A Segment-Aware Multi-Core Scheduler for SystemC PDES

Guantao Liu, Tim Schmidt and Rainer Dömer
Center for Embedded and Cyber-physical Systems
University of California, Irvine
{guantaol, schmidtt, doemer}@uci.edu

***Abstract*—The SystemC IEEE standard is widely used for system design. While the sequential reference simulator is based on Discrete Event Simulation (DES), Parallel DES (PDES) approaches have been proposed for multi-core platforms. This paper proposes a dynamic load-profiling and *segment-aware* scheduling algorithm with optimized thread dispatching to maximize parallel SystemC simulation speed, which generally can be applied to all work-sharing PDES approaches. Based on a compile-time generated Segment Graph (SG), our scheduler can accurately predict the run time of the thread segments ahead and thus make better dispatching decisions. In the systematic evaluation, our segment-aware scheduler consistently shows a significant performance gain on top of the order-of-magnitude speedup of PDES, when compared with the previous scheduling policies.**

I. Introduction

Discrete Event Simulation (DES) has been in use for decades to validate the functionality of Electronic System Level (ESL) designs. In order to improve the performance of DES, Parallel Discrete Event Simulation (PDES) [1] was proposed to run threads in parallel. With the popularity of multi-core hosts, parallel computing platforms are readily available and provide great potential to achieve better performance.

Currently, the SystemC [2] System Level Description Language (SLDL) is used for system design as an IEEE standard. However, the reference simulation library of SystemC still relies on DES, running a single thread at any time, and cannot utilize the computing capabilities of parallel platforms. In recent years, a lot of parallel SystemC simulation approaches [3], [4], [5], [6] have been proposed, which speed up simulation significantly due to parallel execution. However, very few of these approaches [7] address the load balancing problem in their parallel schedulers. In this paper, we propose a segment-aware multi-core thread dispatch algorithm, which can be applied to all work-sharing PDES (SystemC, SpecC, etc.) schedulers. By parsing a design model to a graph of thread segments (a portion of source-code statements between two scheduling points) using static compiler analysis and profiling segment execution at runtime, our approach automatically optimizes the thread dispatch order and consistently achieves a significant speedup over previous thread dispatchers.

The key contributions of this paper are the following:

1) We identify Longest Job First (LJF) as a better than default thread dispatch policy, when thread run time prediction is available.
2) We propose a novel technique to accurately predict thread run times based on a static Segment Graph (SG) and the specific segments threads will execute.
3) We evaluate our proposed segment-aware approach in comprehensive experiments and show that it consistently improves performance for both synthetic and real-world examples.

The rest of the paper is organized as follows: Section II reviews background on parallel SystemC simulation and related work on load-balancing optimizations in PDES. In Section III, we introduce our parallel SystemC implementation, then discuss multi-core scheduling, and propose our optimization algorithm in Section IV. In Section V, we evaluate our segment-aware algorithm with both synthetic and real-world examples. Section VI concludes the paper.

II. Background and Related Work

Parallel SystemC simulation has been a hot research topic in the past few years. In general, these parallel approaches differ in the simulation strategies they apply, the abstraction levels they target, and the host architectures they use. [3] proposes a conservative synchronous parallel simulation approach in which a master thread performs the update and notification phases and a pool of worker threads execute parallel SystemC processes. [4] compares different parallel SystemC approaches, e.g. synchronous PDES, asynchronous PDES, and cycle based simulation, at the Register Transfer Level (RTL). Both [3] and [4] target shared-memory multi-core host architectures. In order to further boost the simulation speed, [5] partitions mixed-abstraction RTL and Transaction Level Models (TLM) into processes suitable for GPU and CPU execution. In [6], the author proposes an approach that explicitly targets loosely timed systems, and runs parallel processes at different simulation cycles. In comparison, [8] proposes a conservative asynchronous PDES approach that also perserves cycle accuracy. The proposed approaches in [6] and [8] run on multi-core hosts.

In this paper, we implement a synchronous PDES approach similar to [3], and propose a segment-aware thread dispatcher inside the PDES scheduler. Our proposed dispatcher is orthogonal to the above approaches, and can be applied to any work-sharing PDES schedulers for shared-memory multi-core machines.

Compared with the many parallel SystemC simulation approaches, load-balancing optimizations on thread dispatching in the context of PDES have gained little attention. [9] presents a dynamic load migration algorithm for reducing the total number of rollbacks in an optimistic PDES environment. [7] proposes a novel parallel SystemC simulation approach with hierarchical multithreading, and optimizes load balancing by

using workload stealing. In [10], authors improve dynamic load balancing of PDES by using the proposed Random, Communication-based and Load-based (RCL) load migration policies. [11] evaluates different process partition strategies (user defined, hash-based, round-robin, etc.) to improve load balancing of parallel simulation. However, in the previous work, the model is divided into distributed logical OS processes and they are allocated to different processors. To avoid workload imbalance on different processors, all the previous work focuses on algorithms to transfer workload among different processes, which is different from ours for work-sharing simulators. To the best of our knowledge, this paper proposes the first scheduler for work-sharing SystemC PDES with thread dispatch order optimized based on the static analysis of the model at hand.

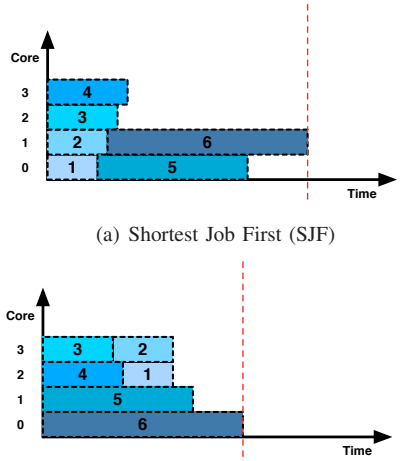
III. Parallel SystemC Simulation

The SystemC reference simulator is based on DES. In the traditional DES scheduler, a single thread is running at all times. When all runnable threads in the *READY* queue finish their current delta cycle, the root thread resumes and performs the update and notification phases. Then the simulation proceeds to the next delta cycle. If no more threads are runnable after the update and notification, the current time cycle finishes. The simulator advances the time and processes the earliest timed event from the *WAIT-FOR-TIME* queue. When the *READY* and *WAIT-FOR-TIME* queues are both empty, the simulation ends.

In contrast to DES, a PDES scheduler dispatches multiple runnable threads concurrently onto multiple available processor cores. In the evaluation phase of our parallel implementation, one more thread (`SC_METHOD`, `SC_THREAD` or `SC_CTHREAD`) is dispatched from the *READY* queue and starts its execution, as long as an idle processor core is available. When all threads finish their evaluation phase, the last running thread performs the update and notification phases and advances the time if the *READY* queue is still empty.

In order to avoid race conditions among accesses to internal shared states, we protect the SystemC kernel with a mutex lock. Whenever a thread needs to modify the state of the kernel, it first acquires the mutex before entering the critical section. Similar to [3] and the synchronous approach in [4], our parallel SystemC implementation only executes threads from the same delta cycle in parallel and has a central *READY* queue. For safe communication, our approach also automatically protects every channel with a local lock and fully supports the standard SystemC semantics, including immediate notification. Differing from [3], our parallel kernel is symmetric and every thread can perform the scheduling functions. Thus, our implementation does not need a special root thread to perform the update and notification phases, which eliminates the frequent context switches from worker threads to the root thread.

Also, a semantics-compliant SystemC implementation is "obliged to analyze any dependencies between processes and constrain their execution to match the co-routine semantics" [2]. Here, we rely on a dedicated compiler to identify potential data hazards inside design models in order to avoid any race conditions on shared variables. Our SystemC compiler performs three major tasks, namely conflict analysis, segment-graph construction, and source-code instrumentation. For this paper, we extend and exploit the segment-graph analysis and the corresponding code instrumentation to accurately predict the next thread run times, so that the dispatcher in the scheduler can quickly make better decisions. We present more details of our SystemC compiler in Section IV-C.



(a) Shortest Job First (SJF)

(b) Longest Job First (LJF)

Fig. 1. Two multi-core thread dispatch policies.

IV. Multi-core Scheduling

In each delta cycle of PDES, a number of threads are available in the *READY* queue, as determined by the PDES scheduler. However, these threads typically have a diverse run time in the evaluation phase. We note that the order of dispatching these threads on a multi-core host has a significant influence on the total execution time. As an illustrative example, Fig. 1 compares the execution time of two classic dispatch policies, namely Shortest Job First (SJF) and Longest Job First (LJF), on a four-core machine. In the order determined by their (predicted) run time, threads are assigned to the available CPU cores. When employing LJF (Fig. 1(b)), the current evaluation phase finishes much earlier than when using SJF (Fig. 1(a)).

In general, multi-core scheduling is a classic load balancing problem in algorithm design, which decides the thread dispatch order and is orthogonal to the parallel DES approach. Following this, we propose a *segment-aware* LJF-based thread dispatch optimization to improve the performance of parallel SystemC simulation for the case that the number of parallel threads in the model is greater than the number of cores on the host. Note that our key contribution here is not LJF itself, but the accurate prediction of thread run times that LJF depends on.

A. Classic LJF Thread Dispatch Policy

Without loss of generality, we will assume Linux as the underlying OS on which the parallel SystemC simulator is

based. For Linux, the default scheduling policy since version 2.6.23 is Completely Fair Scheduling (CFS). Even though CFS maintains fairness of execution time among threads, it always picks first the thread which previously took the least processor time. Thus, it favors interactive processes over batch processes (e.g. PDES simulation). CFS is similar to the SJF policy, except in Linux each core has a separate *READY* queue. If Linux detects an unbalanced load on different cores, thread migration is used for balancing.

The general multi-core scheduling problem is proven to be NP-complete in the literature [12], [13]. Thus, in order to efficiently generate an optimized thread dispatch policy for parallel SystemC simulation, the well-established LJF policy should perform better than the default Linux dispatch policy, as illustrated in Fig. 1. In each evaluation phase of the parallel simulation, our kernel measures the run time of each thread by reading the CPU cycle count registers, which has minimal overhead, and uses this profiling information as the run time prediction for the next evaluation phase[1]. Then, the dispatcher sorts threads in the *READY* queue in decreasing order based on their previous execution time. Thus, the threads estimated to run the longest will run first on the available cores. Shorter threads are dispatched then whenever a core becomes available. With m denoting the number of available processor cores, this greedy scheduling algorithm has been proven to introduce a slowdown of less than $(\frac{4}{3} - \frac{1}{3m})$ compared with the optimal multi-core scheduling [14].

The classic LJF thread dispatch policy works well when threads show identical run time every time they are issued. However, this typically is *not* the case in SystemC simulation. A SystemC thread performs an overall job, but such job usually consists of very different tasks. Thus, the actual run time of a thread in the simulation depends on its specific next task ahead (e.g. reading input data, processing frames, or sending output data). Taking this observation into account, we distinguish between the *segments* of code that a given thread executes, and propose our segment-aware dispatch algorithm.

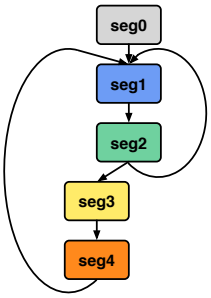
B. Segment Graph (SG)

In PDES, threads switch back and forth between the states of RUNNING (threads in the *READY* and *RUN* queues) and WAITING (threads in the *WAIT* and *WAIT-FOR-TIME* queues). A series of source code statements executed by a thread between two scheduling points can be defined as a thread *segment* [8]. Then, for a SystemC model, it can be converted to a corresponding *Segment Graph* (SG). The SG is a directed graph that represents the code segments executed during the simulation. The nodes in the SG are code segments and the edges indicate the transitions from one segment to another. The code segments always start from a SystemC

[1]Even though the input data to the operations in a thread may vary in different runs, the thread run time likely stays similar. Also, the LJF dispatcher only needs to know the relative order of any two threads' workload, e.g. thread 1 runs longer than thread 2, rather than the absolute values. Therefore, we estimate the next execution time of a thread to be the same as the previous one. Note that it is possible to apply other methods to predict thread run times, but this is not the focus of this paper.

```
          1 SC_MODULE(M) {
          2   sc_port<i_blocking_receiver> InPort1;
          3   sc_port<i_blocking_receiver> InPort2;
          4   sc_port<i_blocking_sender>   OutPort;
          5
          6   void main(void) {
     s0   7     int i, x, y;
  s0s2s4  8     while (true) {
s0s1s2s4  9       InPort1->read(x);
     s1  10       for (i=0; i<LOOP1; i++)
     s1  11         x += 1;
     s2  12       wait(1, SC_MS);
     s2  13       if (x%2 != 0) {
   s2s3  14         InPort2->read(y);
     s3  15         for (i=0; i<LOOP2; i++)
     s3  16           x *= y;
     s4  17         OutPort->write(x);
         18       }
         19     }
         20   }
         21
         22   SC_CTOR(M) {
         23     SC_THREAD(main);
         24   }
         25 };
```

(a) Source code of an `SC_THREAD`



(b) Segment graph

Fig. 2. SystemC thread and Segment Graph (SG).

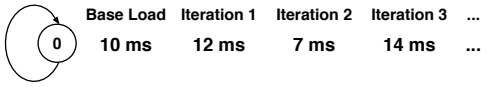
scheduling primitive, e.g. `wait`, `SC_METHOD`, `SC_THREAD`, `SC_CTHREAD`, etc.. Thus, our compiler can instrument these scheduling primitives with segment IDs in the source code. Then, when a thread resumes execution from the WAITING state, the dispatcher can identify the current segment (before actually running it).

Fig. 2 shows a SystemC thread with its segments and the corresponding SG. As shown, every segment starts with a scheduling primitive (including thread creation and context switches, e.g. the `SC_THREAD` in line 23 or the `wait` statement in line 12) and ends before another. The *read* and *write* functions in this example invoke a `wait` statement inside the function calls, so they are all blocking and start new segments (segment 1 and 3 in the *read* function and segment 4 in the *write* function). Also, as indicated in Fig. 2(a), one source code statement (e.g. the `while` statement in line 8) may belong to several segments, depending on the execution paths.

In general, every thread is composed of one or multiple segments. The adjacent segments perform different functions (e.g. in Fig. 2(a) reading input data in segment 0 and processing it in segment 1) and usually run for a different amount of execution time.

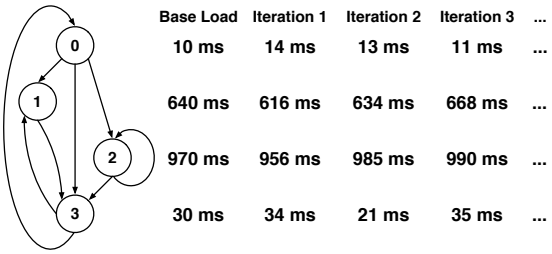
C. Dedicated SystemC Compiler

In order to identify the segment structure of a SystemC model and analyze the interdependencies between threads, we adopted the compiler proposed in [8] for our parallel SystemC simulation framework and built the compiler based on the ROSE infrastructure [15]. Our compiler first parses an input SystemC model into ROSE Abstract Syntax Tree (AST) and constructs a Segment Graph (SG) on top of the AST, following the Algorithm 3 in [8]. Here, we treat the `SC_METHOD`, `SC_THREAD` and `SC_CTHREAD` in the SystemC model as thread creation points (*STMNT_PAR* in Algorithm 3 [8]). Then, according to the SG, the compiler will append the segment ID as an extra argument to the AST nodes of the segment boundary primitives (e.g. `wait`, `SC_METHOD`, `SC_THREAD`, `SC_CTHREAD`, etc.). For those function calls that start new segments (e.g. the *read* function in Fig. 2(a)), our compiler



(a) A single segment in a cyclic thread

(b) Consecutive segments in a cycle



(c) Multiple segments in a general thread

Fig. 3. Different segment structures in a thread.

modifies the function prototype of the called function, adding an integer variable of the segment ID as a new argument. Next, in the function definition, the segment boundary primitives will use the new argument to identify the next segment. Also, based on the SG, the compiler automatically performs static conflict analysis between segments and passes conflict tables to the parallel simulator. In addition, to ensure safe communication, the compiler protects user defined channels (e.g. those derived from `sc_channel` and `sc_prim_channel`) by acquiring a local channel lock at the entry of their public functions and releasing the lock at the exit. The details of static conflict analysis and channel protection are out of the scope of this paper.

D. Segment-Aware Dispatch Algorithm

Fig. 3 depicts three typical segment structures inside a thread. In Fig. 3(a), the thread contains a single segment in a loop[2]. For different iterations, the workload of the segment is similar and typically varies only little due to the input data. Here, the base load in the figure denotes the average amount of execution time of the segment. Fig. 3(b) shows a thread that is composed of three consecutive segments in a loop, e.g. input, processing, and output. The three segments have different workload and their execution time varies significantly. Compared with the cyclic Fig. 3(a) and 3(b), the segment structure in Fig. 3(c) is more general. Threads may have different execution paths in different situations. While the transition between their segments may be unpredictable, we find that a given segment typically carries a similar workload every time it runs.

[2]The loop structure is default for `SC_METHOD` and a common coding idiom for `SC_THREAD` and `SC_CTHREAD` [2].

The classic LJF thread dispatch policy with load prediction based on the previous run only performs well for the simple segment structure in Fig. 3(a). However, for segment structures 3(b) and 3(c), LJF will rely on wrong predictions and thus perform badly. For example, if a thread has the segment structure of Fig. 3(b), LJF will use the run time of segment 0 to predict the execution time of segment 1. Since the classic LJF policy is unaware of the segment structure, it treats the segments the same. However, the workload of the segments is unrelated and varies. Thus, the predicted run time for the next thread segment is inaccurate and LJF performs poorly.

In order to accurately follow the segment structure, we utilize our dedicated SystemC compiler (Section IV-C) to generate the SG of the model and instrument the segment boundary primitives with a segment ID as an extra argument. For example, line 12 in Fig. 2(a) is transformed to `wait(1, SC_MS, 2)`, where `2` is the segment ID. Then the scheduler is aware of the current segment of the runnable threads and can accurately predict their execution time based on the profiling information for the given segment.

Algorithm 1 lists the pseudo code of our segment-aware scheduling algorithm. The calling thread first reads the CPU cycle count register and records the run time of the current segment. Then, the segment ID of the current thread is updated to the next one. Next, while any threads exist in the *READY* queue, our scheduler will dispatch them in order to any available cores, and resume their thread execution. If no core is available but the *READY* queue has remaining threads, the current thread suspends itself. As the sequential scheduler, when the *READY* queue becomes empty, our scheduler per-

Algorithm 1 Segment-Aware Scheduling Algorithm

Input:
Current thread th_{curr}
Next segment $SegID_{next}$
1: $th_{curr}.T_{end} \leftarrow$ CurrentCycles()
2: $RunTime[th_{curr}.SegID] \leftarrow th_{curr}.T_{end} - th_{curr}.T_{start}$
3: $th_{curr}.SegID \leftarrow SegID_{next}$
4: **while true do**
5: **while** READY $\neq \varnothing$ **do**
6: **if** $\exists c \in$ Cores where c is idle **then**
7: $th_{next} \leftarrow$ pop(READY)
8: $th_{next}.T_{start} \leftarrow$ CurrentCycles()
9: dispatch(th_{next}, c)
10: **else**
11: suspend(th_{curr})
12: **end if**
13: **end while**
14: Delta cycle $\delta \leftarrow \delta + 1$
15: process any requested updates in primitive channels
16: process any delta notifications
17: **if** READY $= \varnothing$ **then**
18: advance simulation time
19: process any timed notifications
20: **if** READY $= \varnothing$ **then**
21: terminate the simulation
22: **end if**
23: **end if**
24: sort threads $\forall th \in$ READY in decreasing order of $RunTime[th.SegID]$
25: **end while**

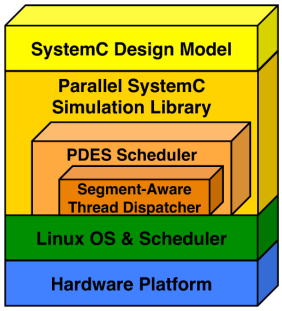


Fig. 4. Software hierarchy of our parallel SystemC simulator implementation.

forms the update and notification to start a new delta or time cycle. Before the beginning of the new cycle, the dispatcher sorts threads in the *READY* queue in descending order of their previous run time in the same segment. For the very first evaluation phase of a segment, the algorithm can use either static compiler analysis, user input, or random values as prediction. Then, in later evaluation phases, the dispatcher predicts the thread execution time using the profiling time of the same segment, instead of the previous run time of the same thread in the classic LJF dispatcher.

Fig. 4 depicts the software hierarchy of our parallel SystemC simulation framework with the segment-aware scheduler and dispatcher. Note that the thread dispatcher is implemented inside the PDES scheduler of the SystemC simulation library (user level), and we do not modify the kernel-level OS scheduler. Compared with the case that the regular parallel SystemC simulator dispatches *all* runnable threads and lets the Linux OS scheduler determine the thread execution, our segment-aware dispatcher only dispatches a number of threads equal to the number of available cores, and fixes their core affinity. Thus, our dispatcher is in full control and the Linux OS scheduler will not modify the thread execution order in our parallel simulation.

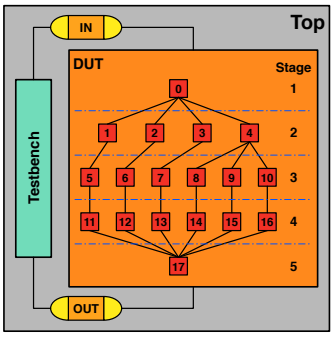
Using the segment-aware prediction, our scheduling algorithm can generate an optimized thread dispatch order[3]. Since the prediction is based on the correct segment, our segment-aware dispatcher will achieve better performance than the classic LJF.

Note that the starvation problem for short threads cannot happen in our segment-aware scheduling algorithm for PDES, since no other threads will be added to the *READY* queue in between a delta cycle, unless an immediate notification occurs. However in the Accellera sequential simulation library and our parallel implementation, even the immediate notified threads will be made runnable only after all current threads in the *READY* queue are dispatched.

V. Experimental Evaluation

We now evaluate our segment-aware optimization on parallel SystemC simulation with synthetic benchmarks and real-world examples to demonstrate the performance gain. Table I lists the hardware specifications of the two multi-core workstations we used in our experiments.

[3]In the case of a single segment per thread, as in Fig. 3(a), the segment-aware approach falls back to the classic LJF.



(a) Task graph block diagram

```
1  SC_MODULE(TASK0) {
2    sc_fifo_in<FLOAT_T> iport0;
3    ...;
4    sc_fifo_out<FLOAT_T> oport0;
5    ...;
6
7    void main(void) {
8      while (true) {
9        FLOAT_T x0 = iport0->read();
10       FLOAT_T y0 = INITIALVAL;
11       ...;
12
13       for (i=0; i<ITER0; i++)
14         y0 = y0 * y0 + y0;
15       ...;
16       oport0->write(y0 + x0);
17       ...;
18     }
19   }
20
21   SC_CTOR(TASK0) {
22     SC_THREAD(main);
23   }
24 };
```

(b) Source code of a task

Fig. 5. Synthetic SystemC benchmark models.

TABLE I
Workstations used for experiments.

Host	8-Core	32-Core
Processor	Intel Xeon E3-1240	Intel Xeon E5-2680
CPU frequency	3.4 GHz	2.7 GHz
Physical CPUs	1	2
Cores/CPU	4	8
Hyperthreads/core	2	2
Total HW threads	8	32

A. Task Graph Benchmarks

Task Graphs For Free (TGFF) [16] is a popular tool to generate standardized random benchmarks for scheduling and allocation research. For our evaluation, we extended TGFF to output actual SystemC models of the generated task graphs. In particular, every task in the task graph is converted to an `SC_MODULE` which initiates an `SC_THREAD` with a specific amount of workload. Here, `SC_THREAD` is used as a test case, but our approach is applicable to `SC_METHOD` and `SC_CTHREAD` as well. Between the tasks, the `SC_MODULE`s use `sc_fifo` channels to communicate and synchronize. Fig. 5 shows the block diagram of a generated SystemC model and the code of a task (a block in Fig. 5(a)). Next, we evaluate the three categories of thread segment structures introduced in Fig. 3 separately.

1) Single-Segment Threads: For the first experiment, we use the extended TGFF to generate SystemC models where each thread has a single segment like Fig. 3(a) with a base amount of workload. Each segment performs data crunching in a `for` loop like lines 13 and 14 in Fig. 5(b), and the workload is determined by the `for` loop iterations. The base workloads of different segments are generated by TGFF as attributes, randomly distributed in a wide range. Then, in each `while` loop iteration the workload of a segment is adjusted by varying the base value with a random factor, to simulate the variation of execution time due to data dependency. To ensure fair comparison between different dispatchers, each thread defines its own reentrant random number generator (*rand_r()*)

TABLE II
PERFORMANCE OF DIFFERENT PARALLEL SYSTEMC SCHEDULERS FOR SINGLE-SEGMENT THREADS (FIG. 3(A)).

Par	Var	8-Core Host				32-Core Host			
		SEQ	PAR	LJF	SEG	SEQ	PAR	LJF	SEG
1 to 2 threads per core	0	*217s*	513%	**+8.0%**	**+8.0%**	*657s*	1669%	**+20.2%**	+20.1%
	20%	*217s*	501%	**+9.2%**	**+9.2%**	*655s*	1607%	**+17.0%**	+16.9%
	40%	*217s*	480%	**+8.8%**	**+8.8%**	*654s*	1507%	**+12.8%**	**+12.8%**
	60%	*217s*	456%	**+6.8%**	**+6.8%**	*653s*	1407%	**+9.3%**	**+9.3%**
	80%	*217s*	433%	**+5.1%**	**+5.1%**	*652s*	1314%	**+6.5%**	**+6.5%**
2 to 3 threads per core	0	*260s*	574%	**+1.6%**	**+1.6%**	*924s*	1999%	**+11.4%**	**+11.4%**
	20%	*260s*	563%	**+4.4%**	**+4.4%**	*923s*	1937%	**+11.8%**	**+11.8%**
	40%	*259s*	545%	**+4.6%**	**+4.6%**	*921s*	1842%	**+7.9%**	**+7.9%**
	60%	*258s*	526%	+2.5%	**+2.7%**	*920s*	1749%	**+4.3%**	**+4.3%**
	80%	*258s*	510%	**+1.0%**	+0.8%	*918s*	1673%	**+1.8%**	**+1.8%**

TABLE III
PERFORMANCE OF DIFFERENT PARALLEL SYSTEMC SCHEDULERS FOR MULTI-SEGMENT THREADS (FIG. 3(B)).

Par	Var	8-Core Host				32-Core Host			
		SEQ	PAR	LJF	SEG	SEQ	PAR	LJF	SEG
1 to 2 threads per core	0	*220s*	515%	-3.3%	**+6.0%**	*772s*	1634%	+8.0%	**+21.6%**
	20%	*219s*	504%	-1.6%	**+7.7%**	*770s*	1587%	+7.4%	**+16.8%**
	40%	*218s*	482%	+0.8%	**+8.5%**	*768s*	1484%	+6.9%	**+12.9%**
	60%	*217s*	459%	+2.4%	**+7.2%**	*767s*	1380%	+6.1%	**+9.7%**
	80%	*216s*	437%	+3.2%	**+5.7%**	*765s*	1290%	+5.0%	**+6.6%**
2 to 3 threads per core	0	*263s*	564%	-2.0%	**+3.2%**	*829s*	1952%	+0.8%	**+13.5%**
	20%	*262s*	555%	-1.3%	**+5.8%**	*828s*	1899%	+0.6%	**+13.8%**
	40%	*261s*	539%	-0.7%	**+5.8%**	*826s*	1818%	+0.2%	**+9.7%**
	60%	*261s*	522%	-0.4%	**+3.6%**	*824s*	1736%	+0.6%	**+5.4%**
	80%	*260s*	507%	-0.8%	**+1.8%**	*822s*	1657%	+1.0%	**+3.2%**

to generate the same sequence of random numbers in different simulation runs.

In our experiments, we set the maximum variation of the workload in different iterations to be 0, 20%, 40%, 60%, and 80%. A variation of 0 means the workload of the same segment in different iterations stays the same, a maximum variation of 20% means that the workload in any iteration is within the range of 80% to 120% of the base of the segment, and so on and so forth. In addition, we generate two sets of task graphs that have a different number of parallel threads at each stage (Fig. 5(a)), except the first and last stages. The average number of parallel threads per core is chosen in the range of 1 to 2, or 2 to 3. Each set of task graphs contains 30 different benchmarks, and runs on the two workstations.

Table II shows the average performance gain of different parallel schedulers over the 30 benchmarks compared with the sequential SystemC simulator from Accellera. The first column *Par* in the table refers to the average number of parallel threads per core at each stage and the second column *Var* refers to the maximum variation of the workload of the same segment in different iterations. For different SystemC schedulers, *SEQ* refers to the sequential SystemC from Accellera, *PAR* refers to our parallel implementation with the Linux scheduling, *LJF* refers to the parallel version with classic LJF dispatching, and *SEG* refers to our segment-aware optmization. The simulation times of *LJF* and *SEG* already include the additional overhead of profiling and sorting. Their relative speedup is compared with *PAR*.

Table II allows the following observations:

1) **Parallel simulation is fast**: Since the benchmarks have plenty of parallelism inside the models, all parallel simulators achieve a good performance gain on the multi-core hosts, up to 5x on the 8-core, and 20x on the 32-core machine. Also, a larger number of parallel threads leads to higher speedup.
2) ***LJF* and *SEG* are faster than *PAR***: When each thread contains a single segment, the *SEG* scheduler with segment-aware optimization shows the same performance as the classic LJF algorithm. But compared with the parallel simulation that relies on the Linux dispatcher, our segment-aware optimization is clearly better. Also, the segment-aware scheduler achieves greater speedup on the 32-core host than the 8-core host for the same type of benchmarks, as a larger number of processing cores leads to greater variability in thread dispatching.
3) **Prediction needs to be accurate**: In Table II, it is clear that the smaller the variation of the workload in different iterations, the higher the performance gain is. In the case that the maximum variation of the workload is 80% (which is rare in real world), all the parallel schedulers have similar performance, as the prediction is inaccurate in *LJF* and *SEG*. However, when the maximum variation of the workload is 40%, the *LJF* and *SEG* schedulers still achieve an additional speedup of 8% on the 8-core and 13% on the 32-core host, in the case that the average number of parallel threads per core is in the range of 1 to 2.

2) Multi-Segment Threads: Next, we evaluate our parallel schedulers with benchmarks where each thread has three consecutive segments, like Fig. 3(b). Now in Fig. 5(a), each block in *DUT* still represents an `SC_THREAD`, and it contains three segments that are separated by `wait` statements. The workload in adjacent segments is unrelated and varies independently. The experimental results are shown in Table III.

In addition to the first three observations from Table II, we make another observation in Table III:

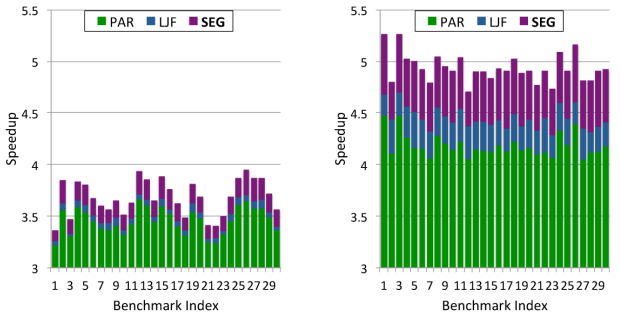
4) **Segment-awareness matters**: In contrast to Table II, the performance gain of *LJF* degrades because the prediction of segment run time is inaccurate. On the other hand, the segment-aware scheduler achieves a significant speedup over the other two parallel schedulers. For example, on the 32-core host, when the number of parallel threads per core is within the range of 1 to 2 and the workload variation is 0, the three parallel schedulers speed up by 16x to 19x. Compared with the parallel scheduler with Linux dispatching, the segment-aware optimization achieves another 20% speedup (while the relative improvement of LJF is less than 10%).

3) General Threads: Finally we evaluate our parallel schedulers with benchmarks in which each thread has multiple segments in a general structure, as Fig. 3(c). Table IV shows the experimental results and allows another observation:

5) **Segment-aware scheduler identifies the correct segments**: In Table IV, even though a thread may take different execution paths, our segment-aware scheduler still identifies the next segment correctly and achieves a high speedup over the other two parallel schedulers

TABLE IV
PERFORMANCE OF DIFFERENT PARALLEL SYSTEMC SCHEDULERS FOR GENERAL THREADS (FIG. 3(C)).

Par	Var	8-Core Host				32-Core Host			
		SEQ	PAR	LJF	SEG	SEQ	PAR	LJF	SEG
1 to 2 threads per core	0	*183s*	370%	+2.4%	**+6.2%**	*402s*	844%	**+4.1%**	+3.4%
	20%	*183s*	362%	+2.2%	**+6.9%**	*402s*	807%	+2.5%	**+2.9%**
	40%	*183s*	345%	+1.7%	**+6.7%**	*401s*	750%	+0.4%	**+3.6%**
	60%	*183s*	326%	+1.5%	**+6.1%**	*401s*	692%	+0.7%	**+2.3%**
	80%	*183s*	308%	+1.3%	**+5.5%**	*401s*	643%	-1.4%	**+3.1%**
2 to 3 threads per core	0	*218s*	439%	+6.8%	**+21.0%**	*572s*	1201%	+19.2%	**+37.0%**
	20%	*218s*	433%	+6.5%	**+19.9%**	*572s*	1183%	+20.9%	**+35.2%**
	40%	*218s*	418%	+6.5%	**+18.2%**	*571s*	1121%	+18.6%	**+31.7%**
	60%	*218s*	401%	+6.0%	**+15.7%**	*571s*	1056%	+17.0%	**+28.8%**
	80%	*217s*	384%	+5.2%	**+13.5%**	*570s*	1008%	+15.2%	**+23.7%**



(a) 1 to 2 threads per core (b) 2 to 3 threads per core

Fig. 6. Performance comparison for general threads (Fig. 3(c)) on a 8-core host.

(more than 35% over *PAR* on the 32-core host, when the number of parallel threads per core is 2 to 3 and the workload variation is 0).

However, the performance of the three parallel schedulers is similar on both hosts when the parallelism is low (1 to 2 threads per core). This is due to the fact that each thread takes different execution paths and has a different number of segments in total. Thus, at a certain point in the simulation, most threads finish all their segments in the current iteration but some threads have extra segments to execute in the following delta cycles. That reduces the parallelism in the simulation (i.e. the number of parallel threads is smaller than the number of cores), in which case the classic LJF dispatcher and our segment-aware optimization perform the same as the Linux dispatcher. The performance of the segment-aware scheduler improves a lot when the parallelism increases to 2 to 3 threads per core.

Fig. 6 shows the performance of different parallel schedulers for each individual benchmark on the 8-core host. Again, each thread contains multiple segments in a general structure and the maximum variation of the workload is 40%. The number of parallel threads per core for Fig. 6(a) is within 1 to 2, and Fig. 6(b) has 2 to 3 parallel threads per core. Here, we make another observation:

6) **Our segment-aware scheduler consistently shows the best performance**: For all 60 benchmarks, even though they have different segment graphs, our segment-aware scheduler always achieves the highest speedup, significantly better than the other two parallel schedulers.

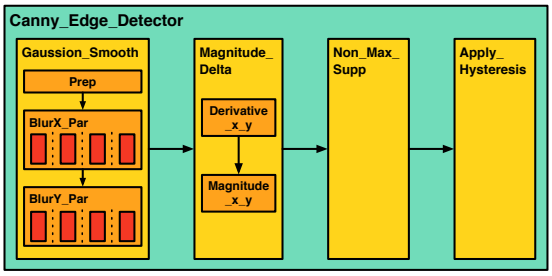


Fig. 7. Pipelined Canny edge detector example.

TABLE V
PERFORMANCE COMPARISON FOR CANNY EDGE DETECTOR ON A 8-CORE HOST.

Benchmark	SEQ	PAR	LJF		SEG		
	Time	Speedup	Speedup	+Speedup	Speedup	+Speedup	Overhead
canny_v1	*138.8s*	**298.1%**	290.4%	-2.6%	284.6%	-4.5%	0.05%
canny_v2	*139.0s*	305.7%	296.4%	-3.1%	**360.4%**	+17.9%	0.15%
canny_v3	*139.0s*	255.9%	261.8%	+2.3%	**329.9%**	+28.9%	1.75%

B. Canny Edge Detector Example

For a first real-world experiment, we use a pipelined Canny edge detector to demonstrate the performance gain of our segment-aware optimization. The Canny edge detector is a popular image processing application to detect a wide range of edges in images. Fig. 7 shows the block diagram of the Canny edge detector in SystemC. In this model, seven functions (i.e. *Prep*, *BlurX_Par*, *BlurY_Par*, *Derivative_x_y*, *Magnitude_x_y*, *Non_Max_Supp* and *Apply_Hysteresis*) are applied to a sequence of input images in a pipelined fashion. In *BlurX_Par* and *BlurY_Par*, multiple parallel threads work on different slices of the image. The number of parallel threads in these two modules is configurable as an exponent of 2. In Fig. 7, all blocks are implemented as `SC_MODULE` and communicate through `sc_fifo` channels. The parallel modules in the pipeline may be blocked by the `sc_fifo` channels, and have multiple segments in one thread. Thus, their segment structure is similar to that in Fig. 3(c). The execution of some segments is optional, depending on whether the buffers in the `sc_fifo` channels are empty or not. As a result, the LJF algorithm degrades due to inaccurate predictions, whereas our segment-aware optimization achieves a much better performance.

Table V shows the performance of different parallel SystemC schedulers for the pipelined Canny edge detector example on the 8-core host. Here, the relative speedup of *LJF* and *SEG* is compared with *PAR*, and *canny_v1*, *canny_v2* and *canny_v3* have 1, 8 and 256 worker threads in *BlurX_Par* and *BlurY_Par* respectively[4]. Clearly, the LJF scheduler has similar or worse performance than the default Linux scheduler, but our segment-aware algorithm achieves an additional speedup of up to 28%. Only when the number of parallel threads in the model is lower than the number of cores on the host (e.g. *canny_v1* has up to 7 parallel threads, made up of the seven pipelined stages in the model), *LJF* and *SEG* are slightly worse due to the small profiling and scheduling overhead.

[4]As these three benchmarks process the same sequence of images, a larger number of parallel threads means a smaller amount of workload in each worker thread.

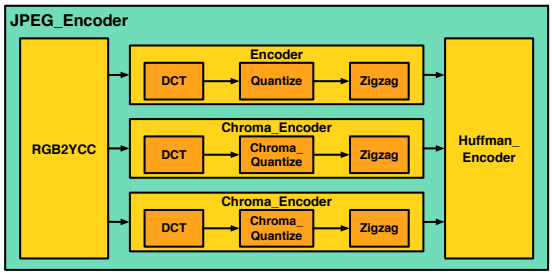


Fig. 8. Pipelined JPEG encoder example.

TABLE VI
PERFORMANCE COMPARISON FOR JPEG ENCODER ON A 8-CORE HOST.

Benchmark	SEQ	PAR	LJF		SEG		
	Time	Speedup	Speedup	+Speedup	Speedup	+Speedup	Overhead
JPEG	*176.1s*	331.4%	324.7%	-2.0%	**400.6%**	+20.9%	0.87%

C. JPEG Encoder Example

Our second real-world experiment uses a JPEG image encoder (an extended version of [17]). Its block diagram is shown in Fig. 8. Here, the *RGB2YCC* module first performs color-space transformation on an image from RGB to YCbCr. Then, the image is split into blocks of 8×8 pixels and each color component (Y, Cb or Cr) of a block undergoes Discrete Cosine Transform (DCT), quantization, and zigzag ordering separately. At the end, the resulting data for all 8×8 blocks is further compressed with the lossless Huffman encoding algorithm. Since encoding of the three color components (Y, Cb and Cr) is independent, the JPEG model executes these three encoders in parallel. Also, to efficiently process a stream of images, our JPEG encoder example is implemented in a pipelined fashion. Same as the Canny edge detector, each block in Fig. 8 is implemented as an `SC_MODULE` and using `sc_fifo` for communication. Thus, each thread has multiple segments and owns a segment structure like Fig. 3(c).

Table VI compares the performance of different SystemC schedulers for the JPEG encoder example on the 8-core host. Again, our segment-aware optimization shows the best performance, achieving an additional 20% speedup over *PAR*. In comparison, the LJF scheduler is slightly worse than the Linux scheduler (*PAR*) due to the inaccurate prediction based on previous run times.

Table V and VI also show the profiling and sorting overhead of our segment-aware optimization. Clearly, the overhead of our proposed algorithm takes less than 2% of the whole simulation time, and sometimes much less (e.g. 0.05% and 0.15% in the cases of *canny_v1* and *canny_v2*). In order to keep the per-segment execution time information, the extra storage overhead is two `unsigned long long` values (start time stamp and previous run time) per segment and one `unsigned int` value (current segment ID) per thread.

VI. CONCLUSION

In this paper, we propose a segment-aware scheduling algorithm with optimized thread dispatching in the context of a parallel SystemC simulator. By taking the execution time for a specific segment as a prediction of the next run time, our approach dispatches threads in an optimized and efficient fashion. Evaluated with synthetic benchmarks and real-world examples, the implemented parallel simulator shows a speedup of up to 20x over the sequential simulator. More importantly, our segment-aware optimization works on top of this and consistently achieves a high speedup over previous thread dispatch algorithms for examples with complex segment structures. Based on these experimental results, we conclude that accurate prediction of the next execution time based on segment information is critical. Our segment-aware approach achieves significantly better performance than previous schedulers.

In future, we plan to evaluate our approach with more real-world examples and also utilize the compiler infrastructure to generate static load estimates. We also intend to improve the segment-aware scheduler for both sporadic and periodic models, and perform a detailed scalability analysis on many-core platforms.

VII. ACKNOWLEDGMENT

This work has been supported in part by funding from Intel Corporation. The authors thank Intel for the valuable support.

REFERENCES

[1] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, October 1990.
[2] *IEEE Standard 1666-2011 for Standard SystemC® Language Reference Manual*, IEEE Computer Society, January 2012.
[3] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parsc: Synchronous parallel SystemC simulation on multi-core host architectures," in *CODES+ISSS*, Scottsdale, AZ, October 2010.
[4] B. Haetzer and M. Radetzki, "A comparison of parallel SystemC simulation approaches at RTL," in *FDL*, Munich, October 2014.
[5] R. Sinha, A. Prakash, and H. D. Patel, "Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs," in *ASP-DAC*, Sydney, NSW, January 2012.
[6] M. Moy, "Parallel programming with SystemC for loosely timed models: A non-intrusive approach," in *DATE*, Grenoble, France, March 2013.
[7] M. Chung, J. Kim, and S. Ryu, "SimParallel: A high performance parallel SystemC simulator using hierarchical multi-threading," in *ISCAS*, Melbourne, VIC, June 2014.
[8] W. Chen, X. Han, C. Chang, G. Liu, and R. Dömer, "Out-of-order parallel discrete event simulation for transaction level models," *IEEE TCAD*, vol. 33, no. 12, pp. 1859–1872, December 2014.
[9] F. Sarkar and S. K. Das, "Design and implementation of dynamic load balancing algorithms for rollback reduction in optimistic PDES," in *MASCOTS '97*, Haifa, Israel, January 1997.
[10] L. F. Wilson and W. Shen, "Experiments in load migration and dynamic load balancing in SPEEDES," in *Simulation Conference Proceedings*, Washington, DC, December 1998.
[11] A. Inostrosa-Psijas, V. Gil-Costa, R. Solar, and M. Marin, "Load balance strategies for DEVS approximated parallel and distributed discrete-event simulations," in *PDP*, Turku, March 2015.
[12] C. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," in *STOC '88*, 1988.
[13] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA: MIT Press, 1989.
[14] R. Graham, "Bounds on multiprocessing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, March 1969.
[15] D. J. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, May 2000.
[16] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *CODES/CASHE '98*, Seattle, WA, March 1998.
[17] "A JPEG encoder model in SystemC," https://github.com/weiweichen/systemc-jpeg.