Analysis of PARSEC Workload Scalability

Gabriel Southern and Jose Renau
Dept. of Computer Engineering, University of California, Santa Cruz
{gsouther, renau}@ucsc.edu

***Abstract*—PARSEC is a popular benchmark suite designed to facilitate the study of CMPs. It is composed of 13 parallel applications, each with an input set intended for native execution, as well as three reduced-size simulation input sets. Each benchmark also demarcates a Region of Interest (ROI) that indicates the parallel code in the application. The PARSEC developers state that users should model only the ROI when using simulation inputs; in other cases the native input set should be used to obtain results representative of full program execution.**

We analyzed the runtime scalability of PARSEC using real multiprocessor systems and present our results in this paper. For each benchmark we analyzed the runtime scalability of both the ROI and full execution for all the input sets. We found that for 6 of the benchmarks the ROI scalability matches that of the full program regardless of the input set used. For the remaining 7 benchmarks, for at least some of the input sets there is significant divergence between the scalability of the ROI and the full program. Three of these benchmarks have much lower scalability for the full program than the ROI, even when run with the native input set. We found that for most of the benchmarks the runtime scalability of the simulation inputs differs significantly from that of the native input set, both for the ROI and the full program.

I. Introduction

The PARSEC benchmark suite is "designed to provide parallel programs for the study [of] CMPs" [6]. It was introduced in 2008 and has been widely used for computer architecture research since then. Developed with the needs of researchers in mind, it has features that make it easier to use with architectural simulators. Each benchmark has multiple input sets, including three that are intended to run with simulators (*simsmall, simmedium, simlarge*), and one that is intended to be representative of a real application (*native*).[1] This allows users to simulate a smaller workload but obtain results representative of a real workload. Each benchmark also defines a *Region of Interest* (ROI) indicating which part of the benchmark executes in parallel. By simulating only the ROI, PARSEC users can reduce simulation time. The ROI is also important for ensuring that results obtained using simulation inputs are representative of real program behavior [5].

Choices in input set size and whether to model the whole program, or only the ROI, can lead to different interpretations when analyzing benchmark results. For instance, Figure 1 shows the scalability of ROI only and full benchmark execution for blackscholes running on a 48-core system and using the native input set. The ROI achieves a maximum speedup of 43 times, while the maximum speedup of the full benchmark is less than 9 times.

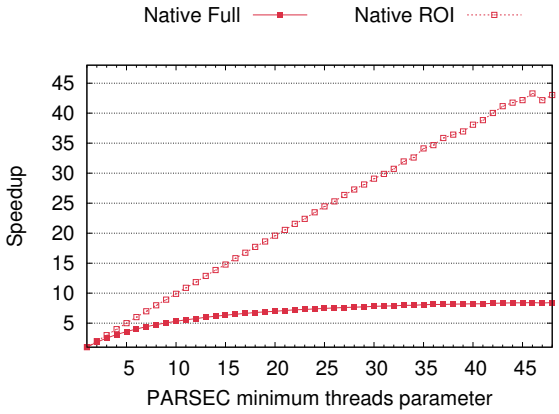

Fig. 1. Speedup for the full execution of Blackscholes is much less than ROI, when using the native input set and running on a 48-core system.

Several papers characterize the behavior of the PARSEC benchmark suite [1], [2], [4], [5], [6], [7], [8], [10], [23] but none of them compare the runtime scalability of the native input sets with that of the simulation input sets, nor do they compare the runtime scalability of the ROI to that of the full program. This paper provides this missing characterization and highlights the importance of PARSEC users reporting data about what input set was used in their own papers.

In this paper we measure the runtime scalability of the four main PARSEC input sets, and we compare the scalability of the ROI with that of the whole program. We do so by running the benchmarks on real multicore systems, varying the number of threads, and measuring the *runtime* of the ROI and the full program for the four different input sets. Our contributions are as follows:

- First systematic analysis of the runtime scalability of all PARSEC input sets. We found that in many cases different inputs to the same benchmark have very different scalability characteristics
- First systematic analysis of the runtime scalability of the ROI compared with full-program execution for all input sets. We identify 6 benchmarks where the scalability

[1]Two additional input sets (*test* and *simdev*) are included for simulator testing and are not appropriate for scientific studies.

of ROI and full program execution has significantly different behavior, while for the other 7 benchmarks ROI and full program execution have very similar behavior.

The rest of this paper is organized as follows: Section II provides background about PARSEC; Section III describes our experiment setup; Section IV provides detailed results; Section V surveys related work; and Section VI concludes.

II. Background

PARSEC was developed between 2005 and 2009 as part of a collaboration between Princeton and Intel [5]. The developers' goal was to create a benchmark suite of emerging parallel workloads that would help architects and researchers design emerging multicore and multiprocessor systems. After its initial release in 2008, PARSEC quickly became popular among computer architecture researchers and has since been widely used in published research.

Six different input sets are defined for each benchmark: *test, simdev, simsmall, simmedium, simlarge, native*. Test and simdev should only be used to test that the benchmark can run. Native is intended to approximate realistic input indicative of how the benchmark application would be used in practice. The remaining three simulation inputs were created by scaling down the native input sets in a way that maintained a representative mix of instructions. The inputs were selected so that serial execution of the native input sets on a real machine should complete in 15 minutes or less, while the simlarge, simmedium, and simsmall inputs should complete execution within 15 seconds, 4 seconds, and 1 second respectively.

The inputs set scaling process skews the amount of time spent in serial phases compared to parallel phases. As a result PARSEC defines an ROI for each benchmark that marks the parallel phase of the benchmark. Although the PARSEC documentation stresses the importance of only using results from the ROI, our results show that this is only important for 6 of the 13 benchmarks.

PARSEC supports three different threading models: pthreads, OpenMP, and Intel Thread Building Blocks (TBB). PARSEC also allows users to specify the *minimum* number of threads run with the benchmark by setting a parameter (n) when the benchmark is started. Table I shows the correspondence between the user-specified number of threads and how many threads the benchmark actually spawns.

In most cases there is a single main thread which spawns n worker threads, but some of the benchmarks use a pipelined parallelization model and spawn multiple threads for each one the user specifies. In addition x264 spawns twice as many threads as there are frames in its input (native has 512 frames), but it uses the parameter n to limit how many threads run in parallel. Most of the benchmarks allow n to range from 1 up to at least 128; however, there are a few restrictions:

TABLE I
Benchmarks and threads in PARSEC

Benchmark	Threads
blackscholes	$1+n$
bodytrack	$2+n$
canneal	$1+n$
dedup	$3+3n$
facesim	$1+n$
ferret	$3+4n$
fluidanimate	$1+n$
freqmine	n
raytrace	$1+n$
streamcluster	$1+2n$
swaptions	$1+n$
vips	$3+n$
x264	$1+2\times$ *frames*

- Facesim is limited to the values 1, 2, 3, 4, 6, 8, 16, 32, 64, 128.
- Swaptions is limited to the number of entries in its input set (16 for simsmall, 32 for simmedium, 64 for simlarge, and 128 for native).
- Fluidanimate requires the number of threads to be a power of 2.
- x264 is limited by the number of frames in its input set.[2] We restricted n to 1–8 for simsmall and 1–32 for simmedium in our experiments.

In this paper we analyze the 13 benchmarks and input sets first released with PARSEC 2.0.[3] We use the native, simlarge, simmedium, and simsmall input sets, and we use pthreads for all benchmarks except freqmine (which requires OpenMP). We vary the number of threads using the minimum threads parameter n and we report n as the parameter of interest in our results instead of reporting how many threads were actually spawned.

III. Experiment Setup

When PARSEC was developed multicore systems only had a few cores, and most of the initial characterization of PARSEC was done using simulators. In the intervening years the number of cores available in mainstream server systems has increased significantly. We wanted to understand how benchmark performance using the simulation inputs compared to the native inputs when running on real systems, which have overheads and bottlenecks that are not always accounted for when using cycle accurate simulation.

We analyzed the scalability of PARSEC by running the benchmarks on three different real multicore/multiprocessor systems and measuring the runtime. The systems each had different microarchitectures and were developed by two different processor vendors.

[2]This limitation is not reported when the benchmark is launched, but in our experiments we observed the output was not correct for n greater than 9 for simsmall and n greater than 33 for simmedium.

[3]We used PARSEC 3.0 downloaded from the PARSEC website, but there are minimal changes between 2.0, 2.1 and 3.0 for the benchmarks we analyzed.

The precise scalability results are specific to the systems that we used for our evaluation. However, we expect that the relative scalability trends we identified will apply to most systems because often the underlying cause is differences in workload distribution between the various different input sets or between the ROI and full benchmark execution. The configuration of the systems we used are as follows:

- A single CPU system with 4 cores, 2 threads per core, for a total of 8 logical processors, along with 16 GB of RAM.
- A dual socket system with 8 cores per socket, 2 threads per core, for a total of 32 logical processors, along with 64 GB of RAM.
- A quad socket system with 12 cores per socket, 1 thread per core, for a total of 48 logical processors, along with 64 GB of RAM.

The detailed system specifications are shown in Table II. In the rest of this paper we refer to the 8-logical processor system as *M8*, the 32-logical processor system as *M32*, and the 48-logical processor system as *M48*.

TABLE II
SPECIFICATIONS OF SYSTEMS USED FOR EXPERIMENTS

System	Configuration
M8	1 x Intel Xeon E3-1275 v3 (4 core, 2-way SMT) 32 KB L1, 256 KB L2, 8 MB L3 cache 16 GB DRAM
M32	2 x Intel Xeon E5-2689 (8 core, 2-way SMT) 32 KB L1, 256 KB L2, 20 MB L3 cache 64 GB DRAM
M48	4 x AMD Opteron 6172 (12 core) 64 KB L1, 512 KB L2, 5 MB L3 cache 64 GB DRAM

All of the systems used the x86_64 version of Arch Linux with version 3.18.6-1 of the Linux kernel. All benchmarks were compiled with version 4.9.2 of gcc/g++. We disabled ASLR but did not do any other special tuning. The OS and hardware were allowed to schedule threads and control CPU frequency using default scheduling algorithms. We used PARSEC hooks to identify the ROI, and for each configuration we recorded the runtime of the ROI and full benchmark execution.

We repeated each experiment at least 10 times and calculated the mean and the confidence interval at a 95% confidence level. For configurations where the initial confidence interval after 10 runs was not within 5% of the mean we repeated the experiment until the confidence interval was within 5% of the mean. The speedup results we present are computed by dividing the mean execution time of a system, input set, and ROI or full configuration with a single thread by the mean execution time of the same configuration with multiple threads.

IV. RESULTS

The two main questions that we sought to answer are: how does the scalability of the ROI compare to the scalability of full benchmark execution? And how does the scalability of each of the simulation inputs compare to the native input set? In this section we present the results of our analysis. First in Section IV-A we analyze the theoretical maximum speedup of the full input sets and identify where it is limited in comparison to the ROI. Next we analyze the actual speedup results we measured using our systems. In Section IV-B we present results for the maximum speedup for each of the input sets; Section IV-C compares the average speedup of the ROI to full, and of the native input set to each of the simulation inputs. Finally, Section IV-D presents a graphical view of scalability trends for each of the benchmarks along with some insights about the reasons different input sets have differing scalability characteristics.

A. ROI Percentage

The ROI is the only part of the PARSEC benchmarks that executes in parallel, and thus the only part where parallel execution can speedup the benchmark. Figure 2 shows the percentage of time that each benchmark spent executing the ROI with the number of threads $n = 1$. Seven of the benchmarks spent over 98% of their execution time in the ROI for all input sets, and so the ROI percentage is unlikely to limit potential speedup from parallel execution.

The other six benchmarks spent less than 90% of their execution time in the ROI for at least some of their input sets, and the maximum theoretical speedup is less than 10 times. Table III lists the percentage of time each of these benchmarks spends executing in the ROI along with the maximum theoretical speedup. The maximum speedup is calculated using Amdhal's law and not listed for input sets where it is not a bottleneck.

TABLE III
ROI PERCENTAGE AND MAXIMUM THEORETICAL SPEEDUP

Benchmark	Native		Simlarge		Simmedium		Simsmall	
blackscholes	89	9	89	9	88	9	87	8
bodytrack	100	–	94	16	86	7	68	3
canneal	81	5	31	1	31	1	23	1
facesim	100	–	69	3	–			
fluidanimate	100	–	89	9	88	9	90	10
raytrace	70	3	20	1	9	1	4	1

- **Blackscholes**: The time outside of the ROI is spent initializing the input array and writing out the results. This amount of work scales linearly with the input set size; consequently a larger input set does not improve the scalability. We confirmed this experimentally by creating an input set 10 times larger than the native input set included with PARSEC and measured the same ROI percentage for this larger input set.

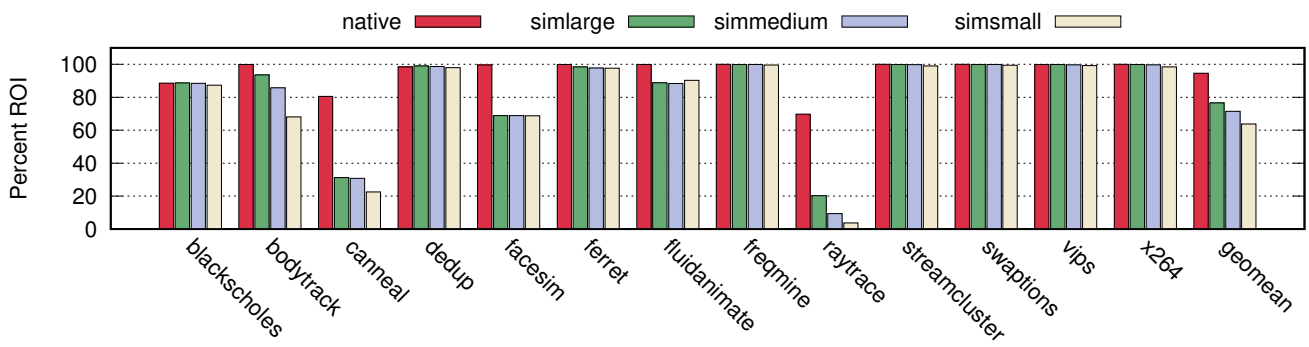

Fig. 2. Percentage of full execution time that is in the ROI measured by running benchmarks on M8 with a single thread. Seven of the benchmarks spend nearly 100% of their execution time in the ROI, while the other six spend significantly less for at least some input sets.

- **Bodytrack**: uses helper threads to load image data for the next frame concurrently with the threads processing the current frame. However, the thread pool must be created and the first image loaded before any parallel computation can occur. This initialization is done before the ROI starts. A negligible percentage of total execution time is consumed for the native input set, but a significant fraction of time is consumed for the two smallest input sets.
- **Canneal**: The majority of the time outside of the ROI is spent initializing the netlist. This initialization time is proportional to the size of the netlist. However, there are two ways to increase the work done by the benchmark. Either the netlist size can be increased, or the number of temperature steps can be increased. We tested using 10 times as many temperature steps for the native input set with the same netlist, which increased the ROI percentage to 98%.
- **Facesim**:[4] There is a fixed amount of work done outside the ROI based on the facial features that will be animated. The work done to animate each frame is in the ROI, and this work scales with the number of frames. The native input set processes 100 frames, while the simulation inputs process only a single frame.
- **Fluidanimate**: The benchmark simulates fluid dynamics for use in animation sequences. The work outside of the ROI is mostly involved with partitioning how a single animation frame is processed. The work in the ROI scales with the number of frames in the workload, and adding frames adds more work in the ROI. The native input set has 500 frames, while the simulation inputs only have 5 frames.
- **Raytrace**: It is possible to increase the amount of work in the ROI by rendering more frames. The native input set renders 200 frames; when we increased this to 2,000 frames the ROI increased to 95%.

[4]Facesim only has one simulation input set

B. Maximum Speedup

The fraction of time that a benchmark spends executing parallel code is not the only limiting factor on its scalability. In many cases other factors, such as inter-thread communication and imbalances in workload distribution, limit scalability more than the fraction of code that can be executed in parallel.

Figures 3 and 4 show the maximum speedup for each input set on the M48 and M32 systems respectively. Since M48 has 48 cores the maximum expected speedup is 48 times (assuming no superlinear effects). The M32 system has 16 cores, and each core can execute 2 threads simultaneously, so the maximum linear speedup is 32 times. However, since simultaneous multithreading shares core resources it is unlikely that benchmarks will have linear speedups for all 32 threads.

Blackscholes executing the ROI of the native input comes closest to achieving the maximum linear speedup. But most of the benchmarks and input set combinations have a much lower maximum speedup. As expected the six benchmarks we identified with low ROI percentage show a big difference between the speedup of full and the ROI. Thus results from ROI and full may not be comparable in these cases and it is important for users to properly specify which region of the benchmark they measured.

The divergence between the speedup of ROI for each of the native inputs and the speedup of ROI for each of the simulation inputs is potentially more problematic because results obtained using simulation inputs may not be representative of actual application behavior. But in many instances simulators are used because real hardware is not available.

C. Quantifying Similarity

The maximum speedup results presented in the previous section demonstrate that there are bottlenecks which limit the scalability of the PARSEC benchmarks, and that these bottlenecks affect different benchmark and input set combinations in different ways. But comparing maximum speedup

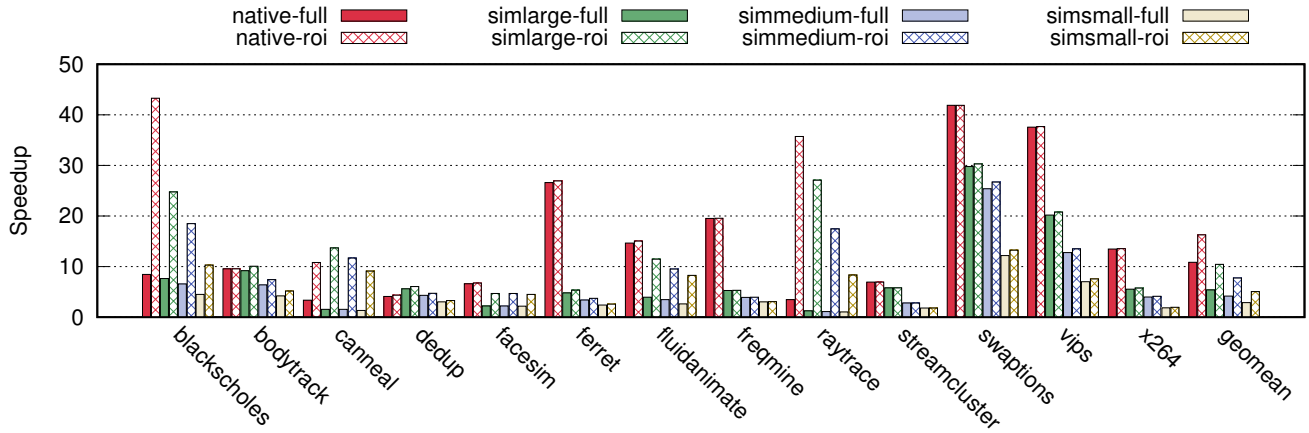

Fig. 3. Maximum speedup measured on M48 for each benchmark, region, and input set combination.

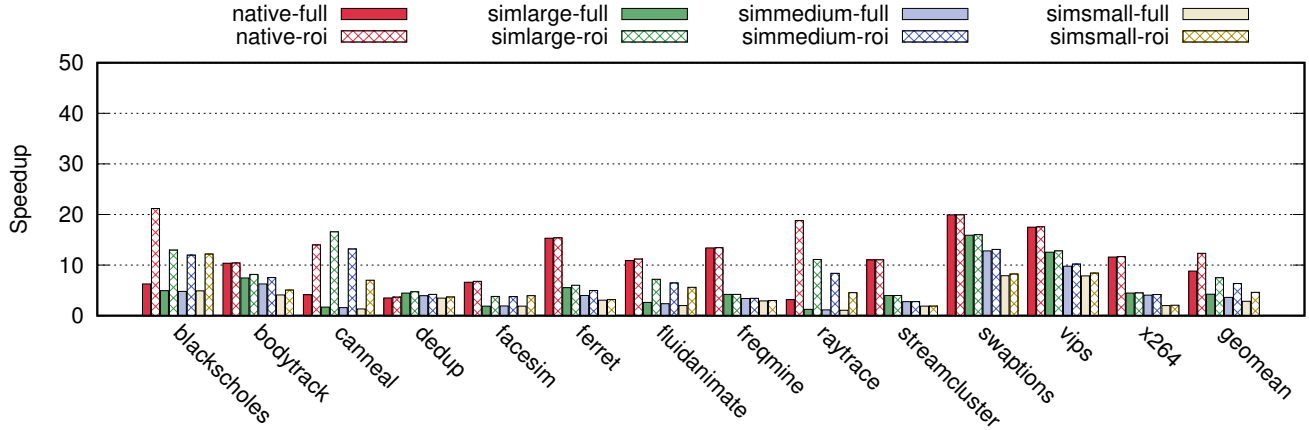

Fig. 4. Maximum speedup measured on M32 for each benchmark, region, and input set combination. Scale is same as Figure 3

only shows the scalability difference for a single data point. We also quantified the average scalability difference between ROI and full, and between the native input set and each of the simulation inputs.

ROI and Full: Table IV shows the percentage difference between the average speedup of the ROI and full benchmark execution averaged over all data points. As an example of how this was calculated consider blackscholes. When running with the full native input set on the M32 system with 8 threads results it has a 4.3 times speedup compared to a single thread, while the speedup of 16 threads is 5.6 times. For the same configuration the speedup of the ROI is 7.7 times for 8 threads and 15.0 times for 16 threads. The average speedup of these two points is 5.0 times for full and 11.4 times for ROI. The difference of these averages is 6.4, and this difference is 128% of the average of full. The reason for presenting the difference as a percentage of full is that the same absolute speedup difference represents less relative error if the total speedup for the full execution is larger. This explanatory example uses only two data points, but the results

TABLE IV
SPEEDUP % DIFFERENCE: ROI AND FULL

Benchmark	Native	Simlarge	Simmedium	Simsmall
blackscholes	174	129	108	93
bodytrack	0	6	11	15
canneal	151	528	463	368
dedup	2	4	4	7
facesim	1	55	–	
ferret	1	9	14	6
fluidanimate	1	59	65	77
freqmine	0	0	1	2
raytrace	394	887	735	414
streamcluster	0	0	0	1
swaptions	0	1	2	2
vips	0	2	4	5
x264	0	2	4	3
geomean	1	9	11	14

TABLE V
SPEEDUP % DIFFERENCE: NATIVE AND SIMULATION

Benchmark	Simlarge		Simmedium		Simsmall	
	full	roi	full	roi	full	roi
blackscholes	10	28	20	54	41	104
bodytrack	11	9	39	24	96	69
canneal	104	15	109	10	142	40
dedup	30	31	15	16	22	24
facesim	89	21	–			
ferret	218	192	337	289	505	471
fluidanimate	99	21	115	24	152	30
freqmine	150	149	230	228	347	337
raytrace	134	28	162	62	176	169
streamcluster	231	231	961	962	3449	3482
swaptions	16	15	33	31	39	36
vips	31	29	75	69	197	181
x264	117	114	140	131	127	120
geomean	61	49	96	79	155	139

in Table IV were calculated by taking the average of all of the data points for each configuration that was compared.

For six of the seven benchmarks that we identified as spending nearly all of their time in the ROI (dedup, freqmine, streamcluster, vips, x264) the average speedup difference is usually less than the 5% margin of error we set when computing average execution time.

Although ferret spends 98% of its execution time in the ROI for the simulation inputs, it also has low scalability. For instance on the M32 system the maximum speedup for the simmedium input set is 4 times for full execution and 5 times for ROI. As a result ferret stands out as having a significant percentage difference between ROI and full execution despite spending nearly all of its execution time in the ROI when running with a single thread.

The remaining six benchmarks are ones we identified as spending a significant percentage of benchmark execution time in the single threaded region. Of these, bodytrack has a relatively small percentage difference between ROI and full, but this is because its overall maximum speedup is less than 10 times.

The other five benchmarks (blackscholes, canneal, facesim, fluidanimate, raytrace) have very large relative differences between the speedup of ROI and full for at least some of the input sets. Consequently it is not a good idea to compare full and ROI results when using these benchmarks. This is particularly noteworthy for blackscholes, canneal, and raytrace where even the native input sets have large speedup differences between ROI and full.

Native and Simulation: Table V shows the average speedup difference between the ROI and full for each of the input sets. These results were computed using a methodology similar to the ones used to compare the percentage difference in ROI and full speedup. In this case the speedup for each simulation input was compared to that of the native input. The percentage difference is calculated by dividing the average speedup difference of the native input by the average speedup difference of the simulation input.

The first thing that stands out is that none of simulation inputs are particularly similar to the native input set. The best case is for bodytrack executing the ROI for the simlarge input set at 9% speedup difference compared to the native input set. As expected the larger simulation input sets are generally more similar to the native input set. However, even for the ROI of simlarge the geometric mean for all the benchmarks of the speedup difference compared to the native input set is nearly 50%.

The most noteworthy outlier is streamcluster where in the worst case for simsmall the speedup differs by over 3,000%. The reason for this is that streamcluster is able to maintain at least slight speedup when executing the native input set. But it suffers extreme slowdown for small simulation inputs. For instance when running on M48 with 48 threads using the simsmall input set, streamcluster is 100 times slower than when running the same input set with a single thread.

D. Measured Scalability

The previous section quantifies the similarity of the different input sets. In this section we present a visualization of the scalability data. We varied the number of threads from 1 to the number of logical processors in the system and measured the runtime of full execution and ROI for all benchmarks using all input sets. For each data point we plotted the speedup of the multithreaded benchmark execution compared to executing with a single thread. Although we plotted 8 data sets for each benchmark, in some cases fewer points are visible because the ROI and full results overlap completely. We split the results into three figures, which are interspersed along with benchmark analysis. Figure 5 shows the scalability of blackscholes, bodytrack, canneal, dedup, facesim, and ferret on both the M48 and M32 systems. Figure 6 shows the scalability of fluidanimate, freqmine, raytrace, streamcluster, swaptions, and vips. It uses the same legend as Figure 5 but the legend is not repeated in order to save space. Finally Figure 7 shows the scalability of x264, again using the same legend shown in Figure 5.

Blackscholes has the best scalability of any benchmark for the ROI with the native input set. The maximum speedup of the full execution is limited to less than 9 times because of the serial portion of the benchmark. It is also noteworthy that even for the ROI, the simulation inputs do not scale as well as the native input set for large numbers of threads. On M48 the execution time of the ROI for the simsmall input when running with 48 threads is 14.4 ms. We tested a modified version where the worker threads spawn and return immediately without doing any work and the ROI time dropped to 4 ms. Thus it does not appear that the thread creation overhead prevents further scaling of the benchmark.

Bodytrack has relatively good similarity between ROI and full for native and simlarge. For simmedium and simsmall,

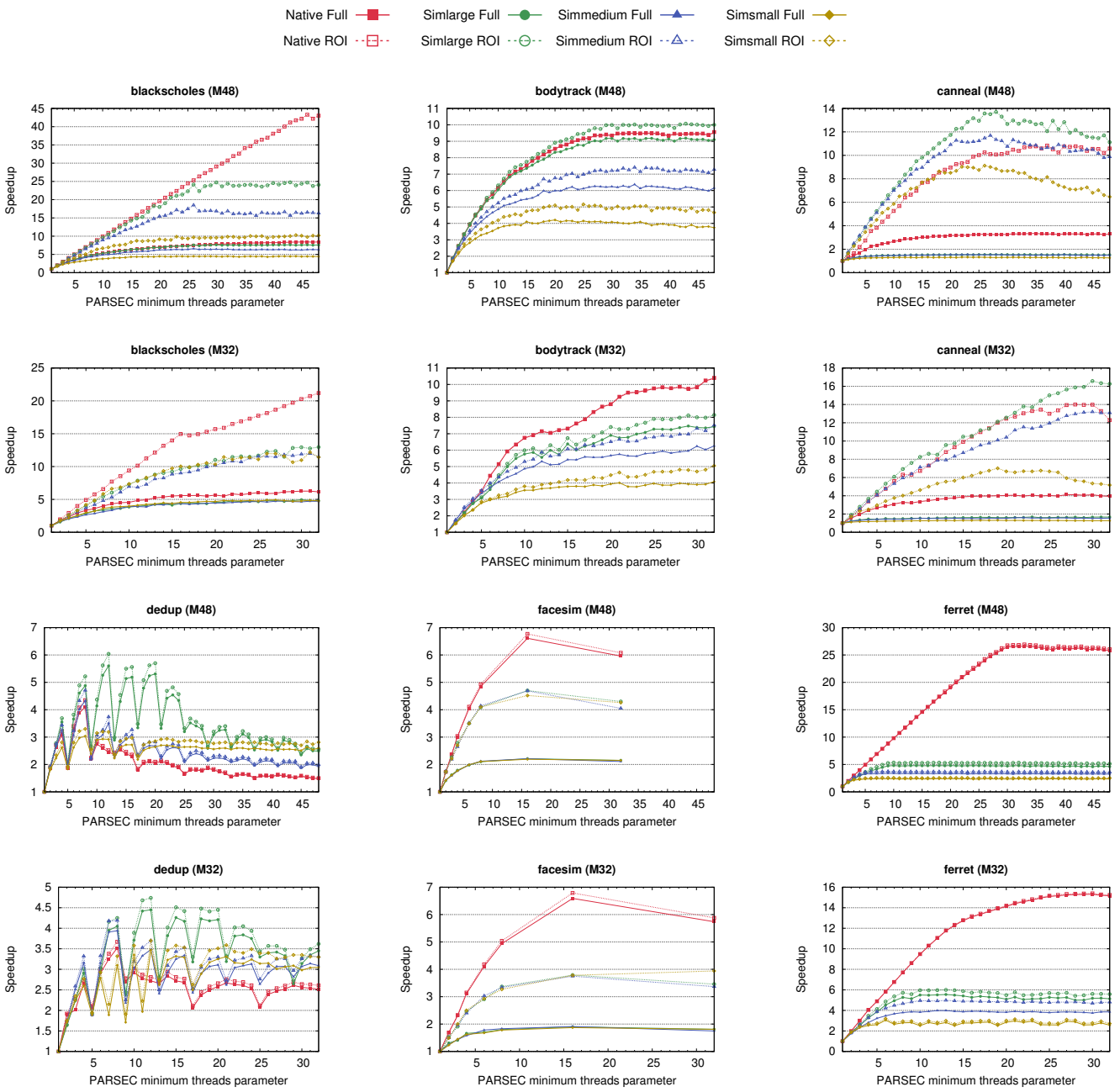

Fig. 5. Scalability of blackscholes, bodytrack, canneal, dedup, and facesim on M48 and M32 systems. Native, simlarge, simmedium, and simsmall are represented respectively with squares, circles, triangles, and diamonds. Points representing ROI are hollow and those representing full are solid.

speedup of full and ROI drops noticeably, particularly on M48 with many threads.

Canneal has limited scalability for full for all of the input sets with a maximum speedup of less than 5 times for native input. Even for ROI only, the total scalability is limited and on M48 the speedup of ROI drops after roughly 30 threads. Canneal uses atomic operations to synchronize data between threads; as a result adding more threads increases the chance of conflicts between threads. There is a tradeoff between the size of the netlist and the number of temperature steps. We think this is why the simlarge input has higher scalability than native. When we tested with an input with more time steps than native we found that the scalability of full improved, but the scalability of the ROI dropped.

Dedup has an erratic speedup pattern due to a work distribution imbalance between threads. Dedup creates queues for partitioning work between threads, and the number of queues created is $n\ threads/4+n\ threads \bmod 4$. Afterwards

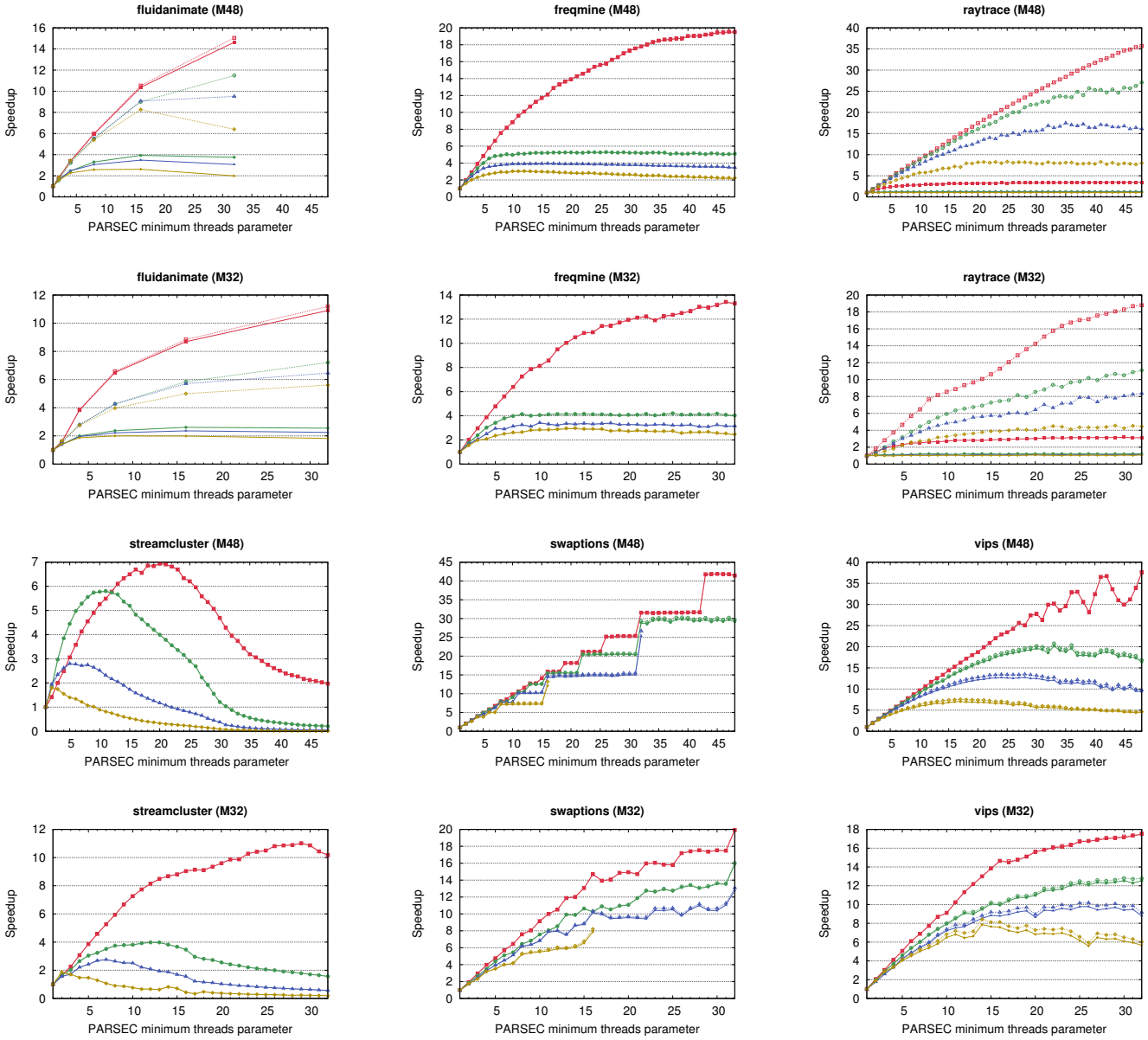

Fig. 6. Scalability of fluidanimate, freqmine, raytrace, streamcluster, swaptions, and vips. Same legend as Figure 5.

each queue is assigned 4 worker threads, except the last queue, which has *n threads* mod 4 threads. This workload imbalance causes the scalability to be best when the number of threads is a multiple of 4. It is also noteworthy that the simlarge input set has a maximum speedup with 12 user threads while the other inputs sets have a maximum speedup with 8 user threads, and the maximum speedup of simlarge input is much higher than that of native for both ROI and full. We suspect this is caused by differences in the input set data. Dedup performs deduplication, and simlarge achieves 2.38X compression factor, while native, simmedium, and simsmall achieve 1.05X, 1.06X, and 1.09X compression respectively.

Facesim has much lower scalability for the simulation input than for the native input set for both full and ROI. The lower scalability of full is explained by the fraction of the benchmark that is in the ROI. Since simulation and native inputs both process the same frame, we expect the differences between the scalability of the two inputs are related to additional initialization overhead that is included in the ROI time but not amortized over multiple frames.

Ferret also has much lower scalability for the simulation

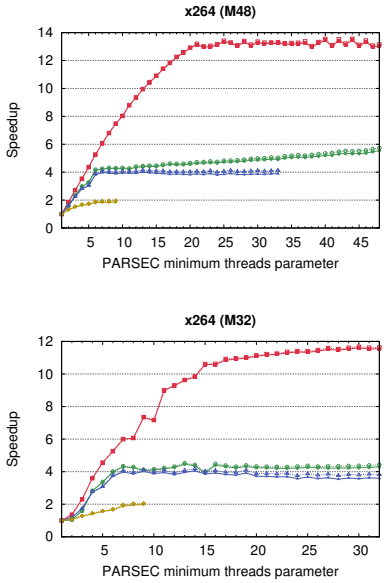

Fig. 7. Scalability of x264. Same legend as Figure 5

inputs than for native. It uses pipelined parallelization, but the first and last stages in the pipeline only spawn a single thread. We experimented with removing these stages from the ROI, but that did not improve the ROI scalability for the simulation inputs. Bienia [5] notes that the simulation inputs for ferret have less opportunity for parallel execution, while Pusukuri et al. [23] found that the speedup for the native input set was limited by lock contention. Although we do not have a definitive explanation for the scalability difference, we do note that the simulation inputs reach their maximum speedup much earlier than when using the native input, and this maximum speedup is much lower than when using native inputs.

Fluidanimate's low scalability for full execution of the simulation inputs is explained by the lower fraction of the benchmark in the ROI. However, even for the ROI the scalability of the simulation inputs is lower than that of the native input set. We suspect this is due to the thread communication overhead that is proportional to the number of particles in the input set, and so the overhead is more for the smaller input sets. It is also noteworthy that the speedup of simsmall on M48 *drops* when increasing from 16 to 32 threads.

Freqmine is another benchmark with much lower scalability for the simulation inputs than the native input set. Both the native and simulation inputs have a high percentage of their work included in the ROI. However, freqmine uses OpenMP and we suspect that the smaller simulation inputs have their scalability constrained by the serial portions of the workload and that the parallel loops are too small to provide much speedup.

Raytrace has extremely limited scalability for the full execution because of the low fraction of the benchmark in the ROI. Increasing the number of frames from the 200 used by the native input set to 2,000 increases the maximum speedup from 3.5X to 16X. Cebrián et al. [10], [11] argue that the version of raytrace in PARSEC should be replaced by one with SIMD instructions. It is also noteworthy that the ROI for the simulation inputs does not scale nearly as well as the native input set, particularly for simmedium and simsmall.

Streamcluster uses barrier based synchronization, and the scalability for simmedium and simsmall is much worse than for native. On M48, running simsmall with 48 threads is 68 times *slower* than running with 1 thread. Of even greater concern, the maximum speedup for simsmall on M48 occurs when the PARSEC minimum thread parameter is 2, and afterwards the performance worsens as more threads are added. Roth et al. [25] observed similar behavior and attributed it to inefficiency in the barrier synchronization.

Swaptions has very similar scalability for the ROI and the full program. The simulation inputs also match native scalability at some points but diverge at others. The application itself has a stairstep type of scalability caused by an imbalance in workload distribution between the threads. Earlier papers [23], [25] also identified this problem and our results support their observations.

Vips also uses a pipelined parallel programming model with two threads for performing I/O and then n threads for processing the data, and we suspect this is what causes worse scalability for smaller input sets. Bienia's dissertation [5] notes that the size of the output buffers can limit parallelism, and that this problem may be corrected in future versions of PARSEC. Although most of our tests used the benchmark code from PARSEC 3.0, for vips we reverted to PARSEC 2.1 because the source code for PARSEC 3.0 was missing ROI annotations. After noting Bienia's comment we tested using the vips code in PARSEC 3.0, but observed the same scalability as PARSEC 2.1.

x264 also has much lower scalability for the simulation inputs than for the native input set. The x264 application compresses an input video stream, and the simulation inputs have fewer and smaller frames than the native input. The smaller inputs have more dependencies between frames, and this limits the overall potential for achieving parallel speedup.

V. Related Work

Christian Bienia's 2011 dissertation [5] is the most comprehensive study of the PARSEC benchmarks and extends material published earlier [4], [6], [7], [8]. The characterization of PARSEC in Bienia's work relies on simulation and is intended to be machine independent. In contrast, our characterization is done using real machines and we focus on runtime as performance metric of interest.

Pusukuri et al. [23] developed *Thread Reinforcer* to pick an optimal number of threads for a parallel application. They evaluated their proposal using 8 of the 13 PARSEC

benchmarks running on a 24-core system. They used the native input sets for their evaluation, and like us, found that the maximum speedup of the full execution of blackscholes and canneal was limited due to the fraction of serial code. Part of the motivation for our study was noting differences in Bienia and Pusukuri et al.'s characterization of PARSEC's scalability.

Several other papers have also studied the problem of thread scheduling and included characterization of some of the PARSEC benchmarks as part of their evaluation [18], [19], [20], [21], [22], [24], [26].

There are also several papers that characterized the performance of PARSEC. Like us, Bhadauria et al. [2] studied the scalability of PARSEC workloads using real machines, but they did not compare different input sets, or ROI and full program execution. Barrow-Williams et al. [1] analyzed communication patterns between threads in PARSEC and SPLASH using Simics. Bhattacharjee and Martonosi [3] analyzed TLB behavior of PARSEC benchmarks using a combination of native execution and simulation. Ferdman et al. [16] analyzed the single threaded performance of a variety of benchmark suites including PARSEC. Cebrián et al. [10], [11] proposed extending PARSEC with better support for SIMD hardware. Bryan et al. [9] examined how synchronization overhead and other system-level effects limited the potential scalability of PARSEC 1.0 benchmarks.

Several papers have also analyzed the scalability of some of the PARSEC benchmarks while developing techniques to find performance bottlenecks in parallel applications [12], [13], [14], [15], [17], [25].

VI. Conclusion

Benchmarks are a critical part of the quantitative approach to computer architecture research. But the complex interaction of the many layers of the computing stack, coupled with the slow speed of architectural simulators, forces architects to make approximations when simulating benchmark execution. PARSEC provides reduced size input sets and demarcates a ROI as ways to reduce simulation time while still approximating the behavior of the actual workload. However initial characterization of the PARSEC input set scalability was performed using simulation and it may have overlooked bottlenecks that exist in real systems.

Our characterization of the scalability of PARSEC using real multiprocessor systems shows two important ways in which results can vary depending on benchmark parameter selection. First, the choice between measuring ROI and full only has a significant impact on the scalability of seven of the benchmarks. Second, we showed that the scalability of the four different input sets that PARSEC provides differs dramatically when the benchmarks are executed on a real system. We recommend that users of PARSEC report the parameters selected for their experiments (both input set size and whether measuring ROI or full).

References

[1] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterization of SPLASH-2 and PARSEC," in *IISWC 2009*, October 2009.

[2] M. Bhadauria, V. M. Weaver, and S. A. McKee, "Understanding PARSEC performance on contemporary CMPs," in *IISWC 2009*, October 2009.

[3] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors," in *PACT 2009*, October 2009.

[4] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *IISWC 2008*, September 2008.

[5] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *PACT 2008*, October 2008.

[7] C. Bienia and K. Li, "PARSEC 2.0: A new benchmark suite for chip-multiprocessors," in *MoBS 2009*, June 2009.

[8] ——, "Fidelity and scaling of the PARSEC benchmark inputs," in *IISWC 2010*, December 2010.

[9] P. Bryan, J. Beu, T. Conte, P. Faraboschi, and D. Ortega, "Our many-core benchmarks do not use that many cores," in *WDDD 2009*, June 2009.

[10] J. Cebrian, M. Jahre, and L. Natvig, "Optimized hardware for suboptimal software: The case for simd-aware benchmarks," in *ISPASS 2014*, March 2014.

[11] ——, "Parvec: vectorizing the parsec benchmark suite," *Computing*, pp. 1–24, 2015.

[12] C. Curtsinger and E. D. Berger, "Coz: Finding code that counts with causal profiling," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, 2015.

[13] S. Dutta, S. Manakkadu, and D. Kagaris, "Classifying performance bottlenecks in multi-threaded applications," in *MCSoc 2014*, September 2014.

[14] D. Eklov, N. Nikoleris, and E. Hagersten, "A software based profiling method for obtaining speedup stacks on commodity multi-cores," in *ISPASS 2014*, March 2014.

[15] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *ISPASS 2012*, April 2012.

[16] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *ASPLOS 2012*, March 2012.

[17] W. Heirman, T. Carlson, S. Che, K. Skadron, and L. Eeckhout, "Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads," in *IISWC 2011*, November 2011.

[18] J. Lee, H. Wu, M. Ravichandran, and N. Clark, "Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications," in *ISCA 2010*, June 2010.

[19] R. Moore and B. Childers, "Inflation and deflation of self-adaptive applications," in *SEAMS 2011*, May 2011.

[20] ——, "Using utility prediction models to dynamically choose program thread counts," in *ISPASS 2012*, April 2012.

[21] ——, "Program affinity performance models for performance and utilization," in *DATE 2014*, March 2014.

[22] A. Navarro, R. Asenjo, S. Tabik, and C. Caşcaval, "Load balancing using work-stealing for pipeline parallelism in emerging applications," in *ICS 2009*, June 2009.

[23] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via OS level monitoring," in *IISWC 2011*, November 2011.

[24] ——, "Thread tranquilizer: Dynamically reducing performance variation," *TACO*, January 2012.

[25] M. Roth, M. J. Best, C. Mustard, and A. Fedorova, "Deconstructing the overhead in parallel applications," in *IISWC 2012*, 2012.

[26] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *PLDI*, June 2014.