ChunkedTejas: A Chunking-based Approach to Parallelizing a Trace-Driven Architectural Simulator

RAJSHEKAR KALAYAPPAN, Indian Institute of Technology Dharwad, India
AVANTIKA CHHABRA, Indian Institute of Technology Delhi, India
SMRUTI R. SARANGI, Indian Institute of Technology Delhi, India

Research in computer architecture is commonly done using software simulators. The simulation speed of such simulators is therefore critical to the rate of progress in research. One of the less commonly used ways to increase the simulation speed is to decompose the benchmark's execution into contiguous chunks of instructions and simulate these chunks in parallel. Two issues arise from this approach. The first is of correctness, as each chunk (other than the first chunk) start from an incorrect state. The second is of performance: the decomposition must be done in such a way that the simulation of all chunks finishes at nearly the same time, allowing for maximum speedup. In this paper, we study these two aspects and compare three different chunking approaches (two of them are novel) and two warmup approaches (one of them is novel). We demonstrate that average speedups of up to 5.39X can be achieved (while employing 8 parallel instances), while constraining the error to 0.2% on average.

CCS Concepts: • **Hardware** → **Modeling and parameter extraction**;

Additional Key Words and Phrases: Architectural Simulator, Simulator Performance, Parallelization

ACM Reference Format:
Rajshekar Kalayappan, Avantika Chhabra, and Smruti R. Sarangi. 2018. ChunkedTejas: A Chunking-based Approach to Parallelizing a Trace-Driven Architectural Simulator. *ACM Trans. Model. Comput. Simul.* 9, 4, Article 39 (March 2018), 22 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Computer architecture research is done with the aid of architectural simulators. Research proposals are typically evaluated by implementing them in simulators, and experimenting with a range of workloads and processor configurations. A thorough design space exploration is necessary to fully evaluate a proposal and determine its merit and applicability. This is a highly time consuming task. Consequently, the speed of the simulator is critical in determining the rate of research and the progress in the area. In this paper, we implement and evaluate a novel class of methods for increasing the speed of architectural simulators.

There are many open-source architectural simulators [3–6, 11, 12] available that can be employed for academic research. Simulators can be classified as execution-driven or trace-driven. An execution-driven simulator takes an executable of the benchmark as input, and emulates its execution – that is, it maintains the state of all registers and the memory, and performs operations

Authors' addresses: Rajshekar Kalayappan, Indian Institute of Technology Dharwad, Dharwad, Karnataka, 580011, India, rajshekar.k@iitdh.ac.in; Avantika Chhabra, Indian Institute of Technology Delhi, Hauz Khas, New Delhi, Delhi, 110016, India, avantikachhabra@outlook.com; Smruti R. Sarangi, Indian Institute of Technology Delhi, Hauz Khas, New Delhi, Delhi, 110016, India, srsarangi@cse.iitd.ac.in.

XXXX-XXXX/2018/3-ART39 $15.00
https://doi.org/0000001.0000001

on them as directed by the instructions in the executable. Simultaneously, the simulator also models the specified hypothetical test processor architecture, and exercises its structures as directed by the emulation, to give performance and power numbers for the test architecture. gem5 [3] is an example of an execution-driven simulator. A trace-driven simulator takes as input a trace that captures the emulation of the executable. This trace can be produced by a separate emulator running in parallel, such as in Sniper [5] and Tejas [12], which use Intel PIN [1] as the external emulator. The trace can also be made available as a stored file that is generated by an external emulator. Both classes of simulators are popular.

Simulators can also be classified as cycle-level or approximate. Cycle-level simulators exercise each structure of the processor, at the cycle level, for all the benchmark instructions that need to be simulated. Approximate simulators choose a sampling approach where they simulate at the cycle level for a brief sampling period, and then extrapolate the statistics to account for a larger number of cycles / instructions. This is based on the insight that a workload typically exhibits similarity in its utilization of the processor resources for large phases. Again, both classes of simulators are widely employed in computer architecture research.

In this work, we focus on improving the performance of trace-driven, cycle-level simulators. The proposed approach can be extended to execution-driven simulators as well, but we do not evaluate that here. The proposed approach is orthogonal to a sampling-based approximate approach, and can be employed to complement the latter.

Modern chips are typically multi-core, and it is common for research groups to have multiple servers, giving them a significant amount of parallel compute capability. A simulator (of a single thread of execution) however runs on a single thread and cannot exploit the available infrastructure. We wish to parallelize this single simulator thread. Our proposed approach is inspired by the work by Nguyen et al. [8]. The essential idea is to break a benchmark thread into contiguous *chunks* of execution, and then simulate each chunk in parallel. As a trivial example, if we are interested in simulating the first billion instructions of a single threaded benchmark, we can simulate it as two chunks: one instance of the simulator simulates the first five hundred instructions, while a second instance simulates the next five hundred million. *Ideally*, this would double the simulator's performance. This technique can be extended to N chunks to get a speedup of N times.

Two issues exist with this approach. Firstly, simulating in this manner induces error. The initial state of the simulated processor is incorrect for all the chunks other than the first. Thus, measures have to be taken to reduce this error to acceptable amounts. Secondly, the decomposition into chunks cannot be done arbitrarily. It must be done in a manner such that the time required to simulate each chunk is exactly the same, thereby providing the maximum possible speedup. In this work, we explore solutions to these two issues, and show that this approach of chunking is a promising technique for improving simulator performance.

In Section 2, we describe the structure of a trace-driven, cycle-level simulator, and the basic chunking methodology in the context of the described simulation model. We then describe an offline approach to chunking based simulation in Section 3, for both single and multi-core test architectures. In Section 4, we describe an online approach to chunking simulation. In Section 5, we describe approaches to reducing the error induced by the chunking process. We then evaluate the various chunking techniques and error reducing approaches in Section 6. Finally, in Section 7, we discuss alternate approaches mentioned in the literature to increasing the speed of architectural simulators, all of which may be complemented by our proposed approach.

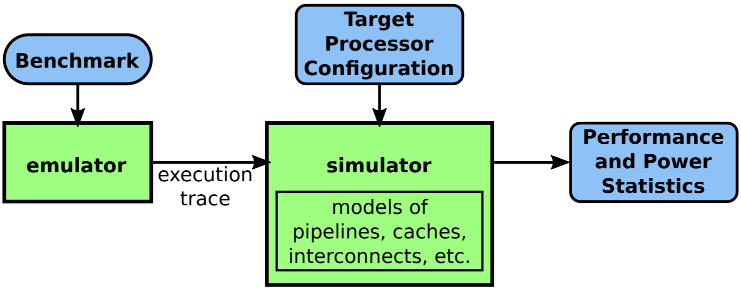
Benchmark
Target Processor Configuration
emulator
execution trace
simulator
models of pipelines, caches, interconnects, etc.
Performance and Power Statistics

Fig. 1. Trace-Driven Simulator: An Overview

2 BACKGROUND

2.1 Trace-driven Cycle-level Simulators

Figure 1 provides a high level description of a trace-driven simulator. To avoid confusion, we will refer to the complete architectural *"Simulator"* with a capital 'S', while its constitutent entities as "emulator" and "simulator" (small 's'). The emulator executes the benchmark and generates a trace that describes the execution. The emulator can be a process running simultaneously with the simulator, or can be run once and the generated trace can be stored in a file. In both cases, the trace contains instructions of the benchmark, in the order in which they were executed. In the latter file-based case, we will assume that a separate process reads the file and forwards the read trace to the simulator, and call this process the emulator, so as to generalize the discussion.

Along with the executed instructions, the trace also records the addresses at which memory operations were performed by the benchmark, as well as the outcome of branch instructions. This information is sufficient to simulate the execution of the benchmark on the test architecture. Given the trace, the simulator exercises the architectural models, configured according to the user's specification, and provides the required performance and power statistics. Note that the simulator in a trace-based Simulator need not simulate the functionalities of the different processor components. Rather, it focuses only on capturing the timing and power consumption aspects. For example, when performing a load, the actual value loaded is of no concern. Rather, the address is important, as it tells us the closest cache level in the memory hierarchy where the cache line is present. This tells us the time needed to perform the load operation, and consequently when instructions dependent on the load may be issued by the pipeline.

Note that the simulation step is the determining factor of the performance of the Simulator. The emulator is typically two orders of magnitude faster than the simulator. But since the emulator's output, that is, the trace must be consumed by the simulator before the former can produce more, the emulator is forced to slow down and operate at the speed of the simulator.

Most popular Simulators (such as Sniper [5], gem5 [3], and Tejas [5], with some minimal changes) support the following four modes of operation:

(1) `Full-Simulation`: This is the default mode of operation for a Simulator. Every aspect of every instruction is simulated – all architectural structures to be exercised are done so. All statistics are collected. The emulator forwards details of every benchmark instruction.
(2) `Full-Simulation-No-Statistics`: Here again, every aspect of every instruction is simulated. However, in this mode, statistics are not collected. The emulator forwards details of every benchmark instruction. The simulation speed is the same as that of `Full-Simulation`.

(3) `Structures-X-Only`: In this mode, only the structures mentioned in list `X` are exercised during simulation. Only the functionality is simulated, and not the timing. No statistics are collected in this mode. The emulator forwards only those instructions that will influence the listed structures, and drops everything else. For example, in the mode `Structures-LLC-BPred-Only`, the simulator only exercises the last level cache (LLC) and the branch predictor. It does not capture the timing. It simply updates the set of addresses present in the LLC when simulating memory accesses, and the predictor tables when simulating the branch predictor. Additionally, the emulator forwards details of only the branches and memory instructions. All other instructions are simply dropped. The number of instructions processed per second when running in the `Structures-LLC-BPred-Only` mode is an order of magnitude higher than that of `Full-Simulation`.

(4) `Fast-Forward`: All instructions are dropped by the emulator in this mode, and no statistics are collected. The number of instructions processed per second is two orders of magnitude higher than that of `Full-Simulation`.

2.2 Chunked Simulation

We now formally describe the basic idea of chunked simulation. Let us assume a single thread of execution of N_{inst} instructions. Let us number the instructions executed by the emulator sequentially from 0 to $N_{inst} - 1$. We then decompose the thread of execution into N_{chunk} chunks of contiguous instructions. Each chunk $Ch_i, 0 \leq i < N_{chunk}$, starts from instruction SI_i, and ends at EI_i, with $SI_i < EI_i$. Also, $\forall i, 0 < i < N_{chunk}, SI_i = EI_{i-1} + 1$. N_{chunk} instances of the Simulator are spawned: one for each chunk. A Simulator instance Sim_i operates in the `Fast-Forward` mode for instructions 0 to $SI_i - 1$, and then in the `Full-Simulation` mode for instructions SI_i to EI_i, collects the statistics, and terminates. Once all N_{chunk} instances finish simulation, the gathered statistics are aggregated through simple summation to provide the final statistics for the entire benchmark.

2.2.1 *Induced Error, and Mitigation Techniques.* Consider the following benchmark application:

```
for(int i = 0; i < 100; i++)
  sum += i;
```

A *correct* simulation of a benchmark on the test architecture is obtained by simulating it entirely in the `Full-Simulation` mode. Now let us consider the case of a 2-way chunked simulation. The first instance of the simulator simulates half the instructions, which we will roughly equate to the first 50 loop iterations for ease of discussion. The second simulator instance simulates the last 50 iterations.

First, let us clarify that there is no functional correctness issue. As mentioned earlier, trace-based simulators do not concern themselves with actual values stored in registers and memory locations. Accurately calculating the sum of the first hundred integers, as described in the above benchmark, is not one of the objectives. Focus is instead on capturing timing, power, and other statistics accurately.

Now, it can be seen that the chunking approach induces an error in the statistics. Let us say that the array is in memory, and is brought to the L1 cache by the load instructions. Let us say that the line size is twenty integers long. So in the full benchmark simulation, there are five L1 misses. However, in the two-way chunking scenario, there are six L1 misses – three in the first chunk and three in the second. Thus, the timing and other statistics differ – an error has been induced in the simulation.

When we begin simulating a chunk Ch_i $(i \neq 0)$, the state of the simulated structures is different from that which exists at the end of simulating the first SI_i instructions in the `Full-Simulation` mode. In the above example, the difference in state was illustrated in the L1 cache. Consequently, there is a difference in the collected statistics as well. For chunked simulation to be employable, measures must be taken to mitigate this error.

Nguyen et al. [8] proposed to have overlapping chunks. In terms of the above defined notation, their proposal functions as follows. We first simulate in the `Fast-Forward` mode for instructions 0 to $SI_i - 1 - W$, for some constant warmup size W, then in the `Full-Simulation-No-Statistics` mode for instructions $SI_i - 1 - W$ to $SI_i - 1$, and then in the `Full-Simulation` mode for instructions SI_i to EI_i. The W instructions simulated in the `Full-Simulation-No-Statistics` mode help "warmup" the state of architectural structures to one that is closer to that which will exist at the end of a correct simulation of the first SI_i instructions. For instance, in the above benchmark, warmup would constitute bringing in the relevant cache line in to the L1 cache before the statistics are collected in the second simulator instance. This reduces the error induced by the chunked approach. However, determining the value of W is non-trivial. Too small a value of W increases the error. At the same time, too large a value reduces the speedup obtained through chunking. The warmup period is essentially overhead – during the simulation of $N_{chunk} \times W$ instructions, statistics are not collected. The appropriate value for W is also heavily benchmark dependent. Nguyen et al. [8] suggest an approach to estimate the least value of W such that the error is relatively low. They propose to utilize the average memory access frequency of the workload, along with the average hit rate, to guess the minimum number of instructions that need to be simulated such that the caches can be populated with the relevant lines. Such apriori knowledge may be available in many cases, when the same benchmarks need to be simulated against different configurations where the memory system remains relatively unchanged. However, this may not always be the case. It may be needed to simulate new benchmarks, and/ or it may be needed to change the simulated memory system in each simulation. Thus, a better alternative to warmup the system state is required.

2.2.2 Reduced Speedup due to Equal-Sized Chunks. The speed of simulation, that is the number of instructions simulated per second, is not dependent on the native machine alone. It is also heavily dependent on the nature of the benchmark. Consider the simulation of the first billion instructions on the benchmark "bzip2" from the SPEC CPU2006 suite. Figure 2(a) shows the speed of simulation, in kilo-instructions per second, recorded for each slice of 100000 instructions, and Figure 2(b) shows the variation in the instructions executed per cycle (in the simulation). Figures 2(a) and 2(b) clearly show the variation in simulation speed as the benchmark progresses through different phases. Thus, a naive approach of having chunks of equal number of instructions, that is, $SI_i = \frac{N_{inst}}{N_{chunk}} \times i$,

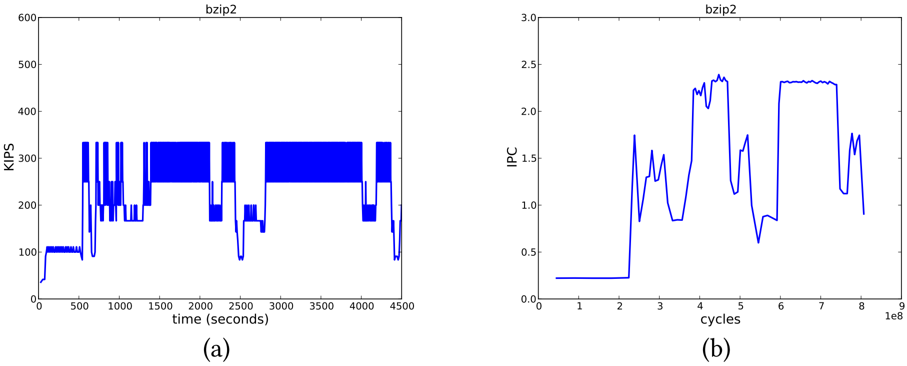

Fig. 2. Relationship between simulation speed and the program phase for the bzip2 benchmark

and $EI_i = SI_i + \frac{N_{inst}}{N_{chunk}}$, may not produce the best performance. Each resulting chunk may then require different amounts of time to simulate. Since the time of simulating the entire benchmark is effectively the time taken by the longest chunk (in terms of simulation time), the achieved speedup may be sub-optimal. Thus, measures need to be taken to ensure that the simulation workload is distributed *equitably* among the different Simulator instances such that they all finish at the same time, and provide higher speedup.

2.3 Contributions of this Work

Given the shortcomings of the prior approach to chunked simulation, we make the following contributions:

- We propose an approach to chunking that attempts to divide the simulation into chunks that require near-equal time to simulate. This enables near-maximum speedup. We call this approach "offline chunking" (Section 3).
- We extend this offline approach for serial workloads, to parallel ones. Chunking-based simulation of parallel benchmarks has not been studied before.
- The offline approach gives good speedups when simulating configurations that are not too different from a base reference run. However, if the configuration to be simulated is very different, then the offline approach gives modest speedups. We thus propose an "online task-stealing approach" for single threaded benchmarks, that does not require a reference run (Section 4.2).
- The task-stealing approach cannot be applied for parallel benchmarks, because the definition of a task is more complicated in a parallel benchmark. We propose a sampling based approach to determining the task definition, which is heaviliy determinant on the program phase, and then propose a parallelized simulation of tasks (Section 4.3).
- The warmup mechanism in prior work has shortcomings as detailed above. We propose a new warmup technique that is based on the principle that different on-chip components require different warmup approaches. This approach allows for reduction in error, while having a minimal effect on speedup (Section 5.2).

3 OFFLINE CHUNKING

This approach works for both single core and multi core test architectures. The basic idea is to have a reference single run of the entire benchmark, in which we periodically record the number of instructions simulated in each thread. This would then allow us to decompose the workload into chunks such that each chunk takes the same time to be simulated.

3.1 Methodology

We first perform a single reference run that will provide us with the required information regarding the progress of each of the benchmark threads over time. Two important facets need to be realized, with the second one being done as a consequent of the first one being achieved. First, for each thread t, given a time τ, we would like to know the number of instructions of thread t that have been simulated in the first τ seconds of simulation. Second, we would like to know the number of instructions n_2 of thread t_2 that have been simulated in the time a given thread t_1 has simulated n_1 instructions.

In our reference run, we periodically record the number of instructions simulated per thread. This helps us realize the function `NumInst`(t, τ) that returns the number of instructions of thread t

simulated in the first τ seconds of the reference simulation. Through this function, we realize the aforementioned facets.

Now, given the function `NumInst`, we devise an algorithm to divide the simulation workload into equal chunks. Let us denote the total time taken for the simulation by T_{total}, and the number of threads in the benchmark by N_{thread}. We then follow Algorithm 1 to decompose the simulation workload and appropriately spawn the N_{chunk} Simulator instances. The `Spawn` function takes the starting instruction number and ending instruction number of each benchmark thread. It performs forward/ warmup to the start of the chunk. Now that the architectural state is close enough to ideal, it then performs simulation in the `Full-Simulation` mode, where the statistics are collected. We will discuss different warmup alternatives in Section 5.

ALGORITHM 1: Offline Decomposition into Chunks

```
for $n_{chunk} \in [0, N_{chunk})$ do
    $S\_array$ = [];
    $E\_array$ = [];
    for $n_{thread} \in [0, N_{thread})$ do
        $SI$ = NumInst($n_{thread}, \frac{T_{total}}{N_{chunk}} \times n_{chunk}$);
        append $SI$ to $S\_array$;
        $EI$ = NumInst($n_{thread}, \frac{T_{total}}{N_{chunk}} \times (n_{chunk} + 1)) - 1$;
        append $EI$ to $E\_array$;
    end
    Spawn($S\_array, E\_array$)
end
Function Spawn($S\_array, E\_array$)
    start Simulator instance;
    instantiate simulated processor of $N_{thread}$ cores;
    simultaneously simulate the $N_{thread}$ threads as follows;
    for $n_{thread} \in [0, N_{thread})$ do
        if $S\_array[n_{thread}] > 0$ then
            simulate first $S\_array[n_{thread}] - 1$ instructions of thread $n_{thread}$, according to the chosen
              warmup technique, on core $n_{thread}$;
        end
        simulate from instruction $S\_array[n_{thread}]$ to $E\_array[n_{thread}]$ of thread $n_{thread}$, in
          Full-Simulation mode, on core $n_{thread}$;
    end
```

A naive approach of decomposing into equal-sized chunks [8] will result in sub-optimal speedups in most workloads, as discussed in Section 2.2.2. The speedup will be optimal only when the benchmark shows uniform behavior, without any change of phase, for its entire run. Not many realistic applications operate in a single phase. The proposed offline algorithm attempts to find the best "*split*" of the benchmark, taking into account its various phases. Also, it must be noted that even in the case of a benchmark with uniform behavior throughout, the proposed offline algorithm provides the optimal split – that is, into equal-sized chunks.

It can be seen that, if configured prudently, the approach of chunking will always result in a speedup. Firstly, like any parallel program, the degree of parallelism N_{chunk} will have to be selected keeping in mind the compute resources available. Increasing N_{chunk} improves the speedup to a certain extent; further increase sees diminishing benefits to the pressure on shared resources like the shared cache, and the main memory. Secondly, the chunking approach increases the amount of work done – each simulator instance, aside from simulating its assigned chunk in the

Full-Simulation mode, also simulates some instructions in the warmup phase. This additional work constitutes an overhead, which reduces the speedup. Thus, the warmup strategy will also have to be chosen prudently. It is important to note that since Fast-Forward (two orders of magnitude) and Structures-X-Only (one order of magnitude, for typical structure sets) are significantly faster than Full-Simulation mode, the overhead of warming up can be brought down significantly. Warm up strategies are discussed in detail in Section 5.

3.2 Limitations of an Offline Approach

The offline approach ensures we get the maximum possible speedup since the workload is evenly distributed among the chunks. However, there are important caveats in the approach. The decomposition is based on the per-thread rates of progress recorded in the reference run. However, the rates of progress may change if the test architecture changes. Consequently, the decomposition points also change. This has a two-fold impact: first, the points determined by this approach may be illegal. For instance, the point may suggest that thread t_1 simulates n_1 instructions in the same time that thread t_2 simulates n_2 instructions. However, under the architecture to be tested, it may be that thread t_2 simulates n_2' instructions in the time that t_1 simulates n_1 instructions. Thus, an error is induced in the simulation. Note that this error is induced only in the case of multi-core test architectures, and not single core ones. Second, the points determined may result in unequal distribution of the simulation workload with a different test architecture, thereby producing a sub-optimal speedup. Design space exploration will, by definition, require the study of many different configurations, resulting in sub-optimal speedups in all studies. Therefore, we need a decomposition approach that does not require a reference run.

For reasonable changes in the configuration, the offline approach still produces appreciable speedups with modest errors to remain a viable candidate for chunked simulation (see Section 6.5). Additionally, the offline approach proves useful in providing a reference against which online approaches may be compared.

4 ONLINE CHUNKING

In this section, we will discuss online approaches to chunked simulation, that do not employ any reference runs, and are consequently expected to have wider applicability.

4.1 Naive Approach

We first develop a naive chunking scheme, online-naive, that simply decomposes the workload into chunks of equal number of instructions, without taking any application phase behavior into account. We simply spawn N_{chunk} instances of the Simulator, each instance Sim_i responsible for the simulation of chunk Ch_i. A chunk Ch_i is defined by $SI_i = \frac{N_{inst}}{N_{chunk}} \times (i-1)$, and $EI_i = \frac{N_{inst}}{N_{chunk}} \times i - 1$. This is essentially as proposed by Nguyen et al. [8]. Since the changing phase behavior is not taken into account, this approach gives sub-optimal speedups as discussed in Section 2.2.2.

4.2 Task-Stealing Approach for Serial Workloads

Here we develop the online-taskStealing algorithm by following the popular task-stealing paradigm that helps achieve a more equitable distribution of the workload.

We first decompose the serial workload into N_{task} small tasks, each of size S_{task} instructions. Just like chunks, tasks also are made up of contiguous instructions, and two tasks do not have any overlap with each other. $N_{task} >> N_{instance}$, where $N_{instance}$ is the number of Simulator instances. Each Simulator runs Algorithm 2 to cooperatively divide the task of simulating the benchmark among themselves. They have an integer variable TASK_INDEX (initialized to 0) shared among themselves, that gives the index of the next task to be simulated. Each Simulator instance atomically reads and increments the value of `TASK_INDEX`. Based on the value read, it performs forward / warmup to reach the corresponding point in the benchmark's execution, and simulates S_{task} instructions in the `Full-Simulation` mode. It then attempts to read `TASK_INDEX` again, and exits if `TASK_INDEX` $> N_{task}$.

Note that an instance does not exit after every task it simulates. Rather, it begins a warmup phase from that point itself.

ALGORITHM 2: Online Task-Stealing based Decomposition for Serial Benchmarks

```
$emulator\_at\_instruction = 0$;
while TASK_INDEX< $N_{task}$ do
    $task\_to\_simulate$ = read_and_increment(TASK_INDEX);
    forward / warmup for ($S_{task} \times task\_to\_simulate - emulator\_at\_instruction$) instructions;
    simulate in Full-Simulation mode for $S_{task}$ instructions;
    $emulator\_at\_instruction = (task\_to\_simulate + 1) \times S_{task}$;
end
```

This task-stealing approach, just like the offline approach, helps us avoid the caveat of equal-sized chunks. It ensures that the workload is more equitably distributed among the Simulator instances. This expectedly gives higher speedups, as discussed in Section 6.2.1.

Deciding the value of S_{task} is non-trivial. A lower value gives a finer granularity, ensuring a more equitable distribution of the simulation load. However, this involves running the task-stealing algorithm (Algorithm 2) more often. This will reduce performance. Also, a finer granularity reduces the chances of any given Simulator instance working on long contiguous sequences of instructions. This increases the simulation error. Our studies regarding the value of S_{task} are presented in Section 6.2.4.

4.3 Online Chunking for Parallel Workloads

The approach for parallel workloads is more involved. The simple approach for serial workloads cannot be directly applied because the task boundaries are not as clear. As an example, let us say a Simulator instance has to simulate task number i. In the serial workload case, it was clear that the Simulator must begin simulating from instruction ($S_{task} \times i$). However, in the parallel workload case, each thread's start point is not clearly known. It is possible that all threads are functionally similar, and so display a similar throughput. So, we could simply start each thread from the same instruction number – we could simply fast-forward/ warmup each thread to $S_{task} \times i$ number of instructions, and then simulate S_{task} number of instructions on each thread in `Full-Simulation` mode. However, benchmarks may have threads that are asymmetric – each thread may perform a slightly different functionality. Additionally, even in the functionally similar scenario, threads may communicate with each other through constructs like spin-locks and barriers. These affect their throughput, and make it less likely that each thread progresses in a near-identical fashion. Thus, it would be incorrect to start all threads from the same instruction number. Suppose we have a 2-threaded benchmark, with thread 1 having twice the throughput of thread 2, then warming up each thread of task i for $S_{task} \times i$ instructions, and simulating each thread for S_{task} instructions, would be incorrect. Thus, we must first be able to predict the task boundaries accurately, to achieve an online approach for simulating parallel benchmarks.

We propose an online sampling-based approach to achieve this. We first employ a single Simulator instance that samples the workload, in order to identify a stable behavioral phase. A stable phase is one where the relative throughputs of the different threads change negligibly as execution progresses. We refer to this identified behavior as the "`phase_definition`": it is a vector comprising of the number of instructions executed by each benchmark thread in the time taken for benchmark thread 1 to execute Q instructions, where Q is a parameter to the algorithm. Once, we have identified the phase definition, we spawn $N_{instance}$ number of Simulator instances, that simulate tasks in parallel. Given a `task_to_simulate`, and the `phase_definition`, a Simulator instance fast-forwards/ warms up to the corresponding start point, and executes a single task from there. The Simulator instances keep doing this until (i) they reach the end of simulation, or (ii) they identify that the behavior of the benchmark has deviated from the `phase_definition`. In the latter case, the sampling operation, and the steps mentioned above, are repeated.

A benchmark with longer predictable phases will be more amenable to the online approach, as opposed to one that changes its behavior often, as the latter demands frequent sampling, which reduces the speedup. The observations are presented in Section 6.3.2.

ALGORITHM 3: Online Chunking based Simulation for Parallel Benchmarks (Part 1 of 2)

```
Function main()
    $cur\_state$ = initialize state;
    $simulation\_done$ = False;
    while $simulation\_done = False$ do
        $< cur\_state, phase\_definition >$=sample($cur\_state$);
        $< cur\_state, simulation\_done >$=guide_simulation($cur\_state, phase\_definition$);
    end
    report statistics from $cur\_state$;
Function sample($cur\_state$)
    $cur\_candidate$ = 0;
    while True do
        $new\_state$ = simulate in Full-Simulation mode from $cur\_state$ until thread 1 executes $Q$
          instructions;
        $candidate\_phase\_definition[(cur\_candidate++)\%window\_size]$ =
          $new\_state.N_{executed} - cur\_state.N_{executed}$ //$window\_size$ is a parameter to the algorithm;
        $cur\_state = new\_state$;
        if $cur\_candidate \geq window\_size$ then
            conduct a chi-square homogeneity test taking each $candidate\_phase\_definition$ as a population,
              and the per-thread executed instruction counts as categories;
            if the contents of the window are homogeneous then
                return $< cur\_state, candidate\_phase\_definition[0] >$ //stable phase found;
            end
        end
    end
```

Algorithm for Online Chunking based Simulation for Parallel Workloads. Algorithms 3,4 give a formal description of the proposed approach. The *state* of the simulation is defined as a tuple of the different statistics: the number of instructions executed by each benchmark thread, $N_{executed}[]$, the number of simulated cycles, the number of hits and misses at each of the caches, to name a few.

Let us define the addition and subtraction operators (+/−) for the $N_{executed}$ vector as follows: for three states A, B, and C, $C.N_{executed} = A.N_{executed} + (-)B.N_{executed}$ results in $C.N_{executed}[i] = A.N_{executed}[i] + (-)B.N_{executed}[i], 0 \leq i < N_{threads}$, where $N_{threads}$ is the number of threads in the parallel workload. Let us define the multiplication operator (×) for the $N_{executed}$ vector as follows: for two states A and B, $B.N_{executed} = scalar \times A.N_{executed}$ results in $B.N_{executed}[i] = scalar \times A.N_{executed}[i], 0 \leq i < N_{threads}$.

ALGORITHM 4: Online Chunking based Simulation for Parallel Benchmarks (Part 2 of 2)

```
Function guide_simulation($cur\_state, phase\_definition$)
    while True do
        TASK_INDEX = 0;
        for $n\_instance \in [0, N\_instance)$ do
            $start\_state_{n\_instance}$ = fast-forward/ warmup (in parallel with other instances) Simulator
              $n_{instance}$ to $cur\_state.N_{executed}$ + TASK_INDEX × $phase\_definition$;
            Run Simulator $n_{instance}$ (in parallel with other instances) in Full-Simulation mode until
              $phase\_definition$ number of instructions are simulated on the threads;
            TASK_INDEX + +;
        end
        for $n\_instance \in [0, N\_instance)$ do
            $end\_state$ = wait for Simulator $n\_instance$ to complete;
            $executed\_task = end\_state.N_{executed} - start\_state_{n\_instance}.N_{executed}$;
            if executed_task and phase_definition are not homogeneous then
                return < $start\_state_{n\_instance}, False$ >;
            end
            else
                $cur\_state = end\_state$;
                if end of simulation then
                    return < $cur\_state, True$ >;
                end
            end
        end
    end
```

5 WARMUP TECHNIQUES

A naive way of warming up would be to simply run the simulator instance in Full-Simulation-No-Statistics mode up to the SI^{th} instruction. This would ensure the ideal state to begin simulation of the assigned chunk, thus, resulting in zero error. However, the time taken by Full-Simulation-No-Statistics mode is the same as Full-Simulation, and so the time taken to simulate the last chunk is the same as the base case. Thus, the speedup is zero.

We describe two more useful approaches to warming up the simulated architectural structures such that the error induced due to the chunked approach is reduced, while the speedup is not compromised as much as the naive approach.

5.1 WT1: Fast-Forward, followed by Full Functional Warmup

The first warmup scheme WT1 is similar to that proposed by Nguyen et al. [8]. To simulate a chunk from the SI^{th} instruction to the EI^{th} instruction, we first run in Fast-Forward mode for $SI - W$ instructions, where W is some pre-defined constant number of instructions. We then simulate for W instructions in the Full-Simulation-No-Statistics mode to warmup the simulated structures. We then simulate for $(EI - SI)$ instructions in the Full-Simulation mode.

5.1.1 Shortcoming. The issue with this approach is the determination of the right value for W. A small value increases simulation error, while a large value reduces performance. The ideal value of W is benchmark behavior dependent and therefore fixing it to a single value does not work well across all benchmarks, as discussed in Section 6.4. We have evaluated the efficacy of different warmup sizes across different benchmarks.

5.2 WT2: *Structures − X − only*

The need for warmup is to achieve a processor state as close as possible to that which would exist if Full-Simulation was done up till that point. Processor state is defined as the contents of all the storage elements: the private and shared caches, the main memory, and in the pipeline, the register files, the reorder buffer, the load-store queue, the instruction window, the rename tables, and the branch predictor tables. In a trace-driven simulator, in line with the discussion in Section 2, processor state is defined slightly differently. For caches (we restrict our discussion to on-chip components for the sake of brevity, and hence don't talk about the main memory), the processor state is the set of line addresses present. The line contents do not matter. For register files, the processor state is the ready status of the different registers. For the remaining structures, which are all pipeline elements, the processor state is the contents themselves.

Now let us look at what it takes to build up the state of these different structures. At one end of the spectrum, we have the different pipeline structures and the register files. These are very small structures and are updated very frequently during execution. Consequently, the state at instruction number i, with very high probability, is the result of only updates made in the last few hundred cycles or so. In other words, only a couple of hundred instructions preceding instruction i contributed towards the state at i. Older instructions' effects are masked. At the other end of the spectrum, we have the last level cache. This is a large structure, and is not updated very frequently. Consequently, updates made hundreds of millions of cycles before instruction i's execution also have a role in the state of the LLC at i.

In the context of warming up, this mode of classifying structures is important. For example, suppose we wish to simulate a chunk of 20 million instructions starting from the 500 millionth instruction. We only need to warm the register files up from the 499999000^{th} instruction, as a 1000 cycle warm up is more than sufficient to get a good starting register file state. Alternatively, we can even skip warming up the register files completely. The first 1000 instructions of the simulation will differ from the ideal. But the remaining 19999000 instructions will match the ideal. Thus, the error is quite minuscule.

But when warming up the LLC, the approach needs to be different. A simple 1000 cycle warmup will hardly produce a satisfactory cache state. The warmup needs to be much longer. Likewise, we cannot afford to skip the warming up of the LLC. A large warmup is mandatory.

We have studied this non-homogeneous approach of warming up only selected structures. Our results are presented in Section 6.4. This approach, termed WT2, involves simulating the first $(SI - 1)$ instructions in the $Structures - X - Only$ mode, and the next $EI - SI$ instructions in the Full-Simulation mode. We have evaluated for different sets for X, and found that the LLC and the branch predictor tables are the most critical. These results are presented in Section 6.4.

WT2 ensures low error regardless of the benchmark behavior, but can potentially result in lower speedups. We compare and contrast the different variations of WT1 and WT2 in Section 6.4.

6 EVALUATION

6.1 Evaluation Set-up

6.1.1 Simulated Architecture. A 16 core processor was simulated. Each core's microarchitecture was similar to the Intel Skylake architecture. A shared last level cache of 8 MB was used. The full architectural details are given in Table 1.

6.1.2 Simulated Workload. For the single threaded workloads, we simulated the SPEC CPU2006 benchmark suite. In the case of each benchmark, the first one billion instructions were simulated. For multi-threaded benchmarks, the PARSEC suite was used. Each benchmark was made to spawn 16 threads. A cumulative instruction count of one billion was simulated. We use the cycle-accurate

Table 1. Simulated processor architecture

Parameter	Value	Parameter	Value
System Configuration			
Cores	16	Microarchitecture	based on Intel Skylake
Technology	14 nm	Frequency	3.4 GHz
Core Configuration			
Retire Width	4	Issue Width	4
Private L1 i-cache, d-cache			
Size	32 kB	Latency	3 cycles
Private L2 Unified Cache			
Size	256 kB	Latency	8 cycles
Shared Elements			
L3 cache	8MB / 24 cycles	Main Memory Latency	120 cycles
Network-on-Chip			
Topology	2-D Torus	Routing Alg.	Simple XY
Flit size	8 bytes	Hop-latency	1 cycle
Router-Latency	2 cycles		

Table 2. Simulation platform

Parameter	Value	Parameter	Value
Processor	Intel(R) Xeon(R) CPU E5-4620	Microarchitecture	Sandy Bridge
Cores	48	Frequency	2.2 GHz
Last Level Cache	16 MB	Main Memory	64 GB

Tejas [12] simulator, which has been rigorously validated with native hardware. Note that for the serial benchmarks, the Intel PIN [1] front-end of Tejas was employed, while for parallel benchmarks, the file front-end of Tejas was used. This was done because with parallel benchmarks, the benchmark threads (16 threads per chunk) themselves would occupy all the cores in our simulation platform, leaving no scope for a speedup.

6.1.3 Evaluation Methodology. First a standard, non-chunked simulation is performed to serve as our base run. An N-way chunked simulation is performed next. The statistics of the N simulations are aggregated by simple addition of each individual statistic. The error is computed as $\frac{|IPC_{base} - IPCaggregated\_chunks|}{IPC_{base}} \times 100$. The speedup is computed as $\frac{time\_taken_{base}}{time\_taken_{longest\_chunk}}$.

6.1.4 Simulation Platform. All simulations were performed on a server containing Intel Xeon processors. The full details are given in Table 2.

6.2 Single Threaded Workloads: Offline v/s Naive Online v/s Task-Stealing Online

In this section, the experiments done have been performed with 8-way chunking. `WT2` warmup technique is used with the branch predictor and the last level cache being warmed up. For the task-stealing approach, a task size of 10 million instructions was used.

6.2.1 Speedup. Figure 3 shows the speedup achieved by employing the different chunking techniques. On average, we observed a maximum speedup of 5.39X. This speedup was observed with the *offline* approach. This approach achieved the highest speedup across all the benchmarks. This is as expected, as the `offline` approach, by design, decomposes the simulation workload amongst the different chunks *equitably*.

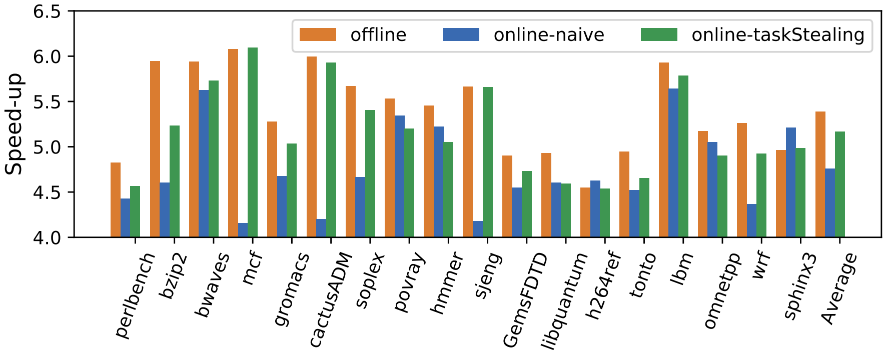

Fig. 3. Single threaded benchmarks: Speedup comparison of the different schemes

Online-naive performs worse than the *offline* approach, offering an average speedup of 4.76X since it adopts a naive decompostion premise. It disregards the phase-wise changes in simulation speed, and simply divides the work based on the number of instructions to be simulated.

`Online-taskStealing` performs much better than `online-naive` as the task-stealing approach helps achieve a near equitable distribution of the simulation workload. It comes close to the ideal speedup of the `offline` approach with a speedup of 5.17X – that is, within 4% of the ideal speedup. Importantly, it achieves this speedup with no prior knowledge of the benchmark's behavior, which is a requirement of the `offline` approach.

`Online-taskStealing` does not match the speedup of the `offline` approach because of three reasons. First, since the tasks have large sizes with millions of instructions, the workload distribution is not exactly equal. Second, the inter-process communication between chunks takes some time. Third, and most importantly, the amount of time spent in the warmup phase is much greater than in the `offline` approach. This can be explained through an illustrative example. Let us consider a simple 2-way chunking of a 1 billion-instruction simulation, while warming up using the `WT2` technique. Assume that the `offline` approach breaks the simulation into two simulations of 600 million and 400 million instructions, the number of instructions used for warmup are 600 million (0 instructions for the first chunk, 600 million for the second). Now, let us assume a task-stealing approach with task sizes of 10 million. Let us further assume that the two simulator instances simulate alternate chunks. This results in the first instance being in the warmup phase for 490 million instructions, and the second instance for 500 million instructions. Thus, a total of 990 million instructions have gone through the warmup phase. This additional work done by the task stealing approach reduces its speedup.

6.2.2 Error. Figure 4 shows the errors observed under the different chunking techniques. On an average, we observed a maximum average error in simulation of 0.32%, and a maximum error of 0.8%. This is meager, and is of the order of the variation typically seen even while re-executing the application on the same native system. This maximum error was seen with the `online-taskStealing` approach. The errors with the other approaches are much lower. This is expected as the `online-taskStealing` approach sees many disjoint simulation phases while the other approaches see contiguous simulation phases. Thus, in the latter, there are fewer cases of a simulation beginning in a predicted state. In our 8-way chunking experiments, exactly 7 of the 8

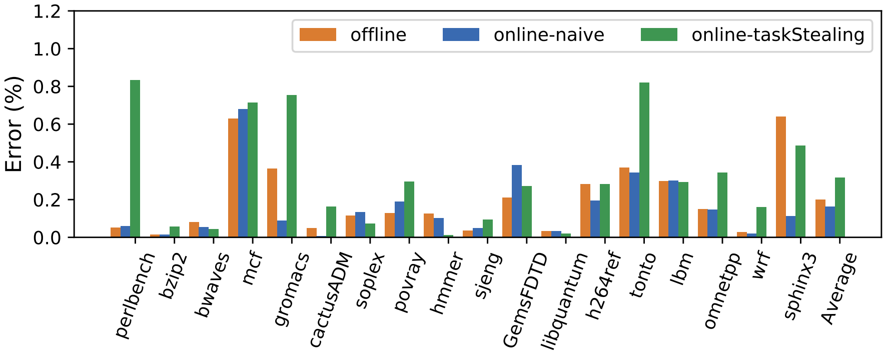

Fig. 4. Single threaded benchmarks: Error comparison of the different schemes

chunks begin execution in a predicted state. Whereas in the `online-taskStealing` technique, up to 99 tasks may begin in a predicted state (total tasks amount to 100).

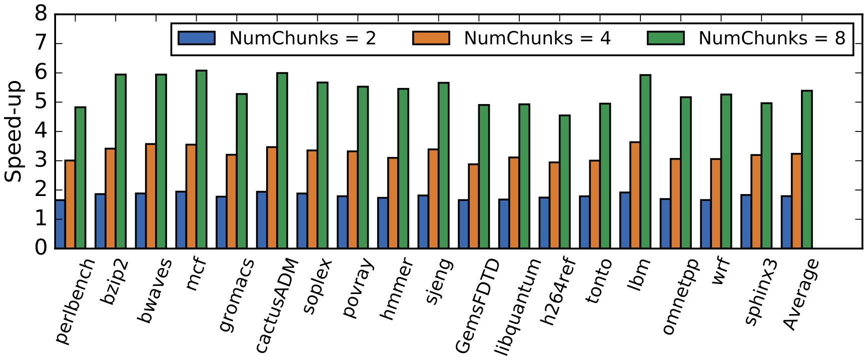

Fig. 5. Relationship between speedup and number of chunks

6.2.3 Analysis of Number of Chunks. We performed experiments to study how the speedup scales and how the error is affected when we increase the number of chunks. Figure 5 shows the scaling of the speedup. Note that the `offline` technique was used in these studies. We see that the speedup increases with increasing the number of chunks, but the relationship is sub-linear. This is because as the number of chunks increases, the pressure on the shared resources such as the shared last level cache and the bus to the memory increases. Thus, we do not see the full gains expected on increasing the degree of chunking. When increased to a 16-way chunking, we witnessed a slowdown in the simulation time due to the high levels of contention for the various shared resources.

Figure 6 shows the effect on the simulation error. With an increase in the number of chunks, the error increases as expected. This is because a larger number of instructions are now executed with an approximate processor state.

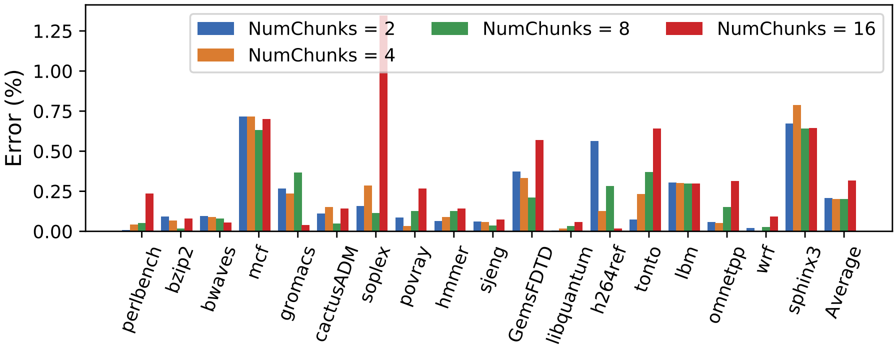

Fig. 6. Relationship between error and number of chunks

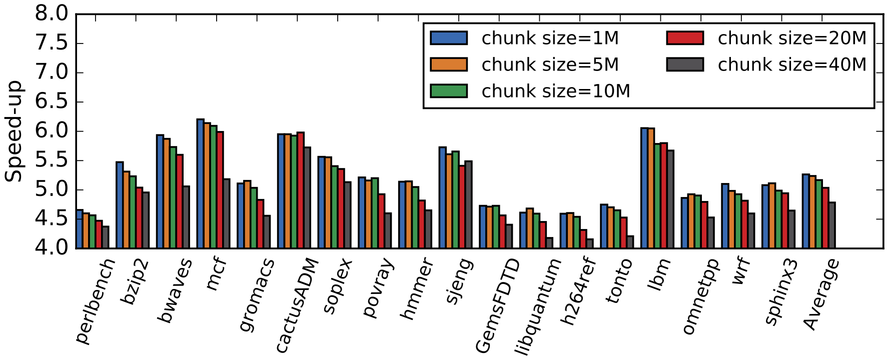

Fig. 7. Relationship between speedup and task size

6.2.4 Analysis of Task Size in the Task Stealing Approach. In the `online` task-stealing approach, we varied the size of the tasks and studied its effects on the speedup and the error. Figure 7 shows the variation in the speedup. As expected, as the task size increases, the observed speedup decreases. The larger granularity hinders the equitable distribution of the workload. With a task size of 1 million instructions, an average speedup of $5.26X$ was observed (within 2.4% of the `offline` approach).

Figure 8 shows the variation in the simulation error. Again, as expected, the error reduces with an increase in the task size. This is because with a larger task size, there are longer contiguous sequences of simulated instructions, and consequently, the number of instructions simulated at a predicted state are fewer. With a task size of 40 million instructions, an average error of 0.24% was observed.

We find a task-size of ten million instructions gives a good trade-off between speedup ($5.17X$) and error (0.32%), and use this for all the other experiments.

6.3 Multi-Threaded Workloads

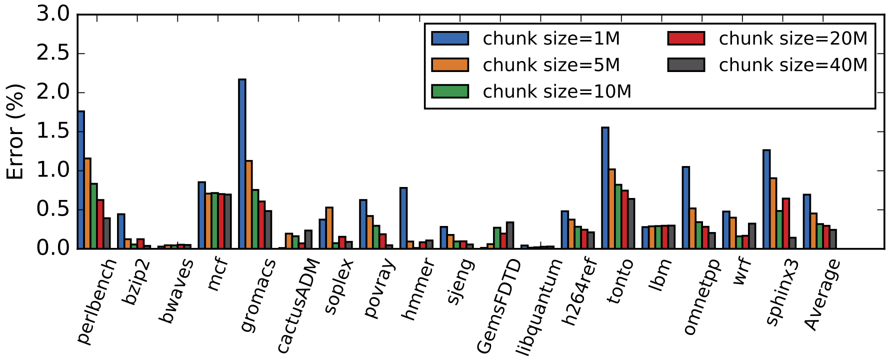

Fig. 8. Relationship between error and task size

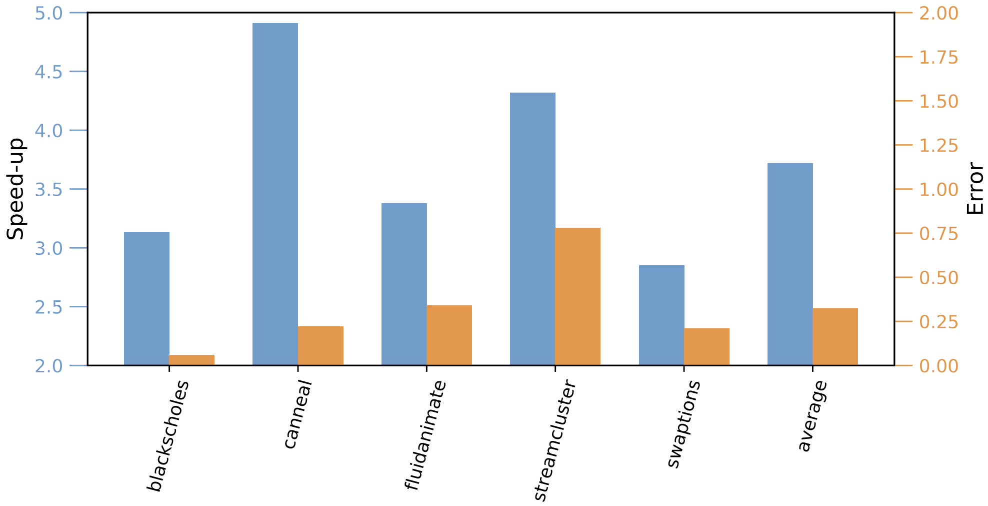

Fig. 9. Multi-threaded benchmarks: Offline Approach

6.3.1 Offline Approach. We studied the efficacy of the `offline` chunking scheme (8-way chunking) while simulating parallel benchmarks from the PARSEC benchmark suite [2]. Figure 9 shows the results. An average speedup of $3.72X$ was observed, and the average error was 0.32%.

6.3.2 Online Approach. We studied the performance of the `online` chunking scheme (8-way chunking) for simulating parallel benchmarks. We chose $Q = 10^7$, $window\_size = 5$ (see Section 4.3) for these studies. Figure 10 shows the results.

The average speedup was $2.53X$. The benchmark `streamcluster` could not be sped up. As discussed in Section 4.3, we first perform sampling to determine task definitions, and then proceed with chunking. However, with `streamcluster`, a stable phase was not recognized. The relative throughputs of the different threads varied too frequently for our scheme to define a task. Hence,

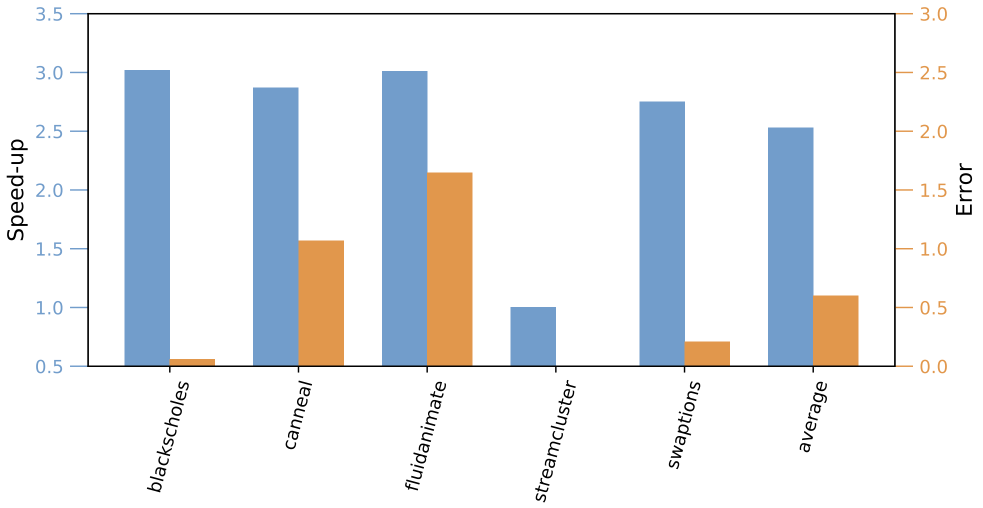

Fig. 10. Multi-threaded benchmarks: Online Approach

the entire simulation was performed by a single Simulator instance, leading to no speedup, and also no error. The benchmark `canneal` has a brief phase of inconsistent relative thread throughputs, before settling down in a stable phase. Our scheme simulates the inconsistent phase using a single Simulator instance, and the larger consistent phase using eight Simulators.

The average error observed was 0.6%. Note that when a chunk begins executing, 16 threads (each benchmark has 16 threads) begin from a predicted core state. This increases the amount of error induced. This is the reason behind the larger error shown by some benchmarks like *fluidanimate*.

6.4 Evaluation of Warmup Techniques

Figure 11 shows the comparison between different warmup schemes in terms of speedup and error. `WT1_X` is used to denote that warmup technique `WT1` was employed for `X` instructions. `WT2_X` is used to denote that warmup technique `WT2` was employed for all the structures listed in `X`. The 8-way `offline` chunking scheme was employed to simulate the SPEC suite of benchmarks.

6.4.1 Different warmup sizes of `WT1`*.* As expected, the speedup decreases as the warmup size W increases (see Figure 11).

Warmup sizes of both 10k instructions and 1M instructions resulted in large errors. Though the average errors are not alarmingly large, – 1.23% for a 10k warmup and 1.07% for a 1M warmup – errors as large as 5% were observed in some benchmarks (*sphinx3* benchmark, 10k warmup). This is because, as discussed earlier, the processor state in these benchmarks is determined by a much larger window of instructions – instructions that executed more than 1M instructions ago determine the processor state. Warming up for 100M instructions brings down the error to acceptable levels, but results in huge reduction in speedup – a mere $3.32X$.

6.4.2 Comparison of different `WT2` *variants.* The warmup strategy `WT2` gives us a better balance of speedup and error. The best configuration was when the branch predictor and the last level cache

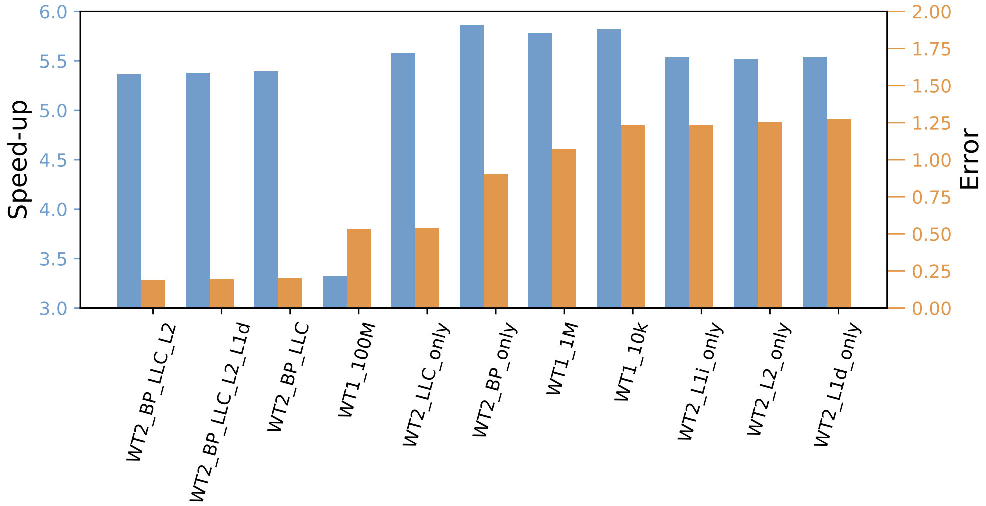

Fig. 11. Comparison of different warmup alternatives

were chosen for warmup. The speedup was 5.39X while the error was 0.2%. This is significantly better than any of the WT1 alternatives.

6.4.3 WT2 *for parallel benchmarks.* When warming up for parallel benchmarks, we found that the branch predictor and the LLC aside, warming up the coherence directory helped reduce the simulation error by 1% on average. Additionally, it is necessary to simulate the interactions between threads – barriers, acquiring and releasing of locks, etc. – during warm up, to ensure correct interactions between threads during regular simulation. Taking the case of barriers, during warmup, the simulator records the arrival of threads at a barrier. This is important because the start state for simulation could be one where some threads have reached the barrier and others have not. If we do not simulate barriers during warm up, then during actual simulation, it is possible that the benchmark threads synchronize on non-corresponding barrier invocations, leading to significant amount of error being introduced.

6.5 What-if Analysis

In the offline approach, a reference run is employed to decide on the chunk boundaries. However, this works best only when the simulation to be performed is with the same architectural configuration as the reference run. Simulating a different configuration may lead to sub-optimal splitting, resulting in reduced speedup. Since architecture research typically involves studying multiple processor design points, we evaluate the efficacy of the guided approach in the face of changing processor configurations.

To the base configuration described in Table 1, we introduced five changes to give five different configurations: (1) bimodal-BPred: the branch predictor was changed from a TAGE predictor to a simple bimodal predictor, (2) small-ROB: the reorder buffer size was reduced from 168 to 128, (3) tournament-BPred: the branch predictor was changed to a tournament predictor composed of a PAg and a PAp predictor, (4) 64kB-L1d: the L1 data cache size was increased from 32kB to 64kB, (5) 4assoc-L3: the L3 cache associativity was decreased from 8-way to 4-way. We performed offline chunking of these five configurations using a reference run based on the configuration described in Table 1.

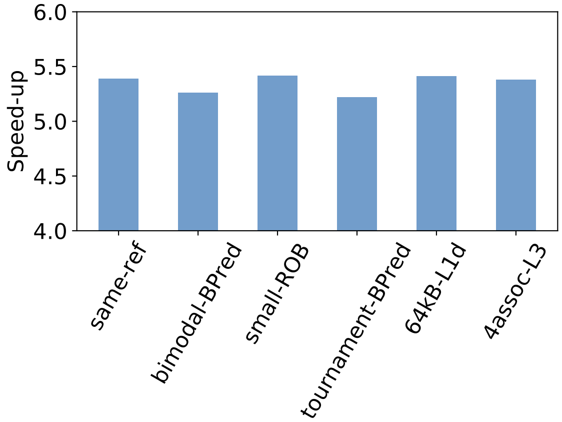

Fig. 12. What-if analysis

The resultant average speedups, across the SPEC suite, is as shown in Figure 12. The first bar `same-ref` denotes the average speedup obtained when the offline chunking approach is done with the same architecture configuration as the reference run. Thus, this bar serves as a reference. It can be seen that the losses in speedup in each of the different configurations is minimal. Thus, the offline approach, that is guided by a reference run, can be a useful simulation performance improving tool, when the configurations being explored have modest differences as compared to the reference run.

7 RELATED WORK

A large community of researchers work on increasing the speed of simulation, given the importance of Simulators in architecture research.

The first class of solutions involves sampling – representative portions of the benchmark are ascertained, and only these are simulated at the cycle-level. The statistics are extrapolated to give the picture of the entire benchmark. One way of doing sampling is through a stand-alone analysis of the benchmark. In this analysis phase, the benchmark is executed in its entirety, and representative portions and their “weights” are identified. Weights refer to the fraction of the execution that behaved in a manner captured by the representative portion. Simpoint [10] is a popular tool that provides such an analysis of benchmarks. It uses the SimpleScalar [4] Simulator to analyze the behavior of the benchmark. Pinpoints [9] is a tool by Intel, based on the Intel Pin instrumentation tool. It categorizes portions based on the methodology specified by the authors of Simpoint. Once such representative portions are identified, then during simulation, only these portions may be simulated, and the statistics appropriately weighted.

Another way of performing sampling is without a standalone analysis phase. We may perform cycle-level simulation of a portion, and then skip detailed simulation of all successive portions until some rare, phase-changing, event takes place, say for example, a last-level cache miss. All the skipped portions are expected to display behavior similar to the simulated portion. Such an approach is followed by the Simulator Sniper [5].

The second class of solutions is applicable when the workload is a parallel benchmark or a bag-of-serial-tasks. Each thread or task is simulated by a separate thread of the simulator. Since the action of cores and private caches are largely independent of each other, the simulation of the threads can run independently of each other in these phases. However, when shared elements like the shared caches, the network-on-chip, the directory and the main memory controllers have to be accessed, the simulator threads have to synchronize with each other. This synchronization affects the speed-up achieved through parallelization. Various strategies exist to bring down this penalty. Such an approach is adopted by Sniper [5], SlackSim [6], as well as the parallel version of the Tejas Simulator used in this work, ParTejas [7].

This family of solutions is similar in spirit to Parallel Discrete Event Simulators (PDES) where different components or modules or sub-systems are simulated in parallel, occasionally exchanging information. The parallel architectural simulators discussed above typically have disjoint subsets of target processor cores simulated by each parallel simulator instance.

A third class of solutions is to employ analytical models for structures deemed to have predictable behavior. Such an approach is adopted by ZSim [11], and a variant of Sniper [5]. In ZSim, the focus is on memory requests that reach the lower levels of the cache, which trigger the coherence protocols as well as shared resources like the last level cache. These portions are modeled in detail, while the working of the processor's compute core is modeled analytically.

Our approach can be used in conjunction with all of the above classes, providing a further increase in the simulation speed. For instance, when a sampling based approach is being used to reduce the number of instructions simulated, our proposed technique may be employed to reduce the time taken to simulate each sample. Similarly, when analytical models are used to approximately simulate some of the microarchitectural structures, our proposed algorithms may be used in conjunction to reduce the time taken for this approximate simulation.

Our approach involved decomposing the simulation of a single thread of the benchmark into contiguous chunks of instructions. This approach was first proposed by Nguyen et al. [8]. The authors recognized the error introduced because of the invalid architectural state that each chunk's simulation begins from. They proposed to have an *overlapping* of successive chunks, essentially resulting in the structures being warmed up according to the technique `WT1` (see Section 5.1). Although this reduces the error for some benchmarks, it is still quite high for others, as seen in our evaluation (Section 6.4). To bring the error down for all benchmarks, massive overlaps are required, that reduce the achieved speedup greatly. We proposed a novel warmup technique `WT2`(see Section 5.2) that dramatically brings down the error with little loss in performance. Nguyen et al. also did not address the issue of sub-optimal chunking that arises out of equal-sized chunks – that is, equal in terms of the number of instructions. Since the speed of simulation depends upon the nature of the workload, equal-sized chunks results in sub-optimal speedup. We address this issue by proposing two techniques – an offline approach that requires a reference simulation run (Section 3), and an online task-stealing based approach that requires no aprioiri knowledge (Section 4).

8 CONCLUSION

The performance and correctness of an architectural simulator is very important in determining the quality and timeliness of the research done using it. In this work, we revisit an elegant technique of chunking the simulation workload, and simulating the resultant chunks on separate cores. There were issues with both large simulation error and low achieved speedup in the original proposal. We solved these in this paper through novel simulation techniques. These techniques enable us to increase the speed of simulating single-threaded workloads. Average speedups of up to $5.39X$ were demonstrated (using 8-way chunking), while ensuring the error is a mere 0.2%, which is well within the jitter normally seen in repeated runs even on the native machine. We were also able to increase the performance of parallel benchmarks by $3.72X$ on average.

REFERENCES

[1] [n. d.]. Pin - A Dynamic Binary Instrumentation Tool. ([n. d.]). http://www.pintool.org

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[4] Doug Burger and Todd M Austin. 1997. The SimpleScalar tool set, version 2.0. *ACM SIGARCH computer architecture news* 25, 3 (1997), 13–25.

[5] T.E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC*.

[6] Jianwei Chen, Murali Annavaram, and Michel Dubois. 2009. SlackSim: a platform for parallel simulations of CMPs on CMPs. *ACM SIGARCH Computer Architecture News* 37, 2 (2009), 20–29.

[7] Geetika Malhotra, Rajshekar Kalayappan, Seep Goel, Pooja Aggarwal, Abhishek Sagar, and Smruti R Sarangi. 2017. ParTejas: A parallel simulator for multicore processors. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 27, 3 (2017), 19.

[8] A-T Nguyen, Pradip Bose, Kattamuri Ekanadham, Ashwini Nanda, and Maged Michael. 1997. Accuracy and speed-up of parallel trace-driven architectural simulation. In *Parallel Processing Symposium, 1997. Proceedings., 11th International.* IEEE, 39–44.

[9] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. 2004. Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 81–92.

[10] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for accurate and efficient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 31. ACM, 318–319.

[11] D. Sanchez and C. Kozyrakis. 2013. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *ISCA*.

[12] Smruti R Sarangi, Rajshekar Kalayappan, Prathmesh Kallurkar, Seep Goel, and Eldhose Peter. 2015. Tejas: A java based versatile micro-architectural simulator. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2015 25th International Workshop on.* IEEE, 47–54.

Received February 2007; revised March 2009; accepted June 2009