaptive scheduling algorithm based on the feedback of parallelism in the application. Many other works that dynamically control number of threads are aimed at

studying power performance trade-offs [11]–[13], [25], [27]. Unlike the above, Barnes et al. [33] presented regression techniques to predict parallel program scaling behavior (processor count). Singh et al., [25] presented scalability prediction models of parallel applications on a multicore system.

The Effect of OS level factors & Scheduling Techniques. Some studies [21]–[23] investigate the performance implications of thread migrations on multi-core machines. Several researchers [14]–[17] use application performance and concurrency characteristics such as speedup, execution time, synchronization information to make better scheduling decisions for parallel applications. Ferreira et al. [18] 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, and [19] identifies a major source of noise to be indirect overhead of periodic OS clock interrupts, that are used by all modern OS as a means of maintaining control. [26] proposed a hardware-based and system software configurable mechanism to achieve fairness goals specified by system software in the entire shared memory system, and consequently it allows to achieve desired fairness/performance balance. In contrast our work focuses on selecting the number of threads under the default scheduling schemes used by OpenSolaris.

Several previous works [29], [31], [32] consider scheduling techniques based upon different application characteristics (e.g., cache-usage) and dynamic estimates of the usage of system resources. However, these techniques allocate the threads that are provided and do not consider the impact of number of threads on the characteristics of applications. Moreover, the work [6] noted that the OS and hardware likely cannot infer enough information about the application to make effective choices such as determining how many number of threads an application should leverage. However, we developed a simple technique for dynamically determining appropriate number of threads without recompiling the application or using complex compilation techniques or modifying OS policies.

V. Conclusions

It is often assumed that to maximize performance the number of threads created should equal the number of cores. While this may be true for systems with 4 cores or 8 cores, this is not true for systems with significantly larger number of cores. In this paper we demonstrated this point by studying the behavior of several multithreaded programs on a 24 core machine. We studied the factors that limit the performance of these programs with increasing number of threads in detail. Based upon the results of this study, we developed Thread Reinforcer, a framework for dynamically determining suitable number of threads for a given application run. Thread Reinforcer is not only effective in selecting the number of threads, it also has very low runtime overhead in comparison to the parallel execution time of the application.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments. This work is supported in part by NSF grants CCF-0963996, CNS-0810906, CCF-0905509, and CSR-0912850 to the University of California, Riverside.

References

- [1] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.
- [2] M. Bhadauria, V.M. Weaver and S.A. McKee. Understanding PARSEC Performance on Contemporary CMPs? In *IEEE ISWC*, 2009.
- [3] C. Bienia, S. Kumar and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *IEEE ISWC*, 2008.
- [4] A. Bhattacharjee and M. Martonosi. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *PACT*, 2009.
- [5] N. Barrow-Williams, C. Fensch, and S. Moore. A Communication Characterization of SPLASH-2 and PARSEC. In *ISWC*, 2009.
- [6] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications. In *ISCA*, 2010.
- [7] R. McDougall, J. Mauro, and B. Gregg. Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris, Prentice Hall, 2006.
- [8] M. A. Suleman, M. Qureshi, and Y. Patt. Feedback Driven Threading: Power-Efficient and High-Performance Execution of Multithreaded Workloads on CMPs. In *ASPLOS*, 2008.
- [9] J. Nieplocha et al. Evaluating the potential of multithreaded platforms for irregular scientific computations. In *Computing Frontiers*, 2007.
- [10] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for SMT multiprocessor architectures. In *SIGPLAN PPoPP*, 2005.
- [11] Y. Ding, M. Kandemir, P. Raghavan, and M. Irwin. A helper thread based edp reduction scheme for adapting application execution in cmps. In *SPDP*, 2008.
- [12] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *ICS*, 2006.
- [13] J. Li and J. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *HPCA*, 2006.
- [14] E. W. Parsons, K. C. Sevcik. Benefits of speedup knowledge in memory-constrained multiprocessor scheduling. *Performance Evaluation* 27/28, pages 253-272, 1996.
- [15] J. Corbalan, X. Martorell, and J. Labarta. Performance-driven processor allocation. In *OSDI*, 2008.
- [16] S. Hofmeyr, C. Iancu, and F. Blagojevic. Load Balancing on Speed. In *SIGPLAN PPoPP*, 2010.
- [17] A. Snavey, D.M. Tullsen, G. Voelker. Symbiotic Jobscheduling For A Simultaneous Multithreading Processor. In *ASPLOS*, 2000.
- [18] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection. In *ACM/IEEE Supercomputing*, 2008.
- [19] D. Tsafir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications. In *ICS*, 2005.
- [20] Kunal Agrawal and Yuxiong He and Wen Jing Hsu and Charles E. Leiserson. Adaptive Task Scheduling with Parallelism Feedback. In *SIGPLAN PPoPP*, 2006.
- [21] C. Theofanis, S. Yiannakis, M. Pierre, F. Damien, and A. Seznec. Performance Implications of Single Thread Migration on a Chip Multi-Core. *SIGARCH Computer Architecture News*, 33(4):80 - 91, 2005.
- [22] Q.M. Teng, P.F. Sweeney, and E. Duesterwald. Understanding the Cost of Thread Migration for Multi-Threaded Java Applications Running on Multicore Platform. In *IEEE ISPASS*, Dec. 2008.
- [23] K. K. Pusukuri, D. Vengerov, A. Fedorova, and V. Kalogeraki. FACT: a Framework for Adaptive Contention-aware Thread Migrations. In *Computing Frontiers*, 2011.
- [24] E.Z. Zhang, Y. Jiang and X. Shen. Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs? In *SIGPLAN PPoPP*, 2010.
- [25] K. Singh, M. Curtis-Maury, S. A. McKee, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, M. Schulz. Comparing Scalability Prediction Strategies on an SMP of CMPs. In *Euro-Par*, 2010.
- [26] E. Ebrahimi, C. J. Lee, O. Mutlu, Y. N. Patt, Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *ASPLOS* 2010.
- [27] M. Bhadauria, S. A. McKee. Optimizing Thread Throughput for Multithreaded Workloads on Memory Constrained CMPs. In *International Conference on Computing Frontiers*, 2008.
- [28] M. Curtis-Maury, K. Singh, S. McKee, F. Blagojevic, D. Nikolopoulos, B. de Supinski, and M. Schulz. Identifying energy-efficient concurrency levels using machine learning. In *International Workshop on Green Computing*, Sept. 2007.
- [29] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *ISCA*, 2004.
- [30] R. Kumar, G. Agrawal, and G. Gao. Compiling several classes of communication patterns on a multithreaded architecture. In *IEEE SPDP*, 2002.
- [31] S. Zhuravlev, S. Blagodurov and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *ASPLOS*, 2010.
- [32] R. Thekkath and S. J. Eggers. Impact of Sharing-Based Thread Placement on Multithreaded Architectures. In *ISCA*, 1994.
- [33] B. Barnes, B. Rountree, D.K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A Regression-Based Approach to Scalability Prediction. In *ICS*, June 2008.
- [34] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. *OSDI* 2010, Vancouver, Canada, October 2010.
- [35] R. McDougall. Taming Your Emu to Improve Application Performance. Sun Microsystems, PAE. Sun BluePrints™ OnLine, Feb. 2004.
- [36] J. Attardi and N. Nadgir. A Comparison of Memory Allocators in Multiprocessors. <http://developers.sun.com/solaris/articles/multiproc/multiproc.html>