Multithreaded Simulation to Increase Performance Modeling Throughput on Large Compute Grids

Carl Beckmann[1], Omer Khan[2], Sailashri Parthasarathy[1], Alexey Klimkin[3], Mohit Gambhir[1], Brian Slechta[1], Krishna Rangan[1]

[1]Intel Massachusetts [2]Massachusetts Institute of Technology [3]Intel Corp., Hillsboro OR

***Abstract*—Simulation for performance modeling of computer architectures is a compute intensive operation requiring many compute resources and CPU cycles. Typical usage models involve running multiple sequential jobs across multiple machines. Multithreading can be used to speed up individual simulation jobs, but only provides higher throughput than multiprogramming on large workloads if speedup scales linearly or superlinearly with the number of threads. Scalable multithreading performance requires effective parallel decomposition of models to eliminate sequential bottlenecks, and minimizing contention on mutual exclusion locks. Scheduling of parallel work is also crucial, and several scheduling algorithms are evaluated. Application-level dynamic scheduling delivers superior speedup than generic OS scheduling with thread oversubscription, on complex models. Cache affinity scheduling improves performance stability of dynamic scheduling. Lookahead execution improves multithreading speedup by reducing barrier synchronization overhead, without requiring speculation and rollback/recovery. Combining these techniques can yield superlinear multithreaded speedup, which results in higher throughput of large-scale simulation workload on distributed compute grids. Our results on a 1024 cores simple processor model indicates that efficient multithreading can yield up to 4X the throughput as job-level parallelism.**

I. INTRODUCTION

Simulation is a crucial tool in the practice of computer architecture and design, and is used extensively in high-level architecture design early in the design cycle, in detailed RTL design and validation, and in post-silicon design validation and debug. Corporate computing installations have evolved over many decades from large mainframes in the 1960s and 70s, to department-oriented minicomputers of the 1980's and 90's, and today are being supplanted by large rack-mount installations of low-cost "blade" servers providing distributed computational capabilities. At the same time, recent trends in integrated circuit manufacturing are forcing chip vendors to pack more CPU cores onto each die, rather than increase clock speeds or single-CPU instruction rate to improve performance. With an ever-increasing degree of available hardware parallelism in the computing infrastructure, and in particular with an increasing degree of relatively easy-to-use shared-memory parallelism available in modern compute nodes, it is reasonable to ask whether the use of parallel processing software techniques within individual simulation jobs is a useful and necessary tool to make maximum use of evolving computational capabilities of modern blade server grids.

Given an experimental workload consisting of J jobs, and a compute grid containing H host nodes with C processor cores each, and assuming $J > H*C$, and that each job could be run either sequentially on a single core, or in parallel using up to C cores, if we want to maximize core utilization, we can choose to use only job-level parallelism, or combine some job-level parallelism with parallel processing of each job on a server (*thread-level parallelism* in the sequel). Job-level parallelism is easy to exploit since it does not require the simulation software to run multithreaded. As the number of jobs per host increases, the virtual, physical, or cache memory usage also increases proportionally.

If we exploit thread-level parallelism, the *speedup* obtained by running a single job using from 1 to C threads on a single host may also be sub-linear, due to additional instructions required to manage and schedule parallel tasks, contention on locks, load imbalance and waiting time at barriers, etc. It may also exhibit *superlinear* behavior on some region, because the smaller memory and cache footprint of each thread could result in better cache locality and hit rate than the sequential program [25][26]. The

memory usage will be close to flat, with perhaps a small increase as the number of threads is increased, for additional thread stack frames, synchronization data structures, etc. Thread-level parallelism can be an attractive supplement to job-level parallelism, under several circumstances: first, if the *speedup* of running a job with *N* parallel threads exceeds the *throughput increase* of running *N* jobs concurrently; and second, the lower memory usage with thread-level parallelism could be exploited by spending a smaller fraction of the compute grid capital budget on memory and more on additional host nodes or more cores per node (*cf.* the discussion of *speedup* versus *costup* in [24]).

To motivate this work, Tables 1 and 2 illustrate typical simulator performance observed on our detailed core models. In Table 1 we ran either a single copy or eight copies of our detailed performance simulator on an 8-core host machine. Throughput does not scale perfectly with the number of jobs, achieving a throughput of only 6.68 with 8 concurrent jobs, which represents an efficiency of only 83.5%. Table 2 shows measurements of the same detailed performance model as a single multithreaded job on the same 8-core host. Although the simulator runs up to 4.02 times faster with eight threads, improving individual job turnaround times, this represents an efficiency of only 50.3%, meaning that less throughput is achievable on workloads with large numbers of jobs. Improving the multithreaded performance of our simulators is an on-going effort, requiring many man-hours of software engineering to analyze, optimize and restructure the code, due to the large size of the software codebase which has been in development for many years.

Table 1. Complex model multi-programmed performance on 8-core host

jobs	*throughput*	*efficiency*
1	1.00	100.0%
8	6.68	83.5%

Table 2. Complex model multithreaded performance on 8-core host

threads	*speedup*	*efficiency*
1	1.00	100.0%
2	1.64	82.0%
4	2.74	68.5%
6	3.51	58.5%
8	4.02	50.3%

A key question we seek to address in the work presented here is whether multithreading software techniques used within individual simulation jobs can deliver comparable or better performance than simply farming out multiple jobs onto the many available cores on a server grid. In this work, we argue that achieving *superlinear* speedup in multithreaded simulations is not only desirable, but necessary in order to provide higher throughput for typical performance modeling workloads compared to simply using job-level parallelism. We show that a series of improvements is necessary, in the organization and characteristics of the performance models, as well as in the performance modeling infrastructure. These include: optimizing shared data structures in the model code to eliminate or minimize locks and contention; an effective parallel decomposition that exposes sufficient parallelism and minimizes sequential bottlenecks and load imbalance; and exploiting long latencies in the model to allow *lookahead* execution; dynamic scheduling using a work queue managed at the simulator application level; "unfair" LIFO-ordered task queueing; and cache affinity scheduling. Our results indicate that perhaps superlinear multithreaded speedup is achievable (up to 12X on an 8 core host machine). This non-speculative and efficient multithreading results in up to 4X throughput of using job-level parallelism to speed up software simulations of future multi-core processors.

II. Related Work

Researchers in computer architecture have explored the use of parallel simulation from the mid 1980s [5][6][7][8][9][10][16][17]. Early work in software infrastructures was motivated by the need to model large-scale parallel systems [10][14][15], and recently there has been a resurgent interest due to the need to model chip multiprocessors [13][18]. Vachharajani, et al. explored how a performance simulator can be automatically parallelized, if the model is structured appropriately [18]. Fujimoto classified parallel simulation approaches into *conservative* techniques (e.g. [5]) and *optimistic* ones (e.g. Time Warp, and related approaches [8][9][17]), depending on how aggressively communication dependencies are enforced among parallel tasks [7]. The work presented here uses conservative parallel simulation, but attempts to exploit *lookahead* for relaxed synchronization based on the ideas in this early work.

Recently, parallel simulation has been used in the industry (Barr, et al. [4]). Our infrastructure on parallel simulation largely borrows from this previous body of research and particularly from

Barr, et al.'s work on paralleling Asim [2][3]. However, we discovered that we needed two key techniques—lookahead execution and dynamic scheduling—to obtain effective speedups in a general parallel infrastructure that can support multiple products and projects.

Although our implementation of dynamic scheduling is based directly on POSIX threads (*pthreads*), we build upon many of the ideas of Reinders [1], in particular the notion of *unfair scheduling* to promote effective use of caches .
While the *Threading Building Blocks* library must be sufficiently general to support many kinds of parallel applications, our software infrastructure is aimed specifically at detailed microarchitecture performance modeling, and we have been able to keep the details of parallel programming largely transparent to model code developers.

An alternate way to improve simulation speed is to create a performance model in an FPGA rather than software [11][12][13]. Although FPGAs can provide orders of magnitude improvement in simulation speed, they are still hard to program. The research community is actively looking at how to make this task simpler. Until FPGAs are easy enough to program or performance models in FPGAs are easy to change, software simulation will continue to be the key vehicle for simulation in the industry. Our techniques apply to such simulators as well.

The prior work on parallel simulation has focused primarily on techniques to speed up individual simulation runs. It appears that little attention has been given to date to the bigger picture of large-scale industrial workloads. In this paper we argue that it is imperative to take this large workload context into account, because there is an inherent tradeoff between parallel processor resources being employed for running multiple simulation jobs versus running parallel threads of a single job. We focus not only on the speedup of individual runs, but on obtaining higher throughput for the workload as a whole. We argue that *superlinear* speedup is necessary to achieve this goal, and demonstrate the feasibility of this by using a combination of practical parallel software techniques.

III. Decomposition For Multithreaded Models

Accurate industrial-quality simulators have very large source code bases often exceeding millions of lines of code, and incurring a significant memory usage at runtime. The need to model multicores and products with large on-board caches and memories also tends to increase the memory footprint of simulators. In order to deal with the software engineering challenges of constructing such accurate and complex models, a structured approach must be taken wherein the software is organized into *modules*, loosely corresponding to the underlying hardware organization, and allowing the software modules to be developed independently, and different versions of modules to be interchanged in order to explore design variations, and exploit speed/fidelity tradeoffs [2][18]. Communication between modules is constrained to use specialized software constructs called channels or *ports*, and modeling languages such as VHDL, SystemVerilog or SystemC have specialized classes or software interfaces for this [22].

The organization of the model into modules and ports provides an obvious opportunity for *parallel decomposition* [23]: If the simulator software is organized isomorphically to the hardware, this parallelism is exposed in the form of software modules that can be run in parallel [4]. The characteristics of *ports* play an important role too, since port communication creates runtime dependencies that must be preserved for correct parallel execution [5][7].

If the ports connecting modules have more than one cycle of latency, e.g. because they model fixed-delay pipeline structures or long distance propagation delay lines across regions of a chip under design, the synchronization requirements for these modules is even looser, allowing, e.g. the consumer side to "run ahead" of the producer side, as long as it does not run so far ahead that it misses a message sent by the producer. This is called *lookahead* and can be exploited to reduce synchronization requirements and expose more parallelism to the thread scheduler in multithreaded simulations. Note that exploiting lookahead while preserving data dependencies still falls under the umbrella of *conservative parallel simulation* [7][14]; in this work, we do not consider *optimistic* approaches (e.g. [9]).

IV. Scheduling

There are several ways to exploit the degree of parallelism in the software (referred to in the sequel as the number of *parallel tasks*) on the parallelism available in the host hardware (the number of *host cores*). If the hardware parallelism exceeds that of

the software then we can simply create one *worker thread* for each *parallel task*, and allow the host operating system to assign one *worker thread* to each *host core*. We call this *static scheduling*. To use static scheduling, we may have to alter our parallel decomposition to keep the number of tasks from exceeding the number of host cores, e.g. by changing the number of simulator modules assigned to each task. If the software parallelism exceeds that of the hardware, then two different strategies are possible: we could create one worker for each task, and allow the operating system to manage the assignment of worker threads to host cores, referred to as *thread oversubscription* [1]. Alternately, we could create only one worker thread per host core, and manage the assignment of tasks to workers in our simulation software, referred to in the sequel as *dynamic scheduling*. Our dynamic scheduler can be more intelligent than the operating system's generic thread scheduler, by exploiting information available to it about the simulator (e.g. the number, duration, or interdependencies among tasks).

Several variations of dynamic scheduling work queue management are possible. In *first-in-first-out (FIFO) queuing*, tasks are reinserted into the queue in the order they arrive, while with *last-in-first-out (LIFO) queuing*, the last task enqueued will be the first dequeued by the next available worker thread. Although FIFO queuing would appear to be more "fair" to tasks, LIFO queuing has two possible advantages: First it allows the longest-running (or rather, the last-finishing) task in a cycle to be the first to execute in the next cycle. Secondly, it increases the likelihood that a given worker immediately re-executes the same task in the next cycle, thus taking better advantage of temporal locality in the host core's cache. It has been observed elsewhere that such "unfair" scheduling policies can have significant performance advantages [1].

A further refinement aimed at exploiting cache locality is to explicitly keep track of which worker previously executed each task. To fetch the next task, each worker scans the queue starting at the head, and dequeues the first ready task that it previously executed. If no such task is found, it dequeues from the head. This scheme is referred to as *task affinity scheduling*.

The barrier synchronization at the end of each simulated clock cycle can be relaxed to allow *lookahead* as follows. A worker uses any of the algorithms described above to enqueue and dequeue tasks, but instead of strictly waiting at the barrier for all other workers to finish the same simulation cycle, it executes the dequeued task as long as its simulation cycle does not exceed the slowest "straggler" task's last completed cycle by more than the *lookahead time*, i.e. one cycle less than the minimum latency on any port connecting modules in different tasks. If the scheduler has knowledge of the module connectivity, the lookahead time can be further relaxed from a global value to a function of just those modules a task actually receives messages from; or the synchronization can further deferred until the task actually attempts to read data from its ports (*port-based synchronization* [4][11]). In the work presented here we have used only global lookahead enforced by barrier synchronization.

V. Experimental Setup

Our experience running production-quality performance models with thread-level parallelism successfully yielded speedup compared with single sequential runs, but significantly less than job-level throughput on our highly utilized compute grid. Preliminary analysis pointed to several factors limiting the achievable speedup: first, making the large legacy software code base thread-safe required the addition of numerous locks, which incurred significant overhead and thread contention at runtime; and second our initial task decomposition of the model resulted in significant load imbalance among tasks because the model of the on-chip interconnect was more difficult to decompose into parallel tasks than the multiple on-chip core models. We believed both of these factors could be addressed with sufficient software engineering effort to restructure the code to eliminate shared data and locks and expose more parallelism. But before committing to this effort, we wanted to understand whether *superlinear* speedups are achievable, yielding higher *throughput* with multithreading, and in understanding what further changes would be needed to the infrastructure and models to achieve this. To investigate this, we have built several performance models that are simpler in some aspects than our full production model, while preserving other essential details and utilizing the same performance modeling software infrastructure.

Our first simplification, which we will refer to as *complex core, no interconnect (CCNI)*, and shown in Figure 1, consists of a model of a *P*-way multicore chip (which we will refer to as the *target architecture*, to distinguish it from the host architecture that runs our model), where the

performance model of the target core is of similar complexity and fidelity as Intel® Core 2™-like full production model. In particular, we made no attempt to eliminate the many locks, due primarily to the complex instruction set functional model. Unlike the full production model, CCNI contained only a trivial model of the target on-chip interconnection network, which we believed would eliminate the primary source of load imbalance in the production model. The number of target cores *P* is a parameter that we can vary. The simplest parallel decomposition for this model is to assign one target core to each parallel task. If *P* is larger than the number of host cores, we can assign more than one target core to each task if we want to statically schedule the tasks. Moreover, since CCNI does not accurately model the on-chip interconnect and the core modules do not communicate over ports, the allowable *lookahead* between core tasks can be made arbitrarily large for purposes of investigating its effects on speedup. In a more realistic model, lookahead would be limited by the amount of core-to-core or core-to-network latency, but latencies in the range of five to ten target core clock cycles or more are not implausible, depending on the fidelity required of the target network model.

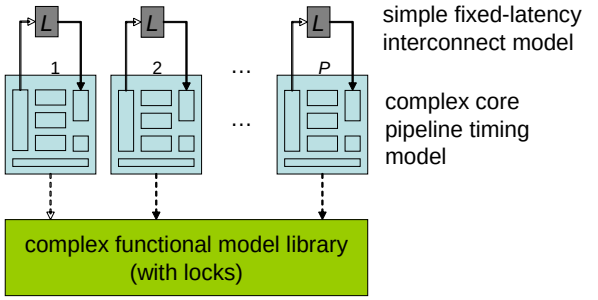

Figure 1. Complex Core, No Interconnect (CCNI) model

For the CCNI model, we use application traces from the SPEC 2000 and 2006 suites generated using the Pintool [27]. We run duplicate copies of the same trace for each target core, providing the worst-case workload as all threads will need the same resources. Unless otherwise noted, we run each trace for 1 million instructions after warming up the cache state. This results in simulating *P* million total instructions on a *P*-way multicore model. Simulation is guaranteed to run in a deterministic and repeatable fashion for both sequential and multithreaded versions of our experiments.

Next, in order to completely eliminate the effects of multithreading locks, we implemented an abstract version of a core model. It consists of several random number generators used to model a very simple processor pipeline, with only three generic instruction types: non-memory, memory-read, and memory-write. Pseudo-random (but deterministically repeatable) random instruction streams are generated by each core model, and the memory read/write instructions exercise an on-chip interconnection network model. The first of these models contains a simple monolithic interconnection network queuing model, referred to as *simple core, monolithic interconnect (SCMI)*, and shown in Figure 2. Although the cores are very simple, it is easy to scale the model to very large *P* to increase the workload and experiment with alternative parallel decompositions and scheduling schemes. The latency *L* between the target cores and the target interconnect is a parameter that can be varied, allowing us to explore lookahead values from zero to *L*-1.

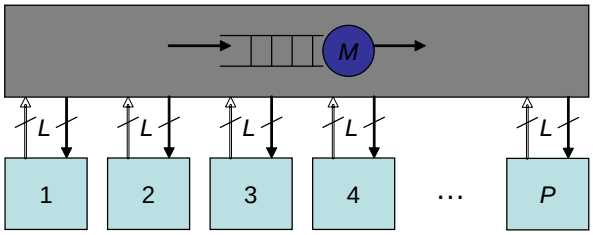

Figure 2. Simple Core, Monolithic Interconnect (SCMI) model

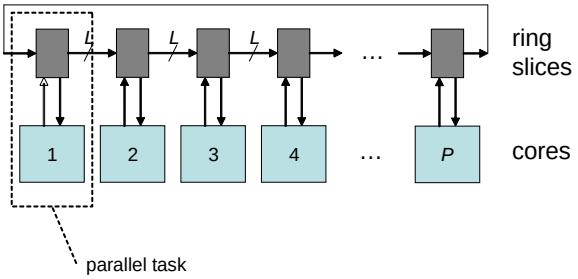

Figure 3. Simple Core, Scalable Interconnect (SCSI) model

Although SCMI does not contain any locks (other than a small number of carefully optimized locks in our simulation kernel), the monolithic interconnect model does present a source of load imbalance, especially with large *P*. Our final model replaces this with a more scalable ring-network model, consisting of separate *ring slice* segments, with one slice per target core, as illustrated in Figure 3. Parallel decompositions of this model place one target core and ring slice into each task, or *M* core/slice pairs for each task (*M*=2..*P*) if required for static scheduling. The latency *L* of ports between ring slices is another model parameter that can be varied in order to allow lookahead. This model is

called *simple core, scalable interconnect (SCSI)*.

Another essential feature of SCMI and SCSI is that due to their extreme simplicity, they use much less memory per target core than either CCNI or our production models. We hypothesized that their small memory and cache footprint would make it easier to achieve superlinear speedups, by allowing the multithreaded program to achieve a better cache hit rate by utilizing multiple host core caches (*superlinear speedup* is a well-known phenomenon in scientific computing, sometimes observed when the sequential version of a program runs out of cache or physical memory page frames with large problem sizes [25][26]). If the model code could be made more efficient in this way, it would also tend to amplify any multithreading overhead and contention in our software infrastructure's scheduler that we wished to study.

All experimental measurements are on host systems using Intel Xeon processors with 8 host cores running at 2.33 GHz, with 16 GB of RAM and 4 MB of cache, running a 64-bit Linux 2.6-based SUSE distribution.

VI. Experimental Results

A. CCNI model basic results

In the first set of experiments, we measured the speedup of the CCNI model to determine whether linear or superlinear speedup could be achieved by eliminating the load imbalance using a model with a simple parallel decomposition (*P* identical target cores). We varied the number of target cores assigned to each parallel task to achieve a number of tasks between 2 and 33, for a model with *P*=32 target cores (in addition to the core tasks, one more "default task" is needed in this simulator to parse command line arguments and start other tasks running, etc.).

Figure 4 plots the speedup of the CCNI model versus the number of tasks. Since the number of host cores is 8, static scheduling is in effect when the number of tasks is 8 or fewer. Two sets of results are shown as the number of tasks exceeds the number of host cores: *thread oversubscription*, and *dynamic scheduling*. As the number of tasks is increased beyond 8, our dynamic scheduler continues to achieve higher speedups, indicating that it benefits from exposing more parallelism in the model software. The speedup with thread oversubscription quickly drops, likely because the thread scheduler is oblivious to what is happening in the model code, and attempts to give each worker thread equal time even if a worker is only spin-waiting on a lock or barrier synchronization. The dynamic scheduler never waits at a barrier as long as there are active tasks in the queue, and worker threads that own locks are never swapped out since the number of workers equals the number of host cores.

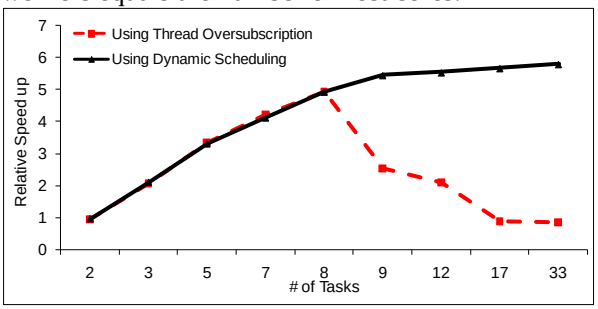

Figure 4. CCNI speedup versus available parallelism (8 host cores)

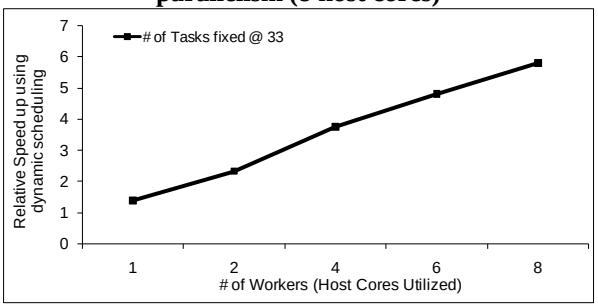

Figure 5. CCNI speedup versus hardware parallelism

Figure 5 shows the performance of the dynamic scheduler as the number of workers (or host cores) is varied from 1 to 8, with a fixed parallel decomposition with 33 parallel tasks (32 core tasks and one default task). With a small number of workers (1 or 2), the speedup is superlinear, but as the number of workers is increased, the parallel *efficiency* (defined as the *speedup* on *N* host cores, divided by *N*) drops off, and speedup is only ~6X for 8 workers. Although plenty of parallelism is exposed to the scheduler (33 tasks for only 8 workers), performance decreases, primarily due to lock contention in the functional model library.

Next we measured the throughput that could be achieved using either job-level parallelism (running 8 sequential jobs concurrently on an 8-core host) versus multithreading (running 8 parallel jobs, one after another, on the same 8-core host). Figure 6 plots the *relative throughput* or *efficiency*, i.e. the absolute throughput or speedup divided by the number of host cores used, for both job-level parallelism and muiltithreading, on our 32-target-core CCNI model. Note that both job-level and thread-level parallelism show decreases in efficiency

(compared to a single sequential run on a single host core) as the number of host cores is increased. Unfortunately, multithreading is not able to outperform job-level parallelism, since the speedup is sublinear, which is consistent with the previous two graphs.

B. SCMI model basic results

To test our hypothesis that mutual exclusion locks in the software limit the speedup of the CCNI model, we developed the SCMI with a much simpler core model that does not include the complex functional model software library. Similar to our full production model, it does contain a nontrivial model of the on-chip interconnect and the interconnect model runs in a single parallel task. To compensate for the much simpler cores compared to CCNI, we increased the number of target cores from 32 to 1024 in our experiments, and we used a core-interconnect latency of 6 cycles to allow for up to 5 cycles of lookahead.

Results presented in Figure 7 show very good speedups when the number of tasks is small (five tasks or less), with superlinear speedups when 3 to 5 worker threads are used. As we increased the number of workers beyond 4, however, the performance quickly saturated at ~6X. We compare various scheduling options when tasks exceed the host cores, by assigning fewer and fewer of the 1024 target cores to each task to create more tasks than host cores. We concluded that the interconnect portion of the model could cause load imbalance as it contained significantly more work than the simple core modules. To compensate for this, we changed the enqueuing policy from FIFO to LIFO (with and without affinity scheduling), hoping that the longest-running and last-finishing tasks in a given cycle would thus become the first task to be executed in following cycle, thereby shortening the critical path and parallel execution time. The LIFO+Affinity curve in Figure 7 shows modest speedup of around 6X. When compared to thread oversubscription (PTHRD in the figure), our LIFO with affinity scheduler shows significant improvement in speedup, especially for number of tasks ≥ 13.

C. SCSI model and scheduling options

The superlinear speedup on small numbers of host cores was encouraging, but we concluded that in order to achieve scalable results, we needed to improve the parallel decomposition of the model. This motivated the SCSI model with a perfectly symmetric decomposition with one (or more) core/ring slice pair(s) per parallel task. The left-hand portion of the graph in Figure 8 (number of tasks ≤ 8 host cores) shows that with static scheduling on up to 8 host cores, scalable superlinear speedup was indeed achievable with this model, with a speedup of greater than 12X observed on 8 host cores.

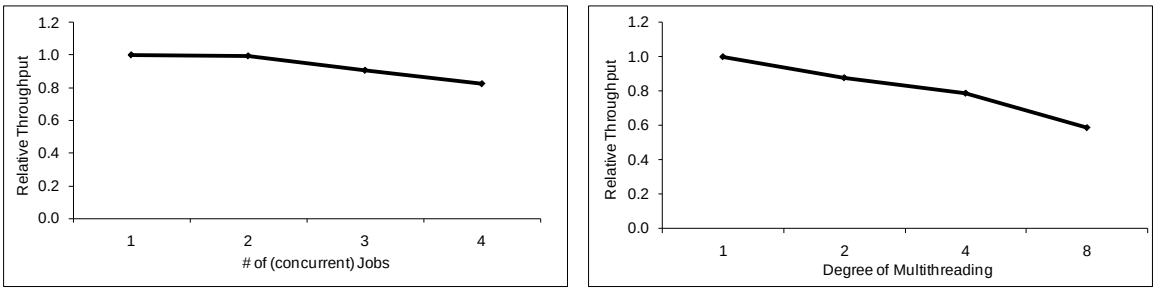

Figure 6. CCNI throughput versus parallelism

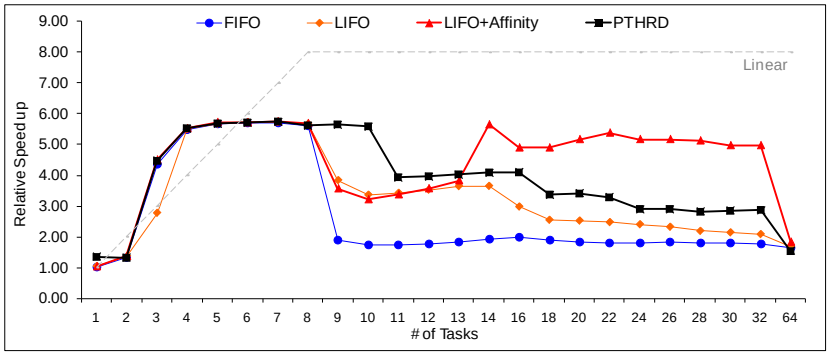

Figure 7. SCMI model speedup versus parallelism

We also investigated the effects of different dynamic scheduling options on this model, by assigning fewer and fewer of the 1024 target cores to each task to create more tasks than host cores. These results are shown on the right-hand side of Figure 8 for number of tasks ≥ 9. Unlike with the CCNI model, thread oversubscription performed much better than FIFO dynamic scheduling, whose speedup plummeted to around 2 with greater than 8 tasks. Oversubscription's speedup also dropped significantly, but still achieved speedups between 6 and 10 as the number of tasks varied from 9 to 16.

One possible explanation is that the additional overhead of the scheduler significantly adds to the execution time of each worker thread, which is more noticeable now that the core model is much simpler than in CCNI. However with P=1024 target cores, and 16 tasks, there are still 64 target cores per task; also it is difficult to imagine that the relatively small amount of scheduling code could pose that much more overhead compared to the operating system code involved in thread scheduling in the oversubscription case. We suspected that something more was at work, namely that the FIFO scheduler was causing an effectively random assignment of tasks to workers (and host cores) thereby destroying any cache temporal locality occurring with static scheduling.

To test this hypothesis, we once again changed the enqueuing policy from FIFO to LIFO, which would at least preserve locality for the last worker arriving at the barrier. In addition, we implemented a task queue with explicit worker affinity, as described in section V, which would attempt to reassign tasks to the same worker thread as long as the task queue was not empty. Although the simple LIFO policy yielded only modest improvements over FIFO, the improvement of the LIFO with affinity scheduler were dramatic for numbers of tasks that are a multiple of eight (16, 24, 32, etc.): in these cases, the number of tasks is evenly divisible by the number of worker threads which not only provides an even load balance of tasks among workers, but as a result may also reduce churn in the dynamic assignment of tasks to workers. With cache affinity promoted by scheduler, the only disadvantage of dynamic scheduling is the overhead of the scheduler, which does appear to cause a modest decrease in the speedup from 12 down to about 10 as the number of tasks is increased to 32 (and the number of target cores per task is decreased from 128 down to 32 cores per task).

D. The effect of lookahead

To further improve the performance, we ran additional experiments with the SCSI model, increasing the amount of lookahead in the scheduler to reduce waiting time at the barrier. The results are shown in Figure 9, using our LIFO with affinity scheduler, and varying the lookahead from zero to five target clock cycles. As expected, increasing the lookahead increases the performance by reducing waiting time at the barrier. With a lookahead value of 5 cycles, the performance is consistently superlinear (greater than the 8 available host cores), and almost compensates for the performance drops when the number of tasks is not a multiple of 8. In addition to reducing barrier wait time, lookahead may have a beneficial effect on cache affinity, because when a worker thread finds that it does not have to wait at the barrier, it also does not need to switch tasks, thus preserving the cache locality of the current task.

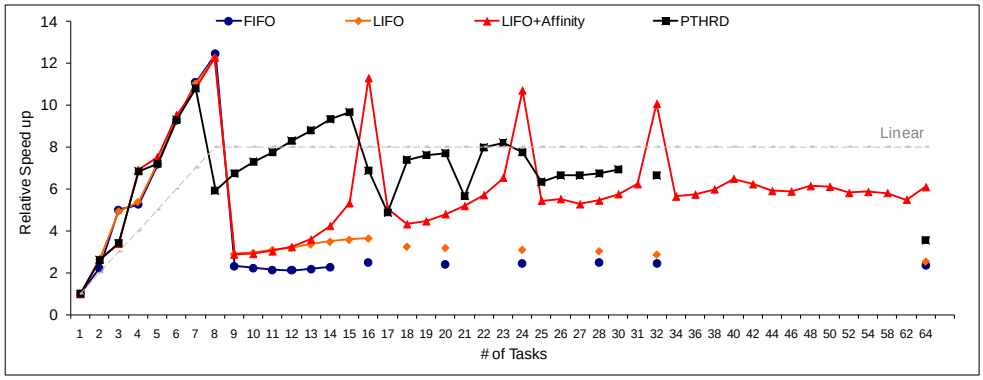

Figure 8. SCSI model speedup

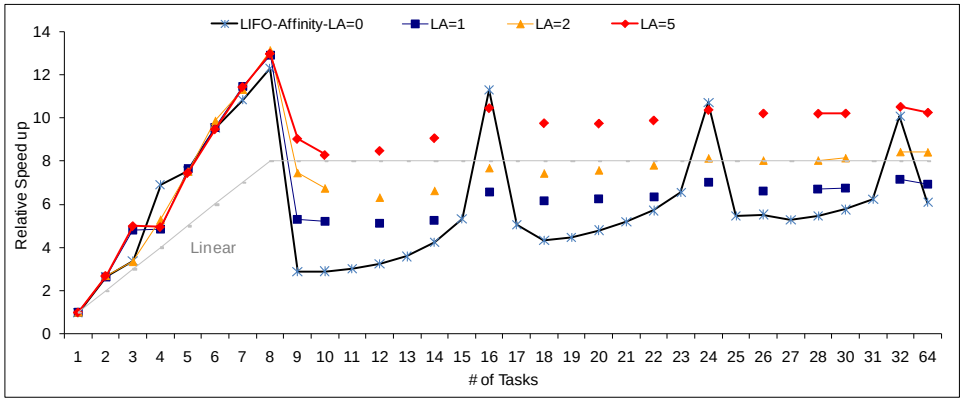

Figure 9. SCSI model performance with lookahead (LA)

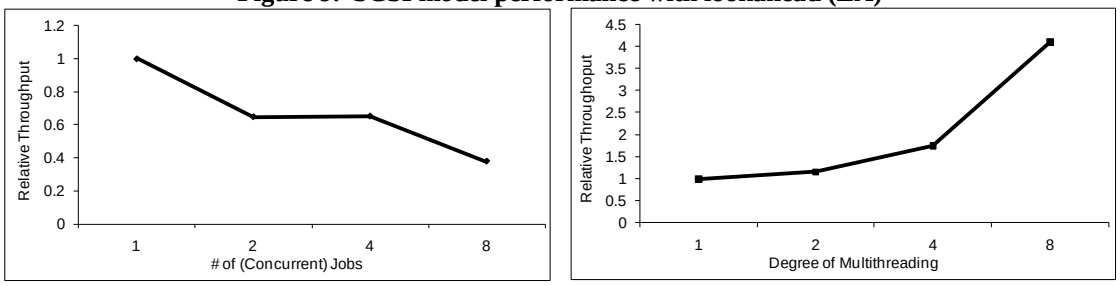

Figure 10. SCSI throughput versus parallelism

E. Job-level versus Multithreading throughput on the scalable model

Finally, to assess the performance of job-level parallelism of the SCSI model, we measured the throughput achieved by running multiple sequential SCSI jobs concurrently on the same multicore host machine. As in the CCNI case, the efficiency of the host drops as the number of concurrent jobs is increased from 1 to 8 jobs, but with the SCSI model, the drop in relative throughput appears to be more dramatic (Figure 10). It is not obvious exactly what is causing this, but perhaps the operating system's thread scheduler is interfering in some way with the cache affinity of jobs, or perhaps the simpler model is more sensitive to memory latency because its baseline cache hit rate is higher in the single-job case and memory contention increases significantly as jobs are added.

Whatever the reason, the marked *sub*-linear throughput increase of job-level parallelism, combined with the significantly *superlinear* speedup achievable with multithreading, make multithreading yield up to four times the *throughput* as job-level parallelism on the SCSI model. Figure 10 shows the relative throughput (efficiency) as the degree of multithreading is increased from 1 (i.e. eight **1**-threaded jobs) to 2 (four concurrent **2**-threaded jobs) to 8 (a single **8**-threaded job). The factor of four is a combination of a factor of almost 2.5 loss in efficiency from job-level parallelism times a factor of more than 1.5 *increase* in efficiency from multithreading (i.e. a speedup of almost 14, divided by 8 host cores).

VII. Conclusions

The results presented in this paper demonstrate that it is possible for multithreaded simulation to deliver scalable performance increases over sequential execution, which in turn provides significantly higher *throughput* on large-scale batch workloads, if certain conditions are met. This makes multithreading a useful tool to improve not only the turnaround time of individual simulation jobs, but also the turnaround time of large multi-job workloads, on highly-utilized multi-server compute grids.

Enabling conditions in the model software include a simulation infrastructure (API, runtime libraries, etc.) that easily exposes the module-level parallelism inherent in the problem domain; software interfaces and coding conventions that promote the development of thread-safe code and minimize the need for mutual exclusion locks and other explicit thread-aware code; the ability of the software infrastructure to take advantage of relaxed synchronization requirements, e.g. by allowing *lookahead* execution where long communication latencies in the model allow it; and the effective

decomposition of the model into sufficiently balanced parallel chunks to run on parallel hardware without sequential l bottlenecks. Moreover, we have found that application-level scheduling of parallel work plays a crucial role in delivering superior performance over generic OS-level thread schedulers, and that in addition to load balancing, must pay attention to cache locality and affinity to achieve the best results.

VIII. ACKNOWLEDGEMENTS

The authors would like to acknowledge Shrirang Yardi (Georgia Institute of Technology) and Paula Petrica (Cornell University) for their prior work on parallel simulation at Intel that helped motivate and enable this work. We would also like to acknowledge Sandip Kundu (University of Massachusetts Amherst) for his insightful discussions and feedback.

REFERENCES

[1] J. Reinders, “Intel Threading Building Blocks”, O’Reilly, 2007, ISBN 978-0-596-51480-8

[2] J. Emer, P. Ahuja, E. Borch, C.-K. Luk, S. Manne, S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, T. Juan, “Asim: A Performance Model Framework”, IEEE Computer, Feb. 2002

[3] J. Emer, C. Beckmann, M. Pellauer, “AWB: The Asim Architect’s Workbench”, MOBS, May 2007

[4] K. C. Barr, R. Matas-Navarro, C. Weaver, T. Juan, J. Emer, “Simulating a Chip Multiprocessor with a Symmetric Multiprocessor,” Boston Area Architecture Workshop (BARC), Jan. 2005

[5] K. M. Chandy, J. Misra, “Asynchronous Distributed Simulation via a Sequence of Parallel Computations,” Commun. ACM, Nov. 1981

[6] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, H. Angepat, “FPGA-Accelerated Simulation Technologoies (FAST): Fast, Full-System, Cycle-Accurate Simulators,” Intl. Symp. Microarchitecture, 2007

[7] R. M. Fujimoto, “Parallel Discrete-Event Simulation,” Communications of the ACM (CACM), 33(10):30-53, Oct. 1990

[8] D. R. Jefferson, “Virtual Time,” ACM Transactions of Programming Languages and Systems, 7(3):404-425, July 1985

[9] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, S. Bellenot, “Distributed Simulation and the Time Warp Operating System.” ACM, 1987

[10] P. Konas, D. Poulsen, C. Beckmann, J. Bruner, P.-C. Yew, “Chief: A Simulation Environment for Studying Parallel Systems,” Intl. J. Computer Simulation, 1994

[11] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, J. Emer, “A-Ports: an efficient abstraction for cycle-accurate performance models on FPGAs,” In *Proceedings of the international Symposium on Field Programmable Gate Arrays* February, 2008

[12] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind; J. Emer, “Quick Performance Models Quickly: Closely-Coupled Partitioned Simulation on FPGAs”, ISPASS, 2008

[13] D. A. Penry, D. Fay, D. Hodgson, R. Wells, G. Schelle, D. I. August, D. Connors, “Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-Processors,” Intl. Symp. High Performance Computer Architecture, 2006

[14] D. Reed, “Parallel Discrete Event Simulation: A Case Study”, Symposium on Simulation, Tampa FL, 1985

[15] D. A. Reed and A. D. Malony "Discrete Event Simulation: The Chandy-Misra Approach," *Society for Computer Simulation Multiconference*, San Diego, CA, February 3-5, 1988

[16] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, D. A. Wood, “The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers,” ACM SIGMETRICS, May 1993

[17] S. Srinivasan, P. F. Reynolds, “Elastic Time,” ACM Trans. Moduleing and Computer Simulation, 8(2):103-139, Apr. 1998

[18] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, D. I. August, “Microarchitectural exploration with Liberty,” Intl. Symp. Microarchitecture, 2002

[19] WWW Computer Architecture Page. http://www.cs.wisc.edu/arch/www/tools.html

[20] B. Calder et al., “SimPoint 3.0: Faster and More Flexible Program Analysis”, MoBS, 2005

[21] T. Wenisch, R. Wunderlich, B. Falsafi, J. Hoe, “TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes”, SIGMETRICS, 2005

[22] Open SystemC Initiative. http://www.systemc.org

[23] D. Culler, J. Singh, A. Gupta, “Parallel computer architecture: a hardware/software approach”, Morgan Kaufman Pub., 1999

[24] D. A. Wood, M. D. Hill, “Cost-effective parallel computing,” IEEE Computer, Feb. 1995

[25] C. R. Mechoso, J. D. Farrara, J. A. Spahr, "Achieving Superlinear Speedup on a Heterogeneous, Distributed System," *IEEE Concurrency*, 2(2):57-61, June 1994

[26] U. Nagashima, S. Hyugaji, S. Sekiguchi, M. Sato, H. Hosoya, “An experience with super-linear speedup achieved by parallel computing on a workstation cluster: Parallel calculation of density of states of large scale cyclic polyacenes”, *Parallel Computing* 21(9):1491-1504, Sep. 1995

[27] C.-K. Luk et al, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”, SIGPLAN Conference on Programming Languages Design and Implementation, 2005