When Less Is MOre (LIMO): Controlled Parallelism for Improved Efficiency

Gaurav Chadha, Scott Mahlke, Satish Narayanasamy
Advanced Computer Architecture Laboratory, University of Michigan
Ann Arbor, MI, USA
{gauravc, mahlke, nsatish}@umich.edu

ABSTRACT

While developing shared-memory programs, programmers often contend with the problem of how many threads to create for best efficiency. Creating as many threads as the number of available processor cores, or more, may not be the most efficient configuration. Too many threads can result in excessive contention for shared resources, wasting energy, which is of primary concern for embedded devices. Furthermore, thermal and power constraints prevent us from operating all the processor cores at the highest possible frequency, favoring fewer threads. The best number of threads to run depends on the application, user input and hardware resources available. It can also change at runtime making it infeasible for the programmer to determine this number.

To address this problem, we propose LIMO, a runtime system that dynamically manages the number of running threads of an application for maximizing peformance and energy-efficiency. LIMO monitors threads' progress along with the usage of shared hardware resources to determine the best number of threads to run and the voltage and frequency level. With dynamic adaptation, LIMO provides an average of 21% performance improvement and a 2x improvement in energy-efficiency on a 32-core system over the default configuration of 32 threads for a set of concurrent applications from the PARSEC suite, the Apache web server, and the Sphinx speech recognition system.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: [Scheduling]; D.4.8 [**Operating Systems**]: [Modeling and prediction]

Keywords

Dynamic Multi-threading, Dynamic Voltage and Frequency Scaling

1. INTRODUCTION

Due to limited success in improving efficiency of a single core and continuous technology scaling, chip multiprocessors (CMPs) have become the standard in providing greater computational power. With CMPs, architects place together many simpler cores on a single chip instead of a single large complex core, while still working within the same power envelope. Running many cores at a lower voltage/frequency expends less energy. So much so that, even phones, tablets and other embedded devices today use multicore processors (e.g. Qualcomm Snapdragon, Apple A5, Samsung Exynos, NVIDIA Tegra 3 - all have quad-core processors). This trend, however, requires programmers to create applications with sufficient thread level parallelism (TLP) to extract performance efficiently from the CMPs.

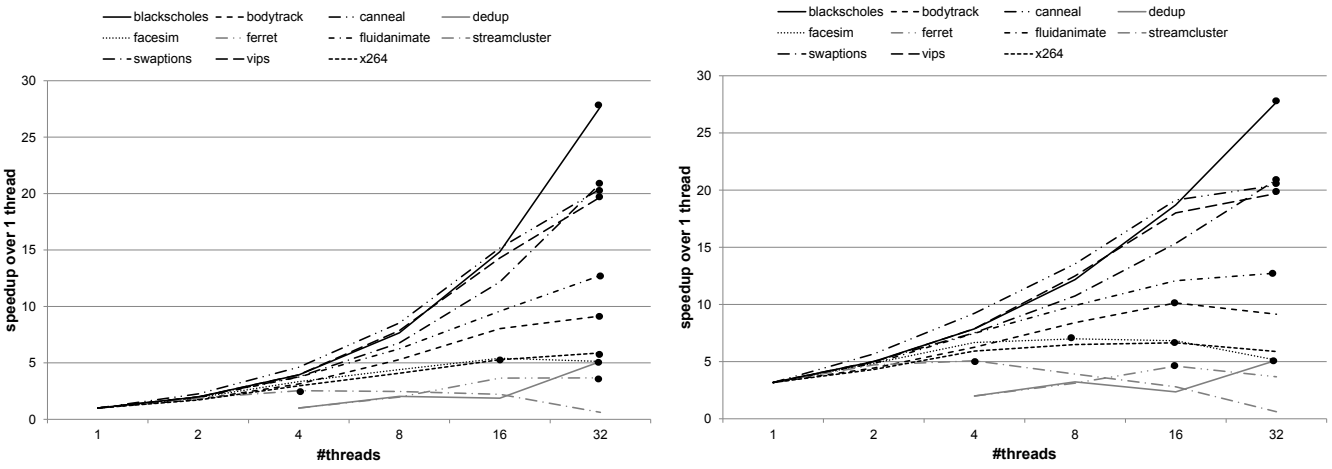
Efficient parallel programming is a difficult task. Many mature parallel programming paradigms like OpenMP, MPI, Nvidia's CUDA, OpenCL, Intel's Ct, TBB, are now available which make this job more feasible and help programmers effectively divide their applications into many threads. However, a very important problem faced by programmers is how many threads should an application be divided into for the best performance and energy-efficiency. Spawning too few threads might lead to underutilization of CMP resources, making the application inefficient. Having too many threads, on the other hand, runs the risk of over-subscribing the resources which again causes performance and energy losses. This problem is magnified by the presence of many different CMPs with varied numbers of cores and configurations (e.g. OMAP 5 vs NVIDIA Tegra 3).

We observe that technology imposed constraints will further shift the scales in favor of running fewer threads than the number of available processor cores. One study found that with a 45 nm TSMC process, less than 7% of a 300mm^2 chip can be operated at the highest possible frequency for a constant power budget of 80W [34]. Commercial processors allow operating systems to perform Dynamic Voltage and Frequency Scaling (DVFS) [11] to increase the frequency of some cores when others are disabled while still working within a fixed power budget. This strengthens the case for using less cores for applications where a higher number of threads does not give a significant performance boost.

A common solution is to set the number of threads equal to the number of available cores. To improve upon this scheme, previous work has proposed techniques that profile applications statically to choose an appropriate number of threads [17, 21, 22] to improve performance by reducing communication and contention for shared resources (however, they did not consider power constraints and DVFS which would further favor running fewer threads, nor were they looking to increase energy-efficiency of the application). Unfortunately, static solutions are limited due to several reasons:

- **Different Inputs**: The same application can exhibit varying degrees of parallelism and performance scalability for different inputs. A static solution would be unsuccessful in pre-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'12, October 7-12, 2012, Tampere, Finland.
Copyright 2012 ACM 978-1-4503-1424-4/12/09 ...$15.00.



(a) Without DVFS

(b) With DVFS

Figure 1: Speedup of Parsec benchmarks with different number of threads over their single threaded runs. These benchmarks were run on a 32 core system.

dicting the optimal number of threads for any input it has not profiled before.

- **Changing available system resources**: The amount of cache capacity and bandwidth available to an application's threads can change as other applications running in the system consume more or less of these resources. A static solution would be completely oblivious of these changes and thus incapable of adapting to these.
- **Different hardware configurations**: Many different CMPs exist with varied number of cores and cache/bandwidth configurations. A static solution will have to have profiling data on all such hardware configurations to be effective which might quickly become intractable.
- **Changing program characteristics during execution**: All of the above remaining constant, an application in itself can have many different execution phases with significantly different characteristics with regards to cache usage, bandwidth utilization, degree of parallelism available, etc. leading to different numbers of optimal threads for different phases.

For the above reasons, static solutions leave much room for improvement. In this work, we propose a lightweight run-time system, *Less Is MOre* or LIMO, which changes the number of running threads of an application dynamically, thus adapting to fine-grained changes in the best number of threads to run. The objective of LIMO is to use DVFS and variable active core count to run an application as efficiently as possible. When many threads exist and resources are not constrained, maximal threads are run at lower frequency. However, when hardware (e.g., shared L2 cache space) or software (e.g., lock variables) resources limit parallel performance, fewer threads are kept active at a higher frequency. For example, if a thread goes into a spin loop waiting on a shared variable, this thread is not doing any useful work and can be disabled by clock-gating or power-gating the core. The power saved from this core is used to boost the voltage and frequency of the remaining cores which are doing useful work. The best number of threads to use for an application particularly those with heterogeneous threads can change frequently as threads move through different code regions.

Traditional OS scheduler level techniques employing DVFS only look for CPU utilization, which if low, the core's frequency / voltage is stepped down to save power. Intel Turbo Boost goes a step further, and apart from disabling cores as requested by the OS, it increases the frequency / voltage of the remaining active cores. This, though, is a purely reactive mechanism, coming in to effect after detecting low CPU utilization on some cores. Distinct from the above and other related works (Section 5), our work takes measures to reduce shared resource contention, employ DVFS and increase performance aggressively and pro-actively. LIMO not only disables cores with inactive / stalled threads, but also those with active threads doing useful work, when it determines that fewer threads running at higher frequency are better for performance. No prior work ever shuts down a core doing useful work. LIMO also monitors contention in shared resources (shared L2 cache and bandwidth) and pro-actively reduces the number of active cores, if they start getting oversubscribed. Detection of spin loops (such cores are disabled by LIMO) also sets this work apart. Detecting these is important as these keep the CPU utilization high without making progress in program execution, subverting the OS' attempt at shutting down cores running unproductive threads.

2. ROADBLOCKS TO SCALABILITY

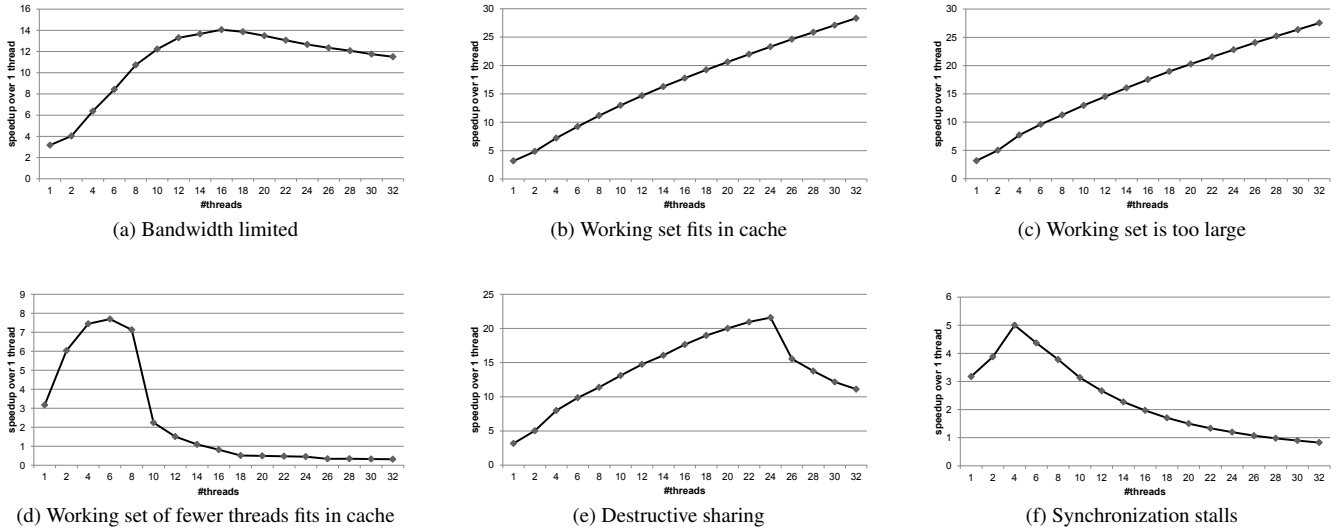
It is common for programmers to create as many threads as the number of processor cores available to execute their program. If a programmer expects that some threads could block for any reason, then she might create more threads than the number of available processor cores in the hope that the operating system scheduler would help her achieve higher performance.

The motivation for our work is that greedily executing as many threads as the system permits may not always yield the best performance.

In this section, we discuss performance scalability issues for shared-memory programs and motivate our work by illustrating how executing fewer threads in some instances can yield better performance using micro-benchmarks and PARSEC benchmarks [4]. The experiments discussed in this section were conducted on a 32-core system containing four 8-core Intel Xeon X7560 processors each with 24 MB last level L3 shared cache, and 32 GB of main memory.

2.1 Lack of Parallelism

Depending on the program input, the amount of parallelism available in an application can vary. As a result, it is possible that a



(a) Bandwidth limited (b) Working set fits in cache (c) Working set is too large

(d) Working set of fewer threads fits in cache (e) Destructive sharing (f) Synchronization stalls

Figure 2: Speedups (with DVFS) of micro-benchmarks over their single threaded runs on a 32 core system.

programmer creates more threads than necessary to expose parallelism, which could lead to contention for shared-data, resulting in decreased performance. There are two ways in which contention for shared-data could hurt performance. One is due to the cost of synchronization waits, and the other is due to the frequent transfer of data from one processor core to another.

Lack of parallelism leads to increased synchronization wait time. If a synchronization operation is implemented as a busy-wait (e.g. using a spin loop), it causes the processor core executing the busy-wait operation to waste power which could have been utilized to increase the frequency of the other cores performing useful work. If a synchronization operation yields a processor core on wait, there is the cost of context switching that thread in and out. Thus, when more threads contend for synchronization resources, performance of the application can degrade. Figure 2(f) shows the performance of a micro-benchmark that exhibits this behavior. All the threads of the program concurrently compute factorial of a number and add it to a shared variable protected by a single lock. Performance drops when we increase the number of threads beyond four while executing on a 32-core system. When we slightly modified the same program by removing the lock, we observed the behavior shown in Figure 2(e). Performance scales slightly better, but eventually starts decreasing after 24 threads. The reason for this performance drop is not due to contention to synchronization variable, but due to the contention to shared-data that tracked the sum of all the factorials. Thus, in addition to synchronization cost, frequent transfer of shared-data between threads could also degrade performance as it could cause frequent coherence invalidations of the cache block containing the shared-data.

If parallelism in an application varies based on input, a programmer would not be able to determine the optimal number of threads to achieve the best performance. Thus, it is important for a runtime system to observe these above effects and control the number of concurrently executed threads to achieve higher performance.

2.2 Lack of Sufficient Shared System Resources

Concurrent threads contend for shared physical resources in a system. Shared caches, on-chip network and memory bandwidth are some of the heavily contended shared resources.

2.2.1 Shared Cache

To study the effects on the shared last level cache (24 MB L3 in our 32-core system), we created a micro-benchmark where each thread walks over its private array with a stride of one cache line size and summed over the elements. The working set (WS) size of a thread was controlled by fixing the maximum array index while keeping the total size of the array constant across all three cases discussed below.

- **Working set fits in cache** (Figure 2(b)): Not considering other scalability limiting factors, if the WS of an application fits in the shared cache, executing the maximum number of threads would yield best performance.
- **Working set is very large without data reuse** (Figure 2(c)): If one thread's WS size exceeds that of the last level cache (LLC) or there is no data reuse (e.g. streaming applications such as video decoders), more threads could yield higher performance by initiating more memory requests in parallel and thereby exploit memory level parallelism (MLP).
- **Working set with fewer threads fits into the shared-cache** (Figure 2(d)): It's a common case where the WS of fewer threads fits in the shared cache, but not of too many threads. It is important to have a runtime mechanism that can adapt the number of concurrent threads to attain high performance.

2.2.2 Memory Bandwidth

Applications with poor locality (such as streaming applications) require high memory bandwidth. As we had discussed earlier, these applications tend to benefit from more threads to exploit MLP. However, as their demand increases to a point when the on-chip network and memory bandwidth are saturated, then we may start to see performance degradation due to destructive interference between memory requests. Figure 2(a) shows the performance of a program that sums the values of elements spread over a very large array.

2.3 Dark Silicon Favors Fewer Threads

While it is true that with shrinking device sizes, more and more transistors can be integrated on a chip, increasing the number of

cores on a processor, multicore scaling has become thermal and power limited. What this means is that though manufacturers can still increase the number of cores in a processor, not all of them can be turned on at their maximum frequency at the same time [9, 34], leading to the term *dark silicon*. This limitation can be seen in commodity processors today, such as the quad-core Intel Core-i7 systems. These processors employ Intel Turbo Boost Technology [11]. However, the frequency / voltage and the number of cores active can only be changed in discrete steps.

In this work we start with disabling cores not doing useful work. The power "saved" from such cores can be used to boost the frequency of the remaining active cores, thus helping improve the performance of applications that do not show good scalability. Frequency scaling is done according to the following power equation:

$$P = ACV^2F \quad (1)$$

where P is power, A is the activity factor, i.e. the fraction of the circuit that is switching, C is the switched capacitance, V is the supply voltage and F is the clock frequency. A is a constant, and assuming we scale voltage and frequency together,

$$P \propto F^3 \quad (2)$$

Thus for every reduction in the number of active cores by half the frequency can be boosted by a factor of $2^{\frac{1}{3}}$. Figure 1(a) shows the performance scalability of PARSEC benchmarks for `sim-large` input on our 32-core system. The optimal number of threads that yields the best performance for an application is indicated using a black dot. We observe that only two programs, `streamcluster` and `facesim` perform better when executed with fewer than 32 threads. However, when we assume DVFS to increase the frequency of the configuration that runs fewer than 32-threads (2.268 GHz for 4-cores, 1.8 GHz for 8-cores, 1.429 GHz for 16-cores), five out of eleven applications perform better with fewer than 32-threads.

However, it is difficult to know apriori the best number of threads to execute for a given application, as it also depends on the program input and system configuration. Also, the same number of threads may not be the best answer throughout the execution of an application as it could have different phases exhibiting varied characteristics. In this paper, we propose a scheme that dynamically varies the number of active cores and their frequencies depending on parallelism available in the application and also application's demand for the shared-cache resource.

3. LIMO

LIMO is a runtime system that dynamically changes the number of running threads of an application to deliver higher performance and energy-efficiency when compared to running threads on all available processor cores. Section 2 listed in detail the different factors that affect a multi-threaded application's scalability. LIMO monitors synchronization stalls, demand for shared cache and off-chip memory bandwidth to determine the number of threads to execute. If LIMO decides to execute fewer threads than the number of available processor cores, it applies DVFS to boost the frequency of the active cores.

3.1 Design Overview

The application is allowed to create as many threads as the programmer had specified (for the applications we analyze, we create as many threads as the number of available processor cores). Thus, one thread is created per processor core. For example, in a 32-core CMP, the application starts out by running 32 threads, one on each core.

```
activeThreads ():
  for each thread t that stalls:
      disableCore (t)
      activeThreadsCount--
  for each thread t that is now ready:
      add t to readyThreadsSet
```

Figure 3: Algorithm for determining the number of threads that can do useful work

```
wsThreads ():
  if quantum instructions executed since last call:
      wsSize = WSEstimator ()
      maxWSSize = wsThreshold x L2CacheSize
      wsSizePerThread = wsSize/avgNumActiveCores
      wsThreadsCount = maxWSSize/wsSizePerThread
  else:
      wsThreadsCount is not updated
```

Figure 4: Algorithm for determining the maximum number of threads that can run without causing thrashing in the L2 cache

```
runningThreads ():
// This is called if
// 1. one or more threads are stalled or are ready
// 2. quantum instructions executed since last call
  activeThreads ()
  wsThreads ()
  maxThreadsCount =
      min (activeThreadsCount, wsThreadsCount)
  if maxThreadsCount < activeThreadsCount:
      disable (activeThreadsCount -
          maxThreadsCount) cores
      if maxThreadsCount <= thresholdLower:
          increase frequency
  else if maxThreadsCount > activeThreadsCount:
      if maxThreadsCount >=
          (activeThreadsCount +
              readyThreadsSet.size):
          enable all threads in readyThreadsSet
      else:
          enable (maxThreadsCount -
              activeThreadsCount) threads
      if maxThreadsCount >= thresholdUpper:
          decrease frequency
```

Figure 5: Algorithm for determining the number of threads to run and the frequency

LIMO monitors the threads' progress (Figure 3). If any thread stalls (in a synchronization function, blocking I/O call or is suspended), the core on which this thread was running is disabled and the number of active threads (*activeThreadsCount*) is reduced. Since this thread is clearly not making any forward progress, disabling that core saves power and leaves room in the fixed power budget to increase the voltage and frequency of the remaining cores. Similarly, when previously stalled threads can now do useful work, they are added to a set keeping track of all ready but not executing threads (*readyThreadsSet*).

To reduce contention over the shared cache, LIMO uses estimates of the working set (WS) size of the application [8] (*WSEstimator*) and keeps it from oversubscribing the cache (Figure 4). After every *quantum* of 100 million instructions, the WS of the application evaluated over the last period is used in the decision of how many threads to run over the current period. If the WS of the application is too big to fit in the shared L2 cache causing thrashing and reducing efficiency, estimates of the WS size of configurations with lower number of threads are calculated using simple linear scaling

of the WS size with the number of threads (our algorithm does not need the exact WS size, and thus this estimate is adequate). We found out empirically that even a configuration whose estimated WS size exceeds the L2 cache capacity by 40% (*wsThreshold*) can deliver good performance. Using this, the algorithm calculates the maximum number of threads to run (one whose estimated WS size does not exceed the L2 cache capacity by more than *wsThreshold*), *wsThreadsCount*.

As detailed in Figure 5, at the end of each *quantum* of instructions or when a thread either gets stalled or becomes ready, the algorithm uses the minimum of *wsThreadsCount* and *activeThreadsCount* as the maximum number of threads that can run (*maxThreadsCount*). Employing DVFS, we can boost the frequency / voltage when fewer cores are active and get better performance and energy-efficiency.

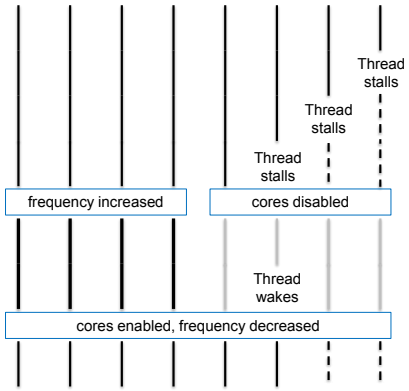


Figure 6: An example of varying the number of running cores in the case of 8 threads.

As discussed in section 2.3 in current systems voltage, frequency and the number of active cores can only be changed in discrete steps and not continuously. In our system we assume 4, 8, 16 or 32 cores can be active at a time (active core levels). This gives three core number thresholds at which the frequency and voltage can be stepped up or down. Assuming IPC/core remains constant while decreasing the number of active cores (a rather conservative assumption since the strain on shared resources decreases), and that performance scales linearly with frequency (an approximate assumption, which is adequate here since we use this only to obtain core number thresholds for our heuristic and not for any actual performance evaluation), we find out the core number thresholds using $n_1 f_1 = n_2 f_2, n_2 = \lfloor \frac{n_1 f_1}{f_2} \rfloor$ where n_i is the number of active cores and f_i the frequency at which they are run. If *maxThreadsCount* is below the next lower core number threshold for the current number of active cores (*thresholdLower*), some active or ready cores are disabled and the frequency is increased. A similar approach is employed if *maxThreadsCount* is greater than *thresholdUpper*.

For example, suppose initially all 32 threads were doing useful work. After some time, 12 of those threads are stalled, leaving only 20 threads that are doing useful work (*activeThreadsCount* = 20). If the frequency at which 32 cores run is 1.134 GHz, keeping the power budget constant and assuming we scale both voltage and frequency together and linearly, 16 cores can run at a maximum frequency of 1.429 GHz following 2. 16 cores running at 1.429 GHz will perform better than 20 cores running at 1.134 GHz. This is because, let's say IPC/core at 1 Ghz is a. 16 cores running at 1.429 GHz give a performance of 16 x 1.429 x a = 22.864a, whereas 20 cores running at 1.134 GHz give a performance of 20 x 1.134 GHz x a = 22.68a. Figure 6 shows an example of the mechanism for 8 threads.

We have designed and used distinct mechanisms to detect different roadblocks to scalability, because as shown in Section 2 the action to be performed (increase or decrease the number of active cores) varies with the cause of reduced performance.

3.2 Implementation

LIMO assumes support from the operating system (OS), hardware and compiler to gather required information and make decisions on the number of active cores that can achieve high performance.

LIMO relies on two specific pieces of runtime information. First, we need hardware support to determine the working set size of a thread, which would allow LIMO to determine the number of threads that can be executed without degrading the performance due to shared-cache capacity constraints.

Second, we need to know the number of threads that can make progress. The operating system already has information about threads that block by invoking a system call. If a synchronization operation is implemented as a blocking wait, then when a thread needs to stall waiting for a synchronization operation to succeed it invokes an operating system call to block itself (e.g. `sys_futex()`). However, we need additional support for detecting threads that block due to busy-wait (spin loop).

Instead of assuming runtime support to detect spin loops, we propose to use static analysis to conservatively determine loops that are likely to be spin loops. The analysis finds loops where the conditional variables can be guaranteed to be not modified within the loop body. Once the compiler finds a spin loop in an application, it transforms it to include a check in the spin loop that checks how many times the loop has iterated. If the number of iterations exceeds a threshold (three in our experiments), the compiler inserts a special system call to yield the thread to the operating system, which would inform the operating system that the thread has blocked due to a spin loop. The thread is scheduled back by the operating system after one time quantum has expired.

Thus, the OS has access to information about how many threads are in a state where they can make useful progress. Also, the OS reads from our performance counter that keeps track of the estimated working set size of the application. Based on this information, the operating system decides to activate the appropriate number of cores (based on the algorithm discussed in Section 3.1). Either when the current interval (used for WS estimation) ends, or when a thread's state changes, OS recalculates the number of cores that should be activated. It signals the hardware specifying the number of cores and the frequency they can operate at. When more threads are in the active state where they can make useful progress, the OS applies its baseline scheduling policies to ensure fairness between the threads.

We assume on-chip switching regulators [15] for DVFS, which can change a processor's power state in 30ns. We also use the working set estimator described in [8]. The memory addresses accessed (at cache line granularity) are hashed into n-buckets, represented by an n-bit vector, using a randomizing hash function. Given the fraction (f) of buckets filled, the working set size can be estimated as $\frac{log(1-f)}{log(1-\frac{1}{n})}$. This calculation is done afer every period of 100 million executed instructions. This information can be stored in special registers and read by the OS when executing the scheduling algorithm.

3.3 Fine grained monitoring

With the ability of fast change of processor power states, LIMO

adapts to fine-grained changes in program characteristics. By disabling (clock-gating) the cores it exploits even small windows of energy saving opportunities and boosts performance by reducing contention in shared resources. Since the cores are clock-gated, they preserve their state and there is no need of a context switch. This facilitates fast wake up of cores (30 ns), and thus the overhead of this scheme is very low (even so, it is included in our results).

4. EXPERIMENTAL EVALUATION

We used full system simulators to design and evaluate our scheme, capturing and negotiating the effects of our scheme on the entire system.

4.1 Methodology

We used a modified timing simulator FeS2 [28], with support for shared L2 caches, that uses the full system simulator Simics [25]. FeS2 is a cycle-accurate x86 simulator with support for running multi-threaded programs. It includes a detailed processor core model. Ruby from the gem5 project [5], is used to model the memory subsystem including non-blocking caches, memory controllers, main memory, etc. We simulated the effects of the design presented in section 3 with our simulation infrastructure. Hardware modifications are proposed in our design, necessitating the use of simulators for this study.

We evaluate our scheme on benchmarks from the PARSEC benchmark suite [4], Apache HTTP server program (httpd) and Sphinx (speech recognition) from the ALP benchmark suite [20]. Blackscholes, dedup, facesim, ferret, fluidanimate, streamcluster, swaptions and vips from Parsec were run with the input simlarge. Apache server was benchmarked using Surge [2].

While we have not evaluated our scheme with multiprogram workloads, for such scenarios we propose that each application be allotted a fixed number of maximum cores that it can use, partitioning the total number of cores among applications. Within each such partition, our scheme presented in this paper can be used as it is.

Our design monitors and is capable of detecting oversubscription of shared resources (bandwidth, L2 cache) and synchronization stalls. Other factors limiting scalability were discussed in section 2. However, we did not observe all those scalability limiting factors in real benchmarks, and hence this section discusses only the ones we did.

The full system simulator used is too slow to simulate these benchmarks for the entire duration of their executions. So, first we executed the benchmarks in their entirety on Simics. Using stores as an execution progress metric, we took checkpoints at regular intervals during these runs. Thereafter, starting from these checkpoints, we ran timing simulations with FeS2 for 80 million *useful* instructions (dynamic instructions executed in user mode, excluding the ones executed in a spin loop). All statistics were cleared after 20 million such instructions, giving sufficient time for the caches to warm up. The benchmarks were sampled in this way so that we would observe different phases of execution of the program showing different characteristics. (Note: Working set estimation may not occur a second time in our simulation window of a single checkpoint. This is fine since working sets don't change significantly in shorter execution spans.)

The hardware configuration parameters are modelled after a Core-i7 system, with private L1 caches and shared L2 cache. Simics simulated a 32 core machine, running unmodified Fedora 5, kernel 2.6.15-1. While 32 cores might seem excessive for embedded devices today, the market trend shows that this might not be a distant possibility. ARM Cortex A15 (to be released this year) will have 8 cores, which is an eight fold increase in the number of cores in the last 3 years.

Following equation 2 we ran 4, 8, 16, 32 cores at 2.268, 1.8, 1.429, 1.134 GHz respectively. The simulated machine had 32 cores for all configurations simulated. To simulate 4, 8 and 16 cores, the remaining cores are simply not simulated in Simics.

4.2 Results

In this section we present our findings regarding the performance of the proposed scheme (LIMO) for the benchmarks discussed above.

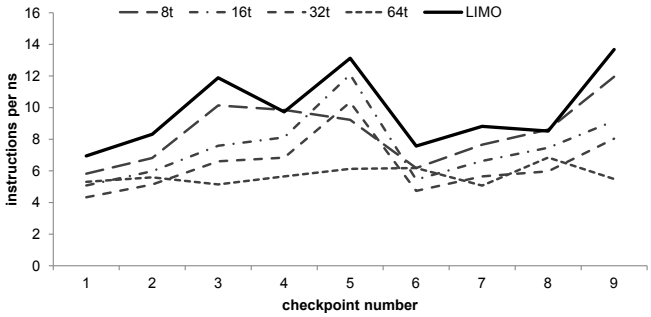
We measure performance of the benchmarks in terms of *useful* instructions committed per nano second. *Useful* instructions are dynamic instructions executed in user mode, excluding the ones executed in a spin loop. Henceforth, in this paper we will refer to useful instructions as instructions. *Synchronization stalls* is the total amount of time spent by all the threads stalled in synchronization functions (e.g. waiting on a lock). If $sync_{tot}$ is the total time spent by all threads stalled in synchronization functions aggregated across all threads, $useful_{tot}$ is the total time spent by all threads doing useful work aggregated across all threads, then

$$\%sync\ stall = \frac{sync_{tot}}{useful_{tot} + sync_{tot}} \times 100$$

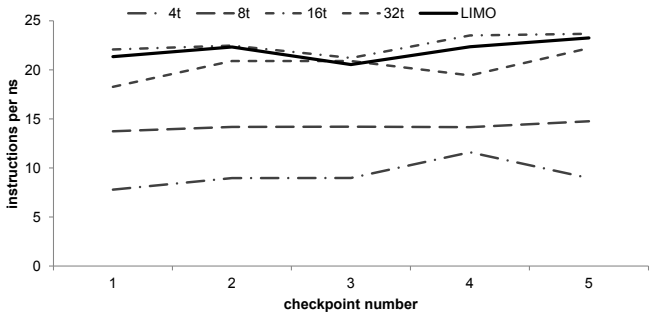
The benchmark ferret performs image similarity search. It has been parallelized using the pipeline model with six stages. The first and the last stage for input and output of data respectively. The middle four parallel stages are for image segmentation, feature extraction, indexing of candidate sets and ranking. The distinctively different tasks need to be done by threads from different stages makes them highly heterogeneous. Figure 7(a) shows the performance of different thread configurations for ferret in terms of instructions committed per nanoseconds (IPS). Each line in the plot has nine discrete points (nine checkpoints) representing progressing execution on the x-axis. The different configurations are 8 threads active (8t), 16 threads active (16t), 32 threads active (32t), 64 threads in the system (64t), variable number of threads active with hardware managed DVFS like Turbo Boost (TB_DVFS) and our scheme (LIMO). For all configurations except 64t, number of threads active equals number of cores active. 64t has 64 threads running on 32 cores. LIMO has a variable number of cores active varying dynamically over the course of execution of the application, and has a maximum of 32 threads in the system.

As can be seen from Figure 7(a), except LIMO, no one configuration performs the best always. 16t is best for checkpoint 5, whereas 8t performs the best for checkpoint 3. LIMO does very well and performs better than all the others for most checkpoints and as good as 8t for the remaining (checkpoints 4 and 8). Figure 7(e) shows $\%sync\ stall$ for ferret. For checkpoints 1, 2, 5, 6 and 7, in configurations 16t and 32t a large number of threads spend a considerable amount of time stalled because of synchronization constructs. LIMO recognizes this and disables some cores (almost all of whose threads are stalled and not doing any useful work), letting fewer cores (which can do useful work) run at a higher frequency, while still working within the same constant power budget. Figure 8 clearly shows the average number of cores active is much lower than 32 for LIMO. It can thus deliver better performance than any other configuration.

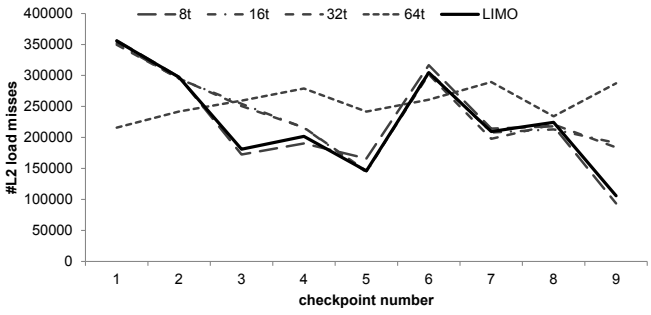
However, as discussed in section 2 synchronization stalls is just one of many factors affecting scalability. 8t performs particularly well on checkpoints 3, 4 and 9. Figure 7(c) shows 8t having markedly fewer L2 load misses on precisely those checkpoints. The working set for 8 threads fits better in the L2 cache, whereas for configurations with higher thread numbers, the working set be-



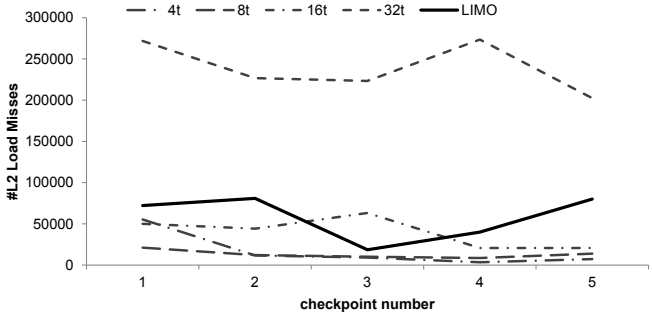
(a) Performance of ferret



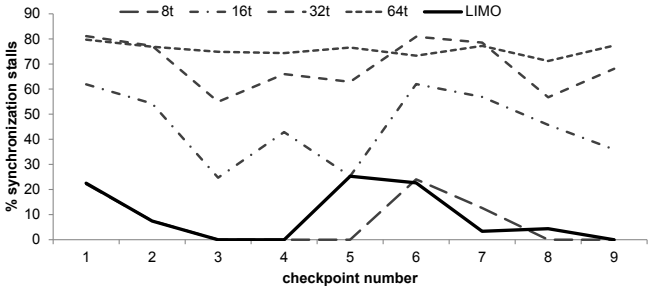
(b) Performance of httpd



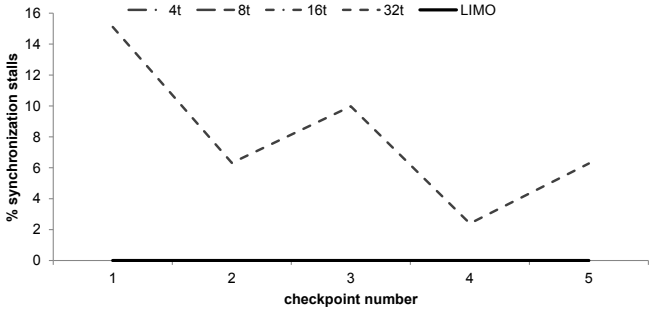
(c) L2 load misses for ferret



(d) L2 load misses for httpd



(e) Synchronization stalls for ferret



(f) Synchronization stalls for httpd

Figure 7: Execution statistics of different numbers of threads and LIMO on a 32 core system for ferret and httpd

comes too large, significantly decreasing the efficacy of the L2 cache. LIMO, having information on the working set size for one configuration, estimates that the working set of 8 threads can fit much better in the L2 cache. It, thus picks 8 threads almost all the time for these three checkpoints thus delivering high performance. Further proof of the analysis just presented can be found in the fact that high performance of LIMO accompanies very low percentage of synchronization stalls and close to the lowest number of L2 load misses among all configurations.

The Apache httpd server benchmark shows very different characteristics than ferret. It uses a work pool parallelization model, where each incoming request is handled by a different free thread. The number of threads that Apache can spawn thus limits the number of requests it can serve concurrently. For a heavy load of incoming requests, there is always work for the threads to do and consequently there are few synchronization stalls (Figure 7(f)). 16t is mostly the answer for best performance (Figure 7(b)), performing better than 32t but, unlike ferret it is not because of synchronization stalls. L2 load misses (Figure 7(d)), on the other hand, are significantly higher for 32t compared to other configurations. The penalty imposed by such a high number of L2 load misses eclipses the benefit obtained from parallel computation with more threads. 16 threads with a higher frequency and a smaller working set are able to make much better use of the L2 cache and deliver higher performance. On an average LIMO picks close to 16 threads throughout the execution of the application and thus gets performance that is very close to 16t. However, it still tries to disable cores and increase the frequency whenever the number of active cores goes below a threshold. This ends up hurting performance slightly, compared to always keeping as many cores active as can do useful work with a maximum of 16 cores (essentially what 16t does).

In Figure 9, we show the performance of TB_DVFS, 64t and LIMO relative to 32t for all benchmarks. A system with an Intel Turbo Boost like hardware is represented by TB_DVFS. As expected, it is not able to capture the benefits of pro-active and low latency disabling of threads and subsequent change in voltage and frequency levels. However, it does well for streamcluster, since

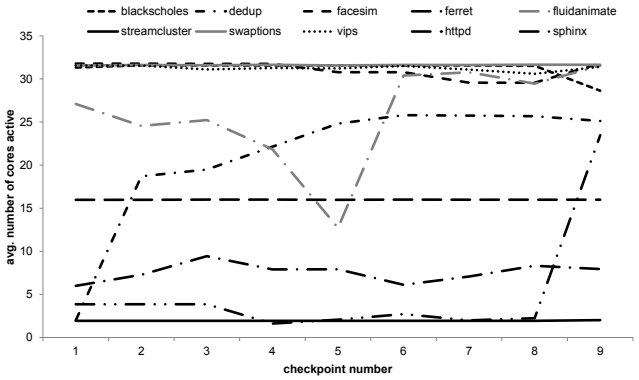


Figure 8: Number of cores active on average during the execution.

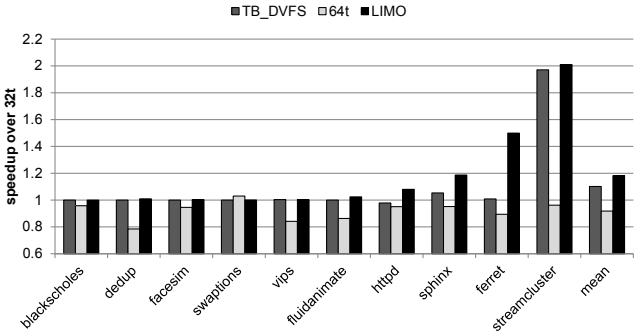


Figure 9: Speedup for all benchmarks over 32t.

this benchmark does not exhibit varying characteristics during its execution. 64t represents what the operating system could do to address synchronization stalls. In this configuration there are 2 threads per core. If a thread stalls, the OS will always have a thread that is not running anywhere else to schedule on this core. However, this can be beneficial only if the application performs better with 64 threads compared to 32, and if threads don't stall too frequently, since otherwise the context switch overhead can outweigh the benefits of switching threads on cores. However, as can be seen from Figure 9, 64t is unable to deliver any performance benefits, and in fact hurts performance in most cases. Lack of scalability, as shown in Figure 1, provides a compelling reason for this trend. While this might indicate 32 cores is over-kill for the benchmarks chosen in this study, we argue that lack of scalability is a real problem which will get exposed, if not at 32 cores with a different set of benchmarks, but definitely at a higher number of cores. For instance, if we had limited our study to a maximum of 8 cores, most of the benchmarks wouldn't have shown a problem with scalability, which gets exposed at 32 cores.

LIMO is the best performing configuration for all checkpoints (Figure 9). The benefits of this scheme get explored specially in cases where there is high variability in the execution characteristics of benchmarks, like ferret and sphinx. However, even for benchmarks like streamcluster, httpd and fluidanimate, LIMO shows benefits. The performance characteristic of streamcluster portrays an important use-case for LIMO, where the programmer is unaware of the best number of threads to run. LIMO recognizes that it runs best with 2 threads, even if there are 32 cores available.

Figure 11 shows that not only does LIMO improve performance of multi-threaded applications, but it is also highly energy efficient. Here energy efficiency is defined as (energy consumed by running 32 threads)/(energy consumed by LIMO). All energy numbers were obtained using McPat [24].

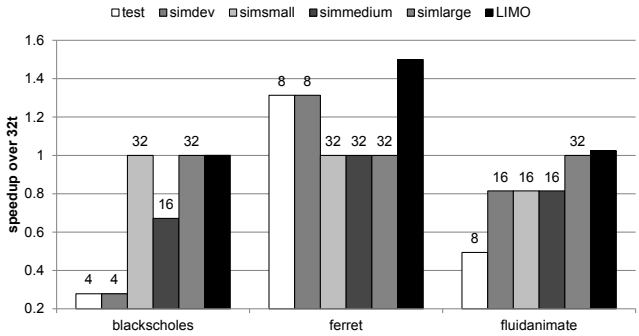


Figure 10: Speedup for blackscholes, ferret and fluidanimate over 32t. `test`, `simdev`, `simsmall` and `simmedium` are sample inputs to the benchmarks that a static profiler could be trained on. `simlarge` is the actual input used for final performance measurements. The numbers over corresponding bars represent the best number of threads chosen.

Figure 10 compares LIMO with possible static profiling schemes. We ran each of the parsec benchmarks with five different inputs, for 4, 8, 16 and 32 threads. Possible static profiling schemes would choose the number of threads that performed the best, as the right number of threads to run for best performance. As can be seen from Figure 10, different inputs give different answers for the right number of threads to run for best performance. LIMO performs the best regardless of the profile-guided answers. Blackscholes shows good scalability at least up to 32 threads and does not show variation in the right number of threads to run even during the course of execution of the program. Therefore, LIMO picks 32 threads to run almost all the time and performs as well as 32t. Ferret, as shown earlier shows significant variations due to both synchronization stalls and cache capacity problems. LIMO exploits the opportunities presented due to such variations and outperforms all other thread configurations. The performance of LIMO is also significant in the case of fluidanimate. Fluidanimate shows good scalability up to 32 threads, like blackscholes. However, unlike blackscholes, it exhibits variations in execution characteristics and thus the best number of threads to run, which is exploited by LIMO delivering better performance than any static choice of active threads.

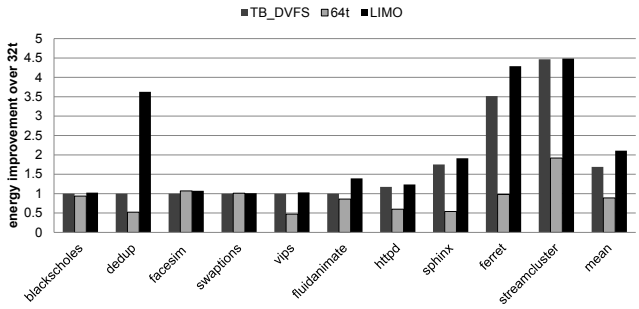


Figure 11: Energy efficiency for all benchmarks over 32t.

5. RELATED WORK

More threads do not always lead to better performance is the cornerstone of the motivation for this work. Other researchers have also observed this fact. Table 1 summarizes notable differences between LIMO and selected previous related works.

Previous works related to ours can be categorized in the following three categories.

Work	Choose optimal #threads	Variable #threads at runtime	DVFS	Heterogeneous threads	No initial/apriori profiling	Heterogeneous resources
LIMO	✓	✓	✓	✓	✓	
Li and Martinez [23]	✓	✓			✓	
Suleman, et. al. [32]	✓					
Jung, et. al. [12]	✓	✓		✓	✓	
Lee, et. al. [21]	✓					
Lee, et. al. [22]				✓		
Suleman, et. al. [31]						✓
Bhattarjee, Martonosi [3]			✓		✓	

Table 1: Comparison with related work

5.1 Variable number of threads

These previous works vary the number of active threads to run for an application.

Li and Martinez proposed a system in [23] which optimized power consumption of an application given a performance target. The optimization space is two-dimensional - varying the frequency / voltage and the number of cores. A combination of binary search algorithm along with a hill-climbing heuristic is employed to determine the best number of threads and frequency. The number of threads is varied during the execution of the program, but only to achieve the performance target with minimum power consumption on average. LIMO does not need to do a space search and varies the number of active threads during runtime to adapt to changing program characteristics and over the course of execution of the program. [23] sets a performance target to meet, which is hard to estimate. LIMO, on the other hand, tries to deliver the best performance possible while working within a constant power budget (dictated by TDP).

Feedback driven threading by Suleman, et. al. [32] presents solutions for two cases manifested by multi-threaded applications, one where the performance is limited by off-chip bus bandwidth and the other when it is limited by synchronization. They have built models to predict the right number of threads to run, which are trained through profiling initial few iterations of the loop in the application. The models used assume data parallel, homogeneous threads in the application. Similarly, Jung, Han, et. al. [12] propose a solution to mitigate shared resource contention in SMT architectures. Even though, they change the number of threads dynamically for performance, they rely on OpenMP library to automatically adjust thread decomposition and their scheme only works on data-parallel loops. These works, addressing data-parallel applications, are limited in scope compared to LIMO, which is applicable for applications with heterogeneous threads. Also, LIMO employs DVFS to boost frequency when fewer cores are active.

There has been a recent work by Lee and Kim [21], which works on a similar problem as LIMO. However, LIMO differs from their work as theirs does not employ DVFS, is for applications running on GPUs, involves exhaustive exploration of search state space and it does not change the number of threads dynamically.

Additionally, an important point that separates LIMO from the above works is that it does not assume anything about the nature of the multi-threaded application. It could be a data-parallel application with identical threads or a pipeline parallel one with heterogeneous threads, etc.

5.2 Scheduling constant number of threads

There is a large body of work concerned with thread scheduling on multi-core architectures for reducing data communication costs and contention for shared resources: caches, off-chip memory bandwidth, etc.

Thread Tailor [22] by Lee and Clark assumes there are many threads per core in a system. It colocates threads on cores such that data communication among them is minimized. Thereafter, they are run just like an OS schedules threads in an oversubscribed system. Kumar, et. al. [17] propose mechanisms (to reduce data communication costs) to be included in the compiler that helped overlap communication of data with computation by identifying the pattern of communication in the algorithm.

A significant body of work has devised schemes dictating partitioning of shared cache among cores, assigning / scheduling of threads to cores as well as frequency throttling of cores to minimize contention for the shared cache, front-side bus (FSB) and prefetching units [6, 13, 14, 16, 26, 29, 30, 33, 35].

This body of work addresses a similar problem as LIMO - reducing contention in shared resources for improved performance or energy efficiency. But they assume that all threads need to run. LIMO differs from them in that it tries to improve performance through changing the number of active threads (often running fewer than maximal threads) and boosting frequency working within a constant power budget.

5.3 Heterogeneous hardware resources

There is still other work concerned with thread scheduling and mapping to cores on heterogeneous CMPs. The work by Bhattacharjee and Martonosi [3] designs thread criticality predictors. These are used to pick the thread most critical for performance, which can then be sped up through DVFS, load shedding or allocating more resources for that thread. Suleman, et. al. [31] assumed a CMP with one big core and many smaller cores. They proposed executing all critical sections in an application on the big core, speeding up critical section execution and essentially reducing synchronization stalls. Similarly, [1, 7, 10, 18, 19, 27] propose schemes for intelligent thread scheduling and mapping on heterogeneous CMPs.

LIMO differs from these works in that it runs fewer threads with increased frequency whenever it is beneficial for performance. It also changes the active thread count dynamically.

6. CONCLUSIONS

Applications today need to be multi-threaded to take advantage of the multi-core processors seen everywhere in computing. Even though mature programming methodologies have made the difficult task of writing efficient and correct parallel programs easier, programmers still have to face the hard task of determining the number of threads to create for best performance. Too many threads can saturate shared resources, degrading performance. Too few threads might make insufficient use of the resources available making the application inefficient. In an attempt to solve the problem of finding the best number of threads to run for an application for peformance, we propose LIMO - a dynamic runtime system that monitors threads' progress, dynamically changes the number of running threads adapting to fine grained changes in application character-

istics and employs DVFS to boost frequency of the active cores when some others are disabled while still working within a constant power budget.

Through this system, we aim to relieve the programmer from the job of determining the best number of threads to run. An application can be simply started with as many threads as the number of cores on any hardware and our scheme strives to deliver the best performance in so far as the number of running threads is concerned. Using mutlithreaded applications from the Parsec benchmark suite, Apache httpd server and Sphinx from ALP benchmark suite we show an average performance improvement of 21% and a 2x improvement in energy-efficiency over the default configuration of running 32 threads on a 32 core system.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This research was supported by National Science Foundation grants CNS-0964478 and CCF-0916770 and funding from Intel Corporation.

8. REFERENCES

[1] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 298–309, Washington, DC, USA, 2005. IEEE Computer Society.

[2] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '98/PERFORMANCE '98, pages 151–160, New York, NY, USA, 1998. ACM.

[3] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 290–301, New York, NY, USA, 2009. ACM.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[6] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[7] L. De Giusti, E. Luque, F. Chichizola, M. Naiouf, and A. De Giusti. Amtha: An algorithm for automatically mapping tasks to processors in heterogeneous multiprocessor architectures. In *Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering - Volume 02*, pages 481–485, Washington, DC, USA, 2009. IEEE Computer Society.

[8] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 233–244, Washington, DC, USA, 2002. IEEE Computer Society.

[9] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[10] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto. Maximizing power efficiency with asymmetric multicore systems. *Commun. ACM*, 52:48–57, December 2009.

[11] Intel. Intel turbo boost technology in intel core microarchitecture (nehalem) based processors, 2008.

[12] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for smt multiprocessor architectures. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 236–246, New York, NY, USA, 2005. ACM.

[13] D. Kaseridis, J. Stuecheli, J. Chen, and L. John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –11, jan. 2010.

[14] D. Kaseridis, J. Stuecheli, and L. K. John. Bank-aware dynamic cache partitioning for multicore architectures. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 18–25, Washington, DC, USA, 2009. IEEE Computer Society.

[15] W. Kim, M. S. Gupta, G. yeon Wei, and D. M. Brooks. Enabling onchip switching regulators for multi-core processors using current staggering. In *In Proceedings of the Work. on Architectural Support for Gigascale Integration*, 2007.

[16] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *Micro, IEEE*, 28(3):54 –66, may-june 2008.

[17] R. Kumar, G. Agrawal, and G. Gao. Compiling several classes of communication patterns on a multithreaded architecture. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 18 –23, 2002.

[18] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32 – 38, nov. 2005.

[19] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 23–32, New York, NY, USA, 2006. ACM.

[20] M. lap Li, R. Sasanka, S. V. Adve, Y. kuang Chen, and E. Debes. The alpbench benchmark suite for complex multimedia applications. In *In Proc. of the IEEE Int. Symp. on Workload Characterization*, pages 34–45, 2005.

[21] J. Lee, V. Satish, K. Compton, M. Schulte, and N. S. Kim. Improving the throughput of power-constrained gpus through adaptive voltage, frequency, and core scaling. In *PACT'11*, 2011.

[22] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 270–279, New York, NY, USA, 2010. ACM.

[23] J. Li and J. F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *HPCA'06*, pages 77–87, 2006.

[24] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[25] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35:50–58, 2002.

[26] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 153–166, New York, NY, USA, 2010. ACM.

[27] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Comput. Archit. Lett.*, 5:4–, January 2006.

[28] N. Neelakantam, C. Blundell, J. Devietti, M. M. K. Martin, and C. Zilles. Fes2: A full-system execution-driven simulator for x86, 2008.

[29] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[30] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. pages 117–128, 2002.

[31] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 253–264, New York, NY, USA, 2009. ACM.

[32] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. *SIGPLAN Not.*, 43:277–286, March 2008.

[33] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 121–132, New York, NY, USA, 2009. ACM.

[34] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 205–218, New York, NY, USA, 2010. ACM.

[35] S. Zhuravlev, S. Blagodurov, and A. Fedorova. 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, pages 129–142, New York, NY, USA, 2010. ACM.