A Pragmatic Approach for Predicting the Scalability of Parallel Applications

EPFL Technical Report EPFL-REPORT-174869.

Aleksandar Dragojević
IC, EPFL, Switzerland
aleksandar.dragojevic@epfl.ch

Rachid Guerraoui
IC, EPFL, Switzerland
rachid.guerraoui@epfl.ch

ABSTRACT

Predicting the scalability of parallel applications is becoming crucial now that the number of cores in modern CPUs doubles roughly every two years. Traditional ways to get some understanding of the scalability of a parallel application rely on extensive experiments or detailed application models. Both are very time consuming and often hard to use.

This paper presents *PreSca*, a pragmatic system for predicting the scalability of parallel applications. *PreSca* uses function approximation techniques to model scalability with an analytical *performance function* extracted from a set of measurements. By considering the application as a black-box without requiring any knowledge about its internals, *PreSca* can be applied with little effort to *any* parallel application. We show how *PreSca* can be used statically to predict the scalability of a given application and decide which synchronization primitive scales best for it as well as how it can be used on-line to dynamically assist scheduling decisions and adjust core assignment. In some sense, *PreSca* shows, for the first time, how function approximation can be used to predict the scalability of parallel applications in a completely general way.

We extensively evaluated *PreSca* using a large number of parallel benchmarks, including some that use locks and some that use transactional memory. We also consider two different multicore systems. Our evaluation shows that *PreSca* produces accurate results. More specifically: (1) *PreSca*'s interpolations based on only 8 measurements have 90th percentile of error lower than 15%, (2) *PreSca*'s extrapolations using measurements with up to m cores predict the performance for $n \leq 2m$ cores with errors lower than 20% in most cases, and (3) *PreSca*'s on-line scheduler determines the optimal thread count using fewer than 7 measurements with errors lower than 3% on average.

1. INTRODUCTION

Parallel programming is notoriously challenging because of the need to handle concurrent threads accessing shared data objects. The difficulty is aggravated by the very fact that the number of CPU cores used during the execution of an application is usually much higher than the number used during its development and testing. Typically, applications developed today can be tested on CPUs with 8 cores, but in just 2 or 3 years the same applications are likely to be run on CPUs with 16 or even more cores. This state of affairs clearly calls for tools that predict the scalability of parallel applications.

In many cases, the performance of parallel applications stops improving, or even starts degrading, at some point as the number of cores it uses increases. Figure 1 depicts the performance of two benchmarks on a 48-core system. It shows that the performance of `kmeans low` peaks around 20 threads and that using more threads impacts the performance adversely.[1] On the other hand, the performance of `vacation high` keeps improving as additional threads are assigned to it until 48 threads. One could wonder, for instance, whether it is worth increasing the number of cores assigned to `vacation high` from 24 to 48 to reduce the execution time by 35%, or whether it is more beneficial to assign them to some other application.

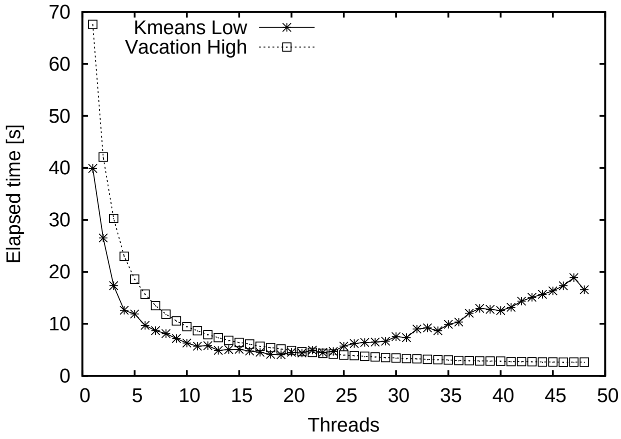


Figure 1: Performance of different parallel applications

The "optimal" number of cores usually depends on many factors, including the used synchronization technique, the characteristics of the application and the computer system. For example, a developer can be tempted by software transactional memory (STM) [27] to synchronize concurrent threads accessing shared data structures. STM requires no hardware support and little effort from the programmers. Unfortunately, STM does not consistently deliver good performance [6, 7], as illustrated in Figure 2. The figure depicts the speedup of parallel code that uses SwissTM [8], one of the fastest state-of-the-art STMs, over sequential, non-instrumented code, for two different benchmarks from the STAMP benchmark suite [20]. Basically, STM delivers good performance only in certain cases—with 64 threads STM outperforms sequential code by almost 30x on the `vacation low` benchmark, but only by 2x on the `intruder` benchmark. Clearly, if the programmers could predict the performance of STM-based application prototypes quickly and accurately, they could decide early in the development process

[1] Each thread is executing on a dedicated core.

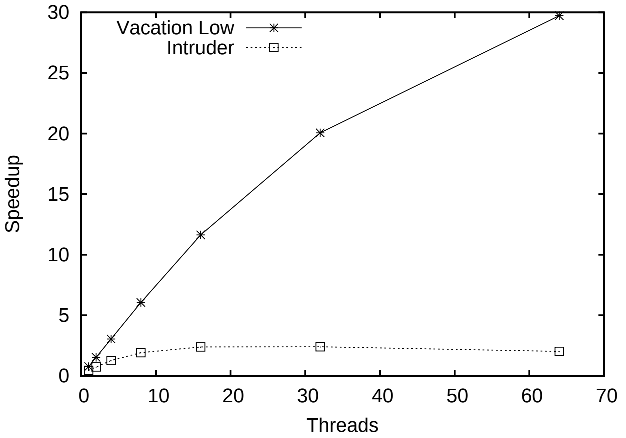


Figure 2: Inconsistency of STM performance

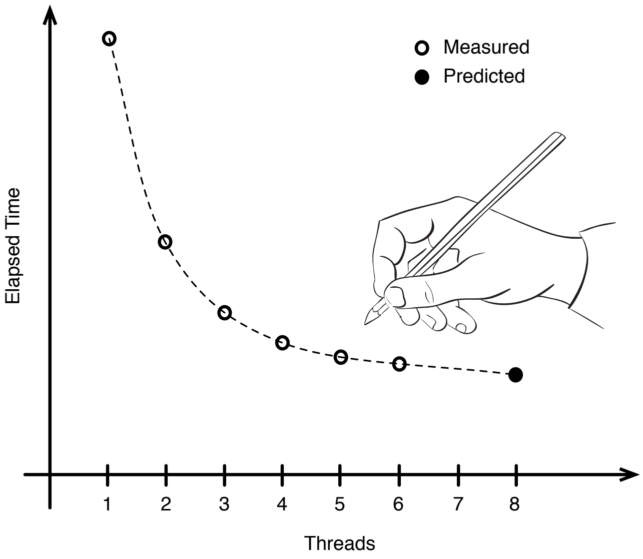


Figure 3: "Eyeballing" approach to predicting scalability

whether to use STM or other, more involved, synchronization techniques, such as fine-grained locking or lock-free programming.

Contribution. Traditional approaches to gain insight into the scalability of parallel applications rely on extensive experiments or detailed, and often application-specific, models [3, 5, 14, 18, 19, 22, 31]. These approaches are time-consuming and require significant effort which makes them difficult to use. This paper presents *PreSca*, a pragmatic system for predicting the scalability of parallel applications. By considering the application as a black-box without requiring any knowledge about its internals, *PreSca* can be applied with little effort to *any* parallel application. As we will show in the paper, *PreSca* can be used *statically* to predict the scalability of an application and decide which synchronization primitive to use as well as *dynamically* to adjust core assignment on the fly.

In a nutshell, *PreSca* takes as input a set of measurements on a number of cores and predicts the performance on a different number of cores. It can be viewed as an automatization of a simple, manual "eyeballing" approach for predicting scalability (Figure 3). The "eyeballing" approach consists of observing the measured performance with *naked eye* and guessing the performance at thread counts for which the measurements are not available. In Figure 3, the performance at 8 threads is predicted based on the measurements with 1–6 threads. To do so, a line that closely follows the measurements is drawn and is extended beyond 6 threads in such a way that it follows the general trend observed in measurements with 1–6 threads, up to 8 threads. In our experience, the manual "eyeballing" approach can be surprisingly accurate if performed by someone experienced.

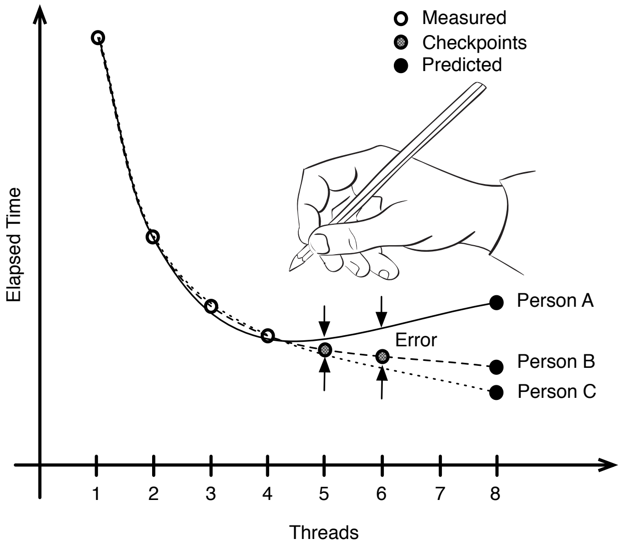


Figure 4: Choosing the best prediction with "eyeballing"

Similarly to the "eyeballing" approach, *PreSca* takes several performance measurements with various thread counts and constructs an analytical *performance function*. The performance function is then used to predict the scalability of the application with thread counts not used for the measurements. The performance function models the whole application and the system it executes on as a black-box and depends on the whole computing environment: application's inputs, synchronization technique, operating system, hardware etc. If any of these changes, *PreSca* needs to construct a new performance function, based on a new set of measurements. Because *PreSca* uses only the performance measurements to construct the prediction function, it does not require any knowledge of application's or system's internals: no modifications to the source code of the application nor to the system components are required.

PreSca constructs the prediction function using well-known approximation techniques [1, 21]. The function approximation takes as input the available data points (e.g. measurements from Figure 3) and the target function type (e.g. polynomial $f(x) = ax^2+bx+c$) and it outputs a specific function that best fits the available data points (i.e. it calculates coefficients a, b and c). *PreSca* uses different function types in cases (1) when the measurements cover the whole range of thread counts (interpolation) and (2) when these cover only a part of the range (extrapolation). In the former case, it uses the polynomial functions. In the latter case, it does not use a predetermined function type. Instead, it chooses the best function type for the application at hand using the available measurements, as illustrated in Figure 4.

In a continued analogy with the manual "eyeballing" approach, we could imagine different people drawing the lines used for predictions, as in Figure 4. Each of the three persons is accurate for some applications, but no one produces the best prediction for all of them. To choose the best prediction, each of the persons draws the lines based only on measurements for 1–4 threads. The remaining measurements are used as *checkpoints* to choose the prediction that has the lowest error at the checkpoints. In the same vein, *PreSca* designates the last c out of m available measurements as the checkpoints and constructs a number of performance functions using different function types and the first $m - c$ measurements. *PreSca* then selects the performance function that has the lowest error at the checkpoints and uses it to predict the scalability. In some sense, *PreSca* shows, for the first time, how function approximation can be used to predict the scalability of parallel applications in a completely general way.

It is important to notice here that we do not claim *PreSca* to be a silver bullet for predicting the performance of parallel applications in all contexts. In particular, it cannot predict the performance of an application for some parallel system based on the measurements from a different system if the characteristics of the two systems differ significantly. However, we can use *PreSca* to understand how much the application would benefit from additional cores available on the same system, or a similar, larger system. *PreSca* can also be used to compare the scalability of different versions of an application implemented using different synchronization techniques (e.g. locks and transactional memory).

Furthermore, and as we will show in the paper, we can also apply our approach to help assign the optimal number of threads to the executing application. We illustrate this by presenting *OPreSca*: a system that can be viewed as on-line execution of *PreSca* to monitor the performance of the application and dynamically adjust the number of cores accordingly. This we believe is particularly important for applications for which performance varies depending on the workload and for which static predictions alone are insufficient.

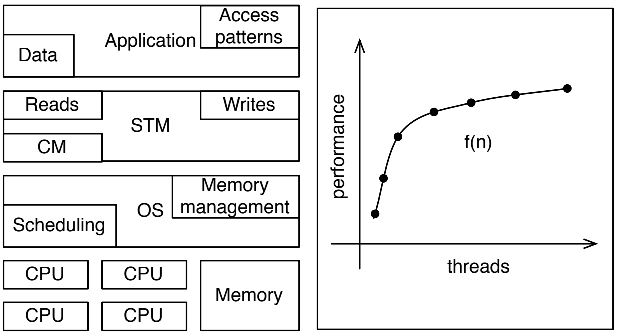
Evaluations. We extensively evaluated the accuracy of *PreSca* using a number of benchmarks and benchmark suites: STAMP [20], PARSEC [4], STMBench7 [9] and the STM micro-benchmarks [7]. We also used two different systems based on the UltraSPARC T2 CPU (with 64 hardware threads) and AMD Opteron 12 core CPUs (48 hardware threads). Our evaluation showed that *PreSca*'s predictions are reasonably accurate and it confirmed that they are indeed useful. In particular, our evaluations showed that:

- The 90th percentile of error of interpolations based on 8 measurements is less than 15% in 62 out of 64 predictions.
- Using measurements with up to m threads, *PreSca* predicts the scalability for $n \leq 2m$ threads with errors lower than 20% in 184 out of 224 predictions performed. Furthermore, in less than 20 cases, the errors are higher than 35%.
- Even slightly inaccurate predictions can be used to gain insight into the scalability of the applications, confirming the usefulness of *PreSca* in various scenarios.
- On average, *OPreSca* uses fewer than 7 measurements and selects the thread count with less than 3% lower performance than the measured highest performance in all 64 cases.

In the rest of the paper, we describe *PreSca* in Section 2 and *OPreSca* in Section 3. We evaluate them in Section 4. We present related work in Section 5. We conclude by discussing the advantages and limitations of our work in Section 6.

2. PRESCA

In this section we describe the notion of performance functions and how they can be used to model the scalability of parallel applications. We also describe how *PreSca* constructs the performance functions.



(a) System components (b) Performance function

Figure 5: The performance function abstracts all the details of a workload and execution environment

2.1 Performance functions

PreSca captures the scalability of the whole system with a single analytical performance function. The performance function takes the number of concurrent threads n as input and outputs the expected performance of the modeled system in the execution with n threads. The performance of the system can be measured in various ways. In this paper we assume that it is either measured as the throughput (i.e. the number of operations executed per second) or as the time required to complete a certain task (e.g. sequencing of a genome or routing a printed circuit).

The performance function abstracts all components of the application and the system it executes on (operating system, synchronization technique, hardware, etc.) and models them as a complete black box. Figure 5 contrasts the detailed view of the application and the system with typical components (Figure 5(a)) and our simple black-box model (Figure 5(b)). Because the performance function only takes the number of threads as the parameter, it is specific to the particular application and system and if any of the underlying components changes, *PreSca* needs to construct a different performance function. This means that *PreSca* actually does not model the scalability of the application, but, instead, the scalability of an application with a particular set of inputs, executing on a particular system. For this reason, in the remainder of the paper we sometimes refer to *PreSca* as a system for modeling the scalability of *workloads*.

2.2 Constructing the performance function

PreSca takes a set of performance measurements, each with a different thread count, and uses function approximation [1] to construct the performance function. More precisely, function approximation takes a set of measurements, a function type (e.g. a polynomial function of degree d) and constructs the function that closely fits the available measurements (e.g. calculates the coefficients of the polynomial). *PreSca* constructs the performance function differently when it is used to predict the performance at thread counts that fall inside (interpolation) and outside (extrapolation) the range of the thread counts for which the measurements are available.

PreSca interpolates performance using polynomials, which are known to be able to approximate any continuous function on a closed interval to any degree of accuracy [15]. With m available measurements, *PreSca* uses polynomials of degree 6 or $m - 2$ if $m \leq 7$. It does not use polynomials of higher degrees to avoid model overfitting.

PreSca extrapolates performance using functions from the kernel depicted in Table 1. In a nutshell, *PreSca* (1) designates a subset

Name	Function
Rat12	$\dfrac{a_0 + a_1 n}{1 + b_1 n + b_2 n^2}$
Rat22	$\dfrac{a_0 + a_1 n + a_2 n^2}{1 + b_1 n + b_2 n^2}$
Rat23	$\dfrac{a_0 + a_1 n + a_2 n^2}{1 + b_1 n + b_2 n^2 + b_3 n^3}$
Rat33	$\dfrac{a_0 + a_1 n + a_2 n^2 + a_3 n^3}{1 + b_1 n + b_2 n^2 + b_3 n^3}$
CubicLn	$a + b\ln(n) + c\ln(n)^2 + d\ln(n)^3$
ExpRat	$e^{\frac{a+bn}{c+dn}}$

Table 1: Kernel of performance function types used for extrapolations

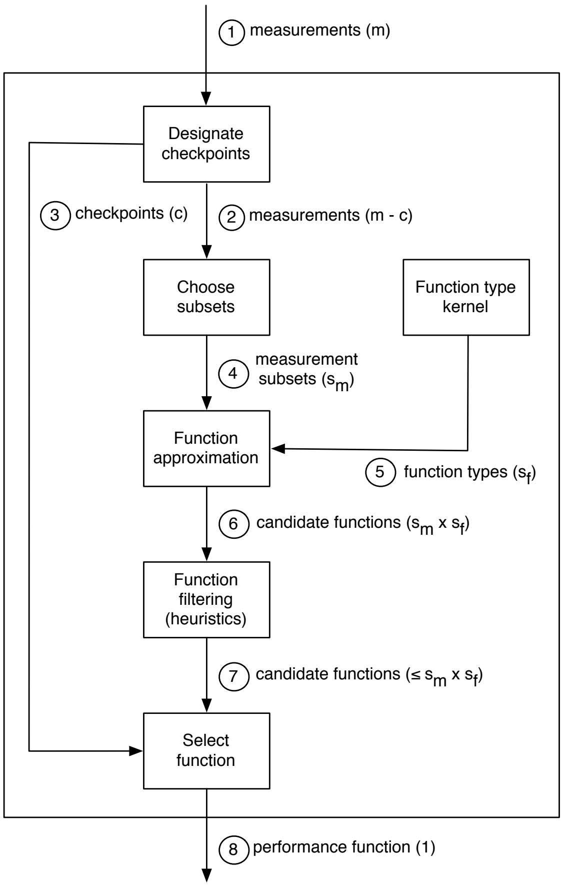


Figure 6: Steps performed by *PreSca* to construct the performance function for extrapolations

of available measurements as *checkpoints*, (2) constructs several candidate performance functions using the kernel function types and the remaining measurements, and (3) selects the most accurate function from the set of candidates using the checkpoints. Figure 6 depicts *PreSca* steps.[2] In the figure, the boxes represent data transformations and the arrows between the boxes represent the data. Each of the arrows is numbered for easier reference and is also annotated with the name of the data set and its size (in brackets).

The input in Figure 6 is a set of m measurements (point 1) and the output is a single performance function (point 8). The input measurements are first split into a set of $m - c$ measurements to be used for approximations (point 2) and a set of c checkpoints that are used to select the best function from a set of candidate functions (point 3). Then, the $m - c$ measurements are split into subsets of different sizes, which produces s_m measurement subsets (point 4). The function approximation step takes all s_m measurement subsets and all s_f function types from the function type kernel and produces one performance function candidate for each pair of measurement subsets and function types. This produces a set of candidate performance functions with $s_m \times s_f$ elements (point 6). Next, *PreSca* eliminates unrealistic functions (e.g. functions with negative values) from the candidate set using several heuristics (point 7). Finally, *PreSca* calculates the overall error for each of the remaining candidate functions at the checkpoints and selects the function with the lowest error (point 8).

```
SUBSET_COUNTS = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20,
    22, 24, 26, 28, 30, 32]
MAX_THREADS = 64
CHECKPOINTS = 4
// Function type kernel
FTYPE_KERNEL = [RAT12, RAT22, RAT23, RAT33, CUBIC_LN,
    EXP_RAT]

fun PrescaExtrapolate(measurements)
    // Designate checkpoints
    checkpoints = GetLast(measurements, CHECKPOINTS)
    all_func = {}

    // Choose subsets
    max_count = size(measurements) - CHECKPOINTS
    for count in SUBSET_COUNTS
        if count > max_count
            break
        // Function type kernel
        for ftype in FTYPE_KERNEL
            // Function approximation
            func = Approximate(measurements, ftype,
                count)
            if func != NULL
                all_func.Add(func)

    // Function filtering (heuristics)
    filtered_func = {}
    for func in all_func
        if ShouldUseFunc(func)
            filtered_func.Add(func)

    // Select function
    SortFuncByErrorAsc(filtered_func, checkpoints)
    return filtered_func[0]

fun ShouldUseFunc(func):
    // filter negative
    for i = 1 to MAX_THREADS
        if func(i) < 0
            return false

    // filter abrupt changes
    X_POW_POS = 8
    X_LIN_NEG = 2./3

    for i = 1 to MAX_THREADS:
        y_max = func(i-1) * pow(i / (i-1), X_POW_POS)
        y_min = X_LIN_NEG * (i-1) / i * func(i-1)
        if func(i) < y_min or func(i) > y_max:
            return false

    return true
```

Algorithm 1: *PreSca* extrapolations pseudo-code for range of 1–64 threads

Pseudo-code for *PreSca* extrapolations for the range of 1–64 threads is given in Algorithm 1. The comments in the algorithm denote the boxes in Figure 6 for easier reference. Function

[2] The accompanying technical report [2] contains its pseudo-code.

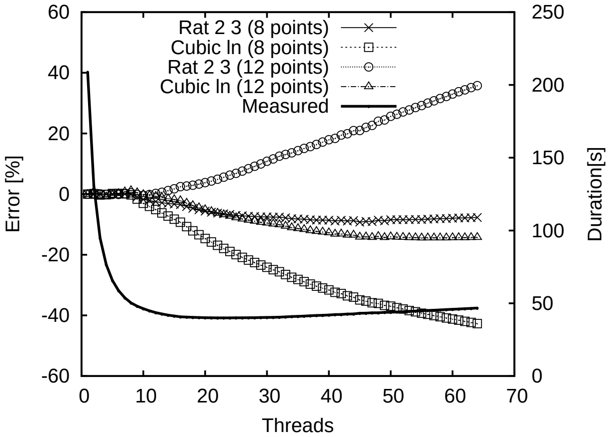


Figure 7: Best `intruder` extrapolations based on 8 and 12 measurements

`PrescaExtrapolate` (line 7) performs the extrapolations. It first designates the highest thread counts as the checkpoints (line 9) and then uses different subsets of the measurements (lines 13–16) and function types from the kernel (line 18) to perform the approximations and construct candidate function set (line 20). Each approximation either produces a function or a `NULL` value (line 21). `NULL` value is produced in cases in which there are not enough measurements to perform the approximation with the selected function type. Next, the set of candidates is filtered using the heuristics (lines 25–28). The filtering is performed by function `ShouldUseFunc` (lines 34–50). The function eliminates all functions with negative values (lines 36–38) and all functions that change too abruptly (lines 41–48) in the target thread count interval. After the filtering, `PrescaExtrapolate` sorts the remaining functions in the order of increasing error at the checkpoints (line 31) and chooses the function with the lowest error overall (line 32) as the performance function.

We discuss some of our design choices and provide the rationale behind them. We motivate the design choices using the predictions from Figure 7. The predictions in the figure do not use *PreSca*. Instead they are obtained by simply performing function approximation with the specified function type and all available measurements. The figure shows the most accurate extrapolations for the `intruder` benchmark using functions from the kernel and measurements for 1–8 and 1–12 threads. The figure depicts the approximation errors for each of the predictions (left y-axis) and the measured performance (right y-axis).

Function type kernel. Figure 7 clearly shows that, even for the same workload, the best function type depends on the set of measurements considered. In the figure, *Rat23* is the most accurate with 8 measurements, whereas *CubicLn* is the most accurate with 12 measurements. For this reason, *PreSca* does not rely on a single function type. Instead it uses the function types from the kernel (Table 1) and chooses the best one for the workload at hand using the checkpoints. We created the kernel empirically, by including in it the function types that accurately extrapolate performance of the workloads we used. Interestingly, polynomials are not in the kernel, as they do not provide accurate performance extrapolations.

Measurements. Figure 7 also illustrates that, surprisingly, it is sometimes better to use only a subset of available measurements instead of using all of them. In the figure, the approximations based on just 8 measurements are better than the approximations based on 12 measurements. Intuitively, small deviations in the measurements with higher thread counts sometimes steer the performance function in the wrong direction, thus producing less accurate predictions despite using more measurements. To avoid possibly negative impact of measurements with higher thread counts *PreSca* splits the measurements into several subsets (between points 2 and 4 in Figure 6). It then chooses the best subset for the workload at hand using the checkpoints.

Candidate function filtering. *PreSca* uses heuristics to detect and discard candidate performance functions that do not provide realistic predictions, such as negative functions (between points 6 and 7 in Figure 6). More precisely, *PreSca* discards all candidate performance functions that (1) are negative for some thread counts of interest or (2) change (increase or decrease) too abruptly. *PreSca* considers that a function changes too abruptly if, for some thread count n, increasing the thread count to $n + 1$ results in: (1) a performance improvement of more than $\frac{3}{2}\frac{n+1}{n}$ or (2) a performance degradation of more than $(\frac{n}{n+1})^8$. The first heuristics follows from the observation that increasing the number of threads by 1 typically improves the performance by a factor of $\frac{1}{n}$ or less. The constant $\frac{3}{2}$ serves as a safety margin. The second heuristics follows from the observation that increasing the number of threads can have only a limited negative impact on the performance. We empirically chose a conservative function to detect the negative performance impact that is too high.

3. OPRESCA

For some workloads, using too many threads increases the contention so much that it negatively impacts workload's performance. For example, `kmeans low` performs best on *x86* with 20 threads (Figure 1) and the performance degrades if more threads are used. Executing `kmeans low` with 48 threads (which is the number of CPU cores in the system) results in a slowdown of more than 4x.

If the characteristics of the workload do not change during the application's execution, *PreSca* can be used to statically predict the best thread count based on several measurements. However, for the applications where the workload dynamically changes, such as a server with a varying rate of incoming client requests, it can be difficult, or even impossible, to statically predict the performance for all possible workloads. In this case, it is useful to dynamically determine the best thread count for the current workload.

In this section we describe *OPreSca*, which uses the performance functions on-line to determine the optimal thread count for a workload (the optimal thread count is the one for which the workload has the highest performance). In a nutshell, *OPreSca* can be viewed as an on-line execution of several instances of *PreSca*: it (1) measures the performance of the workload with different thread counts during short intervals, (2) constructs the performance functions at the end of each interval using all past measurements, and (3) uses the performance functions to estimate the optimal thread count and assigns it to the application for the next interval. *OPreSca* converges when the estimated optimal thread count has already been used in one of the previous intervals. When this occurs, the set of the measurements used to construct the function remains the same, and, hence, the constructed function and the estimate of the optimal thread count also remain the same.

We assume that the workload does not change too frequently and that there are enough intervals between the workload changes for *OPreSca* to converge. The intervals have to be long enough to obtain meaningful performance measurements. Also, we assume that the number of executing threads in the application can be adjusted quickly after *OPreSca*'s decision (i.e. threads can be suspended and resumed quickly).

```
1  fun SchedulingIntervalExpired()
2      thread_count = GetThreadCount()
3      measured[thread_count] = GetPerformance()
4
5      if size(measured) >= 3
6          predicted = Approximate(measured)
7          thread_count = MaxPerfThreadCount(predicted)
8      else
9          thread_count = ChooseInitThreadCount(measured)
10     SetThreadCount(thread_count)
11
12 fun Approximate(measured)
13     max_prf_thread_count = MaxPerfThreadCount(measured)
14     max_thread_count = MaxThreadCount(measured)
15     min_thread_count = MinThreadCount(measured)
16
17     if max_perf_thread_count < max_thread_count and
            max_perf_thread_count > min_thread_count
18         // interpolate
19         poly_deg = min(size(measured) - 1, 6)
20         return ApproxPoly(measured, poly_deg)
21     else
22         // extrapolate
23         if size(measured) >= 7
24             return ApproxRat(measured, 3, 3)
25         else if size(measured) >= 6
26             return ApproxRat(measured, 2, 3)
27         else if size(measured) >= 5
28             return ApproxRat(measured, 2, 2)
29         else if size(measured) >= 4
30             return ApproxRat(measured, 1, 2)
31         else if size(measured) >= 3
32             return ApproxRat(measured, 1, 1)
33
34 fun ChooseInitThreadCount(measured)
35     thread_count = Random() % MAX_THREAD_COUNT
36     while thread_count in measured
37         thread_count = Random() % MAX_THREAD_COUNT
38     return thread_count
39
40 fun WorkloadChange(measured)
41     EmptySet(measured)
```

Algorithm 2: *OPreSca* pseudo-code

OPreSca pseudo-code is given in Algorithm 2. As discussed, *OPreSca* is activated periodically on interval expiry (line 1). First, *OPreSca* reads the number of used threads and the performance measurement for the previous interval and stores them into the set of available measurements (lines 2 and 3). If there are more than 3 measurements (line 5), it approximates the performance of the workload (line 6) and selects the thread count with the predicted best performance (line 7). If there are fewer than 3 measurements, *OPreSca* cannot perform the approximation and instead it selects an arbitrary number of threads (line 9). The selected number of threads is assigned to the application for the next interval (line 10).

OPreSca uses different function types for extrapolations and interpolations. If the best thread count is inside the range of the thread counts used so far (line 13), *OPreSca* interpolates performance using polynomials of degree 6 or less, similarly to *PreSca* (lines 19 and 20). Otherwise, *OPreSca* extrapolates performance using various rational functions (lines 23—32). *OPreSca* does not use *PreSca*'s extrapolation algorithm because it typically operates on fewer measurements than *PreSca*. *OPreSca*'s simpler approach works well because it is used only to determine the optimal thread count for the workload instead of predicting its scalability accurately.

During the initial 3 intervals *OPreSca* assigns the thread counts arbitrarily (line 34). In the pseudo-code, *OPreSca* assigns the thread counts randomly (line 35), only ensuring that the assigned thread count has not been used in the previous intervals (line 36). We experimented with several different initial thread assignments, as discussed in Section 4.

Upon workload change, *OPreSca* clears the current set of measurements and starts over (line 40). In this paper we do not focus on the workload change detection. Instead, we assume that the workload change is detected and signaled to *OPreSca* by another component in the system. For instance, the workload change could be determined simply by detecting that the performance of the workload at the stable state (after *OPreSca* converges) has significantly changed. Alternative, more robust approaches for detecting the workload change also exist and could be deployed [30].

4. EVALUATION

We present an extensive evaluation of *PreSca*'s and illustrate its usefulness on several examples. We also present simulation-based evaluation of *OPreSca*.

4.1 Experimental setup

The evaluation uses performance measurements from a number of parallel benchmarks executed on two computer systems. We first collected the performance measurements, averaged the results over 10 runs and then performed the predictions and scheduler simulations offline.

Computer systems. We used two computer systems. The first has UltraSparc T2 CPU with 8 cores clocked at 1.2GHz, supporting 64 hardware threads with hardware multithreading. The second has 4 AMD Opteron 6172 CPUs with 12 cores each clocked at 2.1GHz, for a total of 48 hardware threads. The two systems have different characteristics: the first has a single CPU which heavily uses hardware multithreading and has low single-thread performance, while the other has 4 CPUs, does not use hardware multithreading and has a much better single-thread performance. In the remainder of the text, we refer to the two systems as *Sparc* and *x86* respectively.

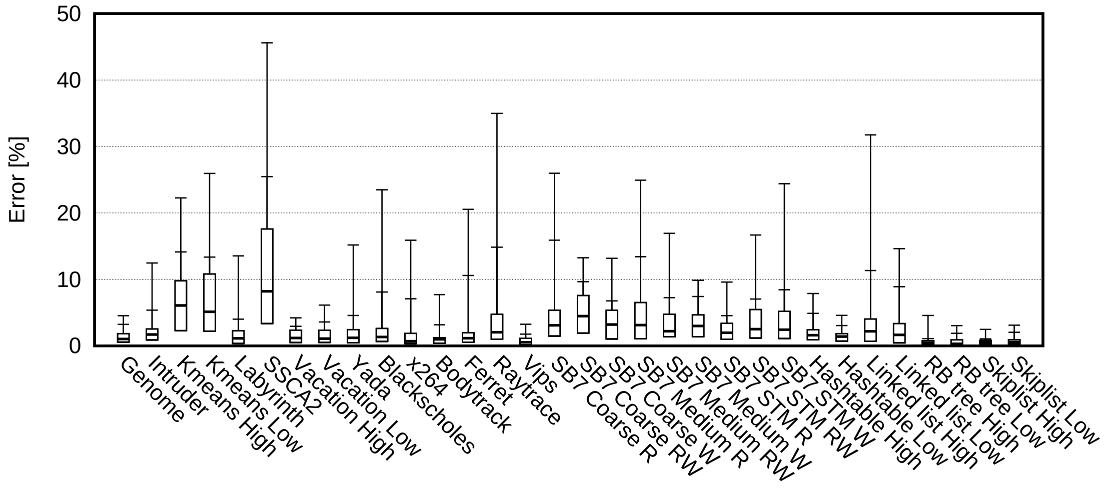
Benchmarks. We used several benchmark suites: STAMP [20], Parsec [4], STMBench7 [9] as well as standard STM microbenchmarks (used in e.g. [6]). These benchmarks span a wide range of workload characteristics, with different lengths of critical sections, levels of contention and synchronization techniques. In total, we used 32 different workloads; 20 STM-based and 12 lock-based.

STAMP [20] is a widely-used benchmark suite for STMs. It consists of 8 benchmarks. The benchmarks are real-world applications from various computation domains. Each of the benchmarks can be configured in a number of ways to obtain workloads with different characteristics. In our experiments we used all benchmarks except `bayes` which exhibits non-deterministic behavior. We configured the benchmarks using the default parameters from STAMP 0.9.10 distribution, which gave us 9 different workloads. We modified STAMP benchmarks to support execution with thread counts that are not a power of two.

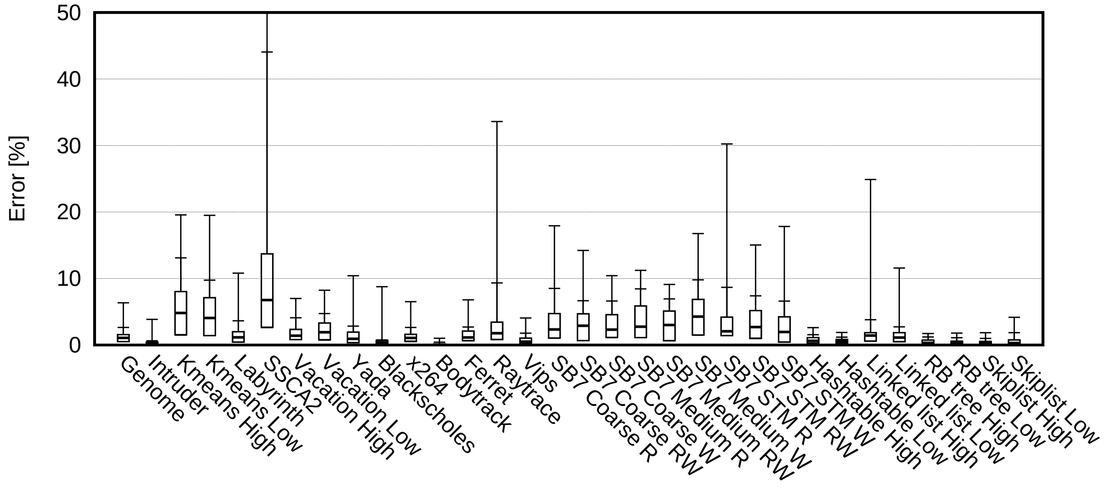
Parsec [4] is a benchmark suite composed of several concurrent applications and kernels. Each of the benchmarks is implemented using several different concurrency control techniques (including locking, STM and message passing). We used 6 lock-based Parsec applications that support execution with thread counts that are not a power of two and that exhibit stable performance.

STMBench7 [9] is an STM benchmark that models complex, large-scale object-oriented workloads. We used the 3 default workloads of STMBench7 with long traversals turned off.

The STM micro-benchmarks are based on different implementations of an integer set. In the experiments, all of the threads access the same integer set and perform a random mix of lookup, insert and remove operations. The level of contention can be changed by varying the element range and operation ratios. We used hashtable,



(a) 6 measurements



(b) 8 measurements

Figure 8: Interpolations for *Sparc* measurements

skiplist, red-black tree and linked list sets and a low and high contention workloads (90% lookups and 65536 elements and 10% lookups and 1024 elements respectively).

All STM-based workloads use SwissTM [7, 8], a state-of-the art STM that performs well under a variety of workloads.

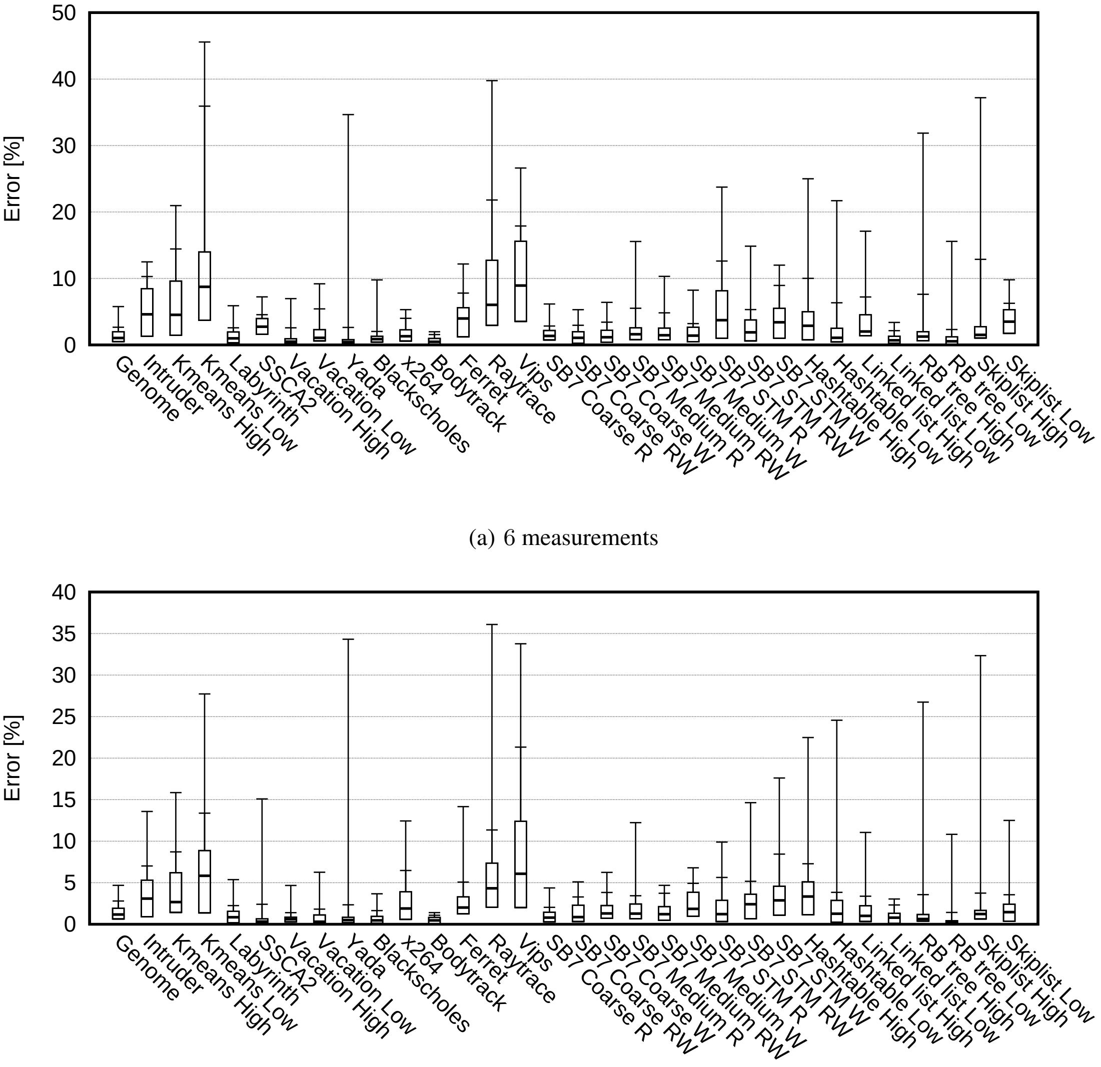
Implementation. We implemented *PreSca* in Python. We used Python's libraries for scientific computing [26] and the framework from [24] to perform the function approximations. The approximations use the downhill simplex algorithm [21] that minimizes the relative error between the function values and the measured data.

4.2 Interpolations

We present our interpolations of performance measurements in Figures 8 and 9. The interpolations are based on measurements with 6 and 8 thread counts which are evenly spread across the range of available thread counts. For each of the workloads, we present the first, second and third quartiles of absolute errors (in the rectangle) and 90th percentile and maximum errors (above the rectangle). For instance, for `genome` on *Sparc* with 6 measurements the errors for three quarters of thread counts are less than 2%, for 90% of thread counts are less than 3% and the maximum error is around 9%.

Sparc. Figure 8 shows that *PreSca* produces accurate predictions for *Sparc* measurements. With 6 measurements (Figure 8(a)) the 90th percentile of error is lower than 15% for 30 out of 32 workloads we used. Furthermore, third quartile of error is less than 10% for 30 workloads and is less than 20% for all of them. Using more measurements further improves the predictions. With only 2 more measurements (Figure 8(b)), *PreSca* predicts the performance with maximum error of less than 20% for all thread counts in 28 workloads, and the 90th percentile of error less than 15% in 31 workloads.

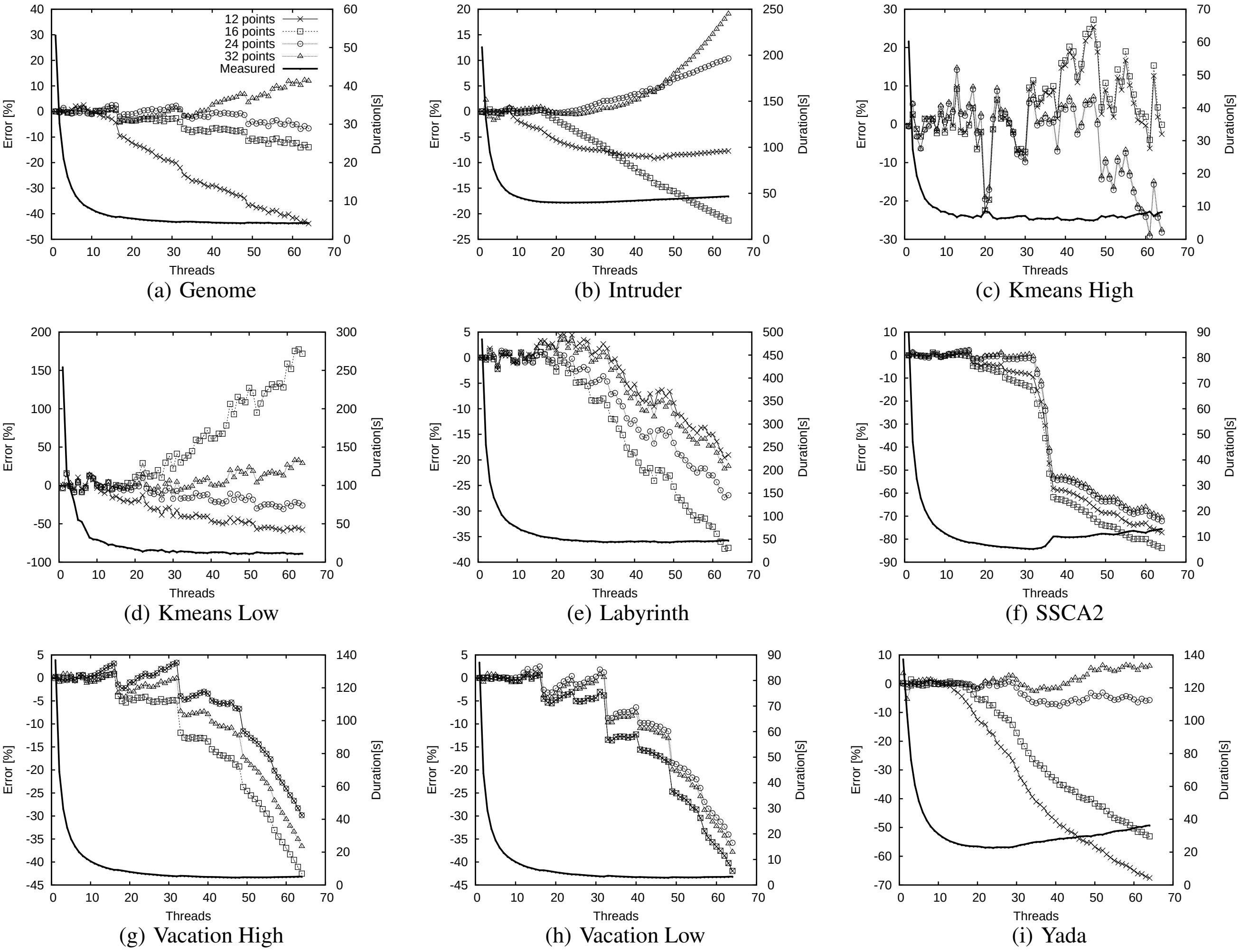
The reason for the error outliers in the predictions are either high variations in the measurements (e.g. in the `kmeans` workloads) or significant changes in the measured performance at some thread counts (e.g. in the `ssca2` and `raytrace` workloads). Despite



(a) 6 measurements

(b) 8 measurements

Figure 9: Interpolations for *x86* measurements



(a) Genome

(b) Intruder

(c) Kmeans High

(d) Kmeans Low

(e) Labyrinth

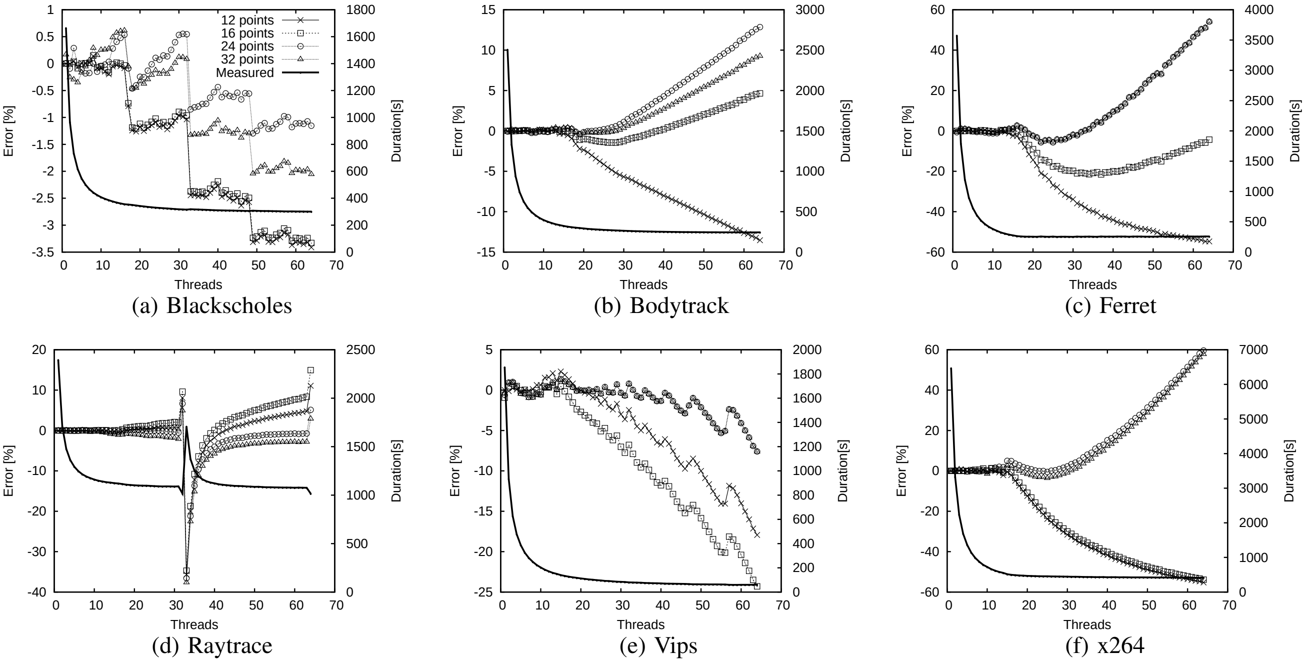
(f) SSCA2

(g) Vacation High

(h) Vacation Low

(i) Yada

Figure 10: Predictions for STAMP on *Sparc* using 4 checkpoints



(a) Blackscholes (b) Bodytrack (c) Ferret

(d) Raytrace (e) Vips (f) x264

Figure 11: Predictions for Parsec on *Sparc* using 4 checkpoints

the outliers, the predictions for the majority of thread counts are accurate.

x86. *PreSca* provides equally accurate predictions for *x86* measurements. Figure 9 depicts the interpolation errors for *x86* measurements. The figure shows that *PreSca* achieves the same accuracy for *x86* and *Sparc* measurements, but that there are more outliers due to higher variance in measurements on *x86* in general. With 6 measurements (Figure 9(a)), the 90th percentile of error is less than 15% for 31 out of 32 workloads we used and it is less than 15% for 29 of them. The third quartile of error is less than 10% for 29 workloads and less than 20% for all workloads. With 8 measurements (Figure 9(b)), the accuracy of the predictions further improves. The maximum error is less than 20% for 24 workloads and the 90th percentile of error is less than 10% for 29 workloads.

Similarly to predictions for *Sparc* measurements, the main reason for the error outliers are high variations in measurements (e.g. `kmeans` workloads) and significant changes in the measured performance (e.g. `raytrace` and the microbenchmark workloads).

4.3 Extrapolations

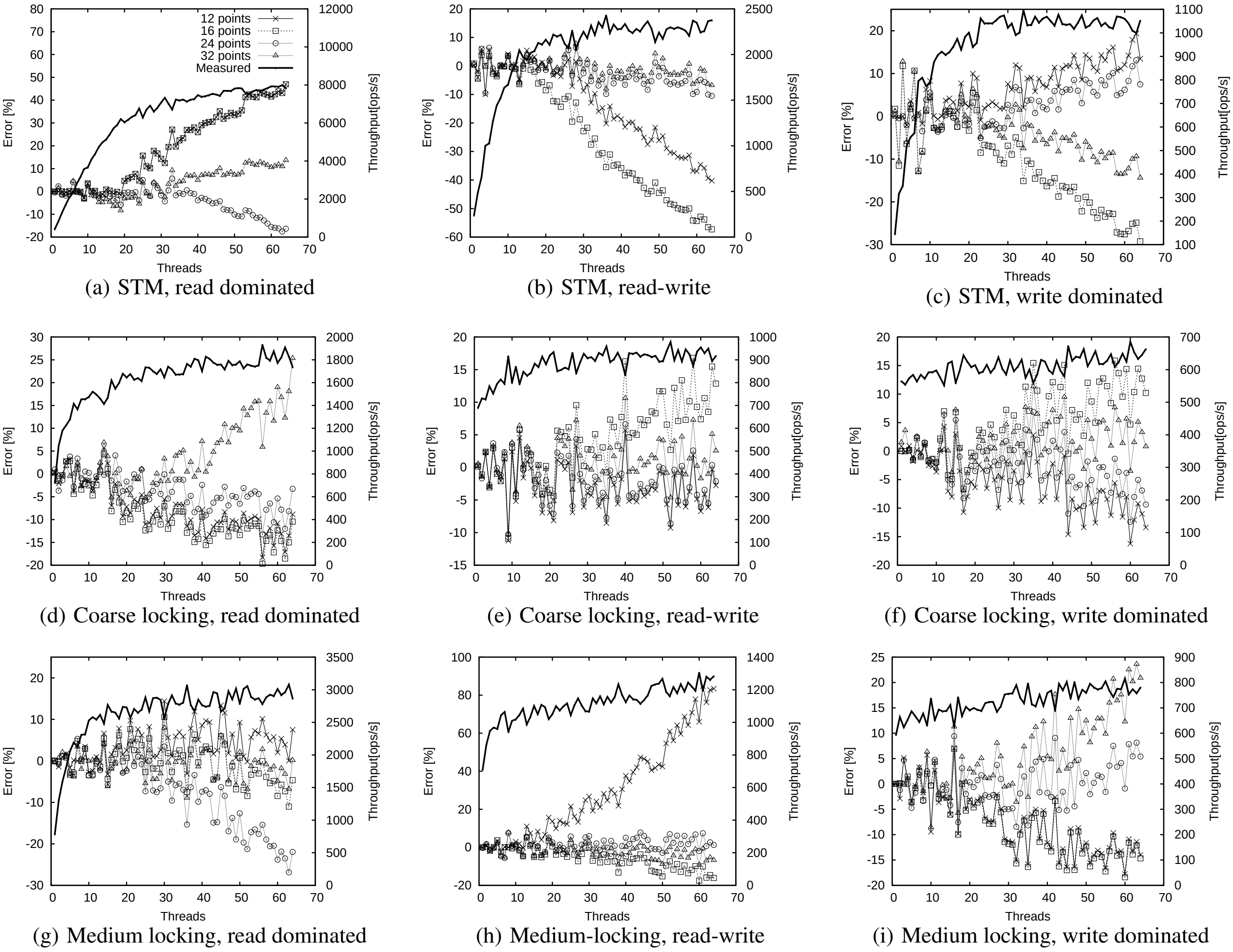
We present our performance measurements extrapolations in Figures 10—17. The figures depict the prediction errors for extrapolations based on measurements with 1 to m threads, where $m = 12, 16, 24, 32$ for *Sparc* and $m = 12, 16, 24$ for *x86*. The measurements with the highest 4 thread counts were the checkpoints. Each of the graphs conveys errors for the predictions (left y-axis) and the measured performance (right y-axis) to put the predictions in perspective.

4.3.1 Sparc

STAMP. Figure 10 depicts the prediction errors for STAMP workloads on *Sparc*. In general, *PreSca*'s performance predictions are good, as *PreSca* accurately predicts the performance of all workloads except `ssca2`.

With $m = 12$ and $m = 16$ measurements, the prediction errors are less than 20% up to $2m$ threads for all workloads except `kmeans low`, where the errors are slightly higher due to high variance in the measured performance. With $m = 24$, the prediction errors are less than 20% up to 48 threads for all workloads except `ssca2`. The performance of `ssca2` changes significantly around 32 threads which results in lower accuracy of *PreSca*'s predictions based on fewer than 32 measurements. With $m = 32$, the prediction errors are less than 20% up to 50 threads for all workloads except `ssca2`. With more than 50 threads, the errors are slightly higher for the `kmeans high`, `kmeans low`, `vacation high` and `vacation low` workloads. Overall, the predictions based on m measurements have errors of less than 20% up to $2m$ threads in 28 out of 36 cases. Furthermore, errors are higher than 35% only for 2 `ssca2` predictions.

The predictions for the `ssca2` workload reveal the biggest limitation of *PreSca*'s black-box approach. The execution times of `ssca2` suddenly increase at around 32 threads. The reason for this is the architecture of UltraSparc T2 CPU. The CPU uses hardware multithreading to support 64 hardware threads on top of 8 cores. This means that the characteristics of the CPU effectively change at multiples of 8 threads. To accurately predict the impact of the CPU's architecture on the performance of a particular workload one has to reason about the internal details of the CPU and the workload—depending on the workload the impact varies from negligible to significant. When using measurements with up to 32 threads, *PreSca* cannot anticipate the sudden drop in performance for `ssca2` with more than 32 threads. This is why the prediction errors for `ssca2` become much higher (60-75%) with more than 32 threads. The reason for lower accuracy of the predictions of the `vacation` workloads' performance is similar. With both `vacation` workloads, the execution times decrease slower with more than 32 threads than with fewer than 32 threads. As *PreSca* bases the predictions on the measurements with up to 32 threads, it cannot accurately predict this change which results in higher errors with more threads.



(a) STM, read dominated

(b) STM, read-write

(c) STM, write dominated

(d) Coarse locking, read dominated

(e) Coarse locking, read-write

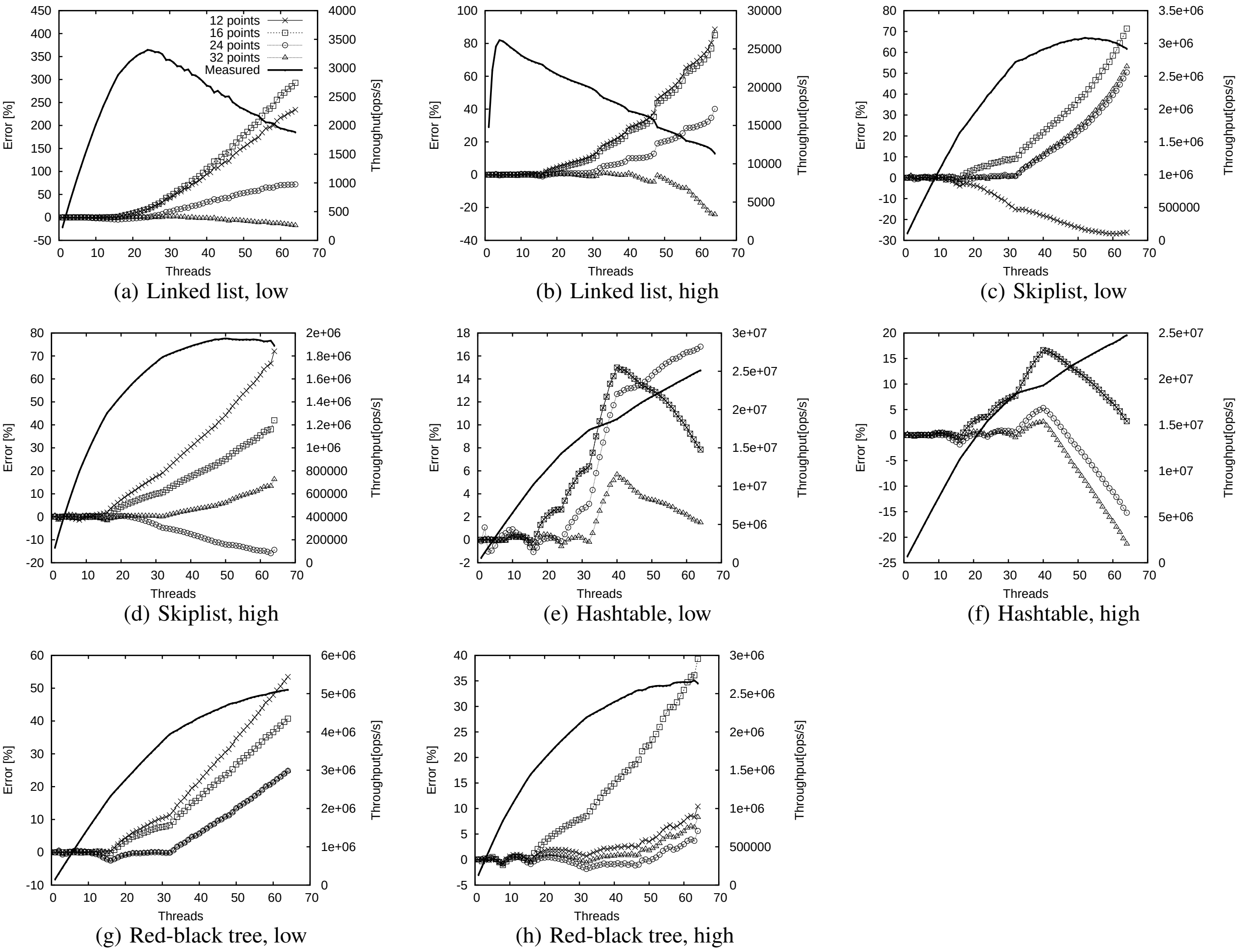
(f) Coarse locking, write dominated

(g) Medium locking, read dominated

(h) Medium-locking, read-write

(i) Medium locking, write dominated

Figure 12: Predictions for STMBench7 on *Sparc* using 4 checkpoints



(a) Linked list, low

(b) Linked list, high

(c) Skiplist, low

(d) Skiplist, high

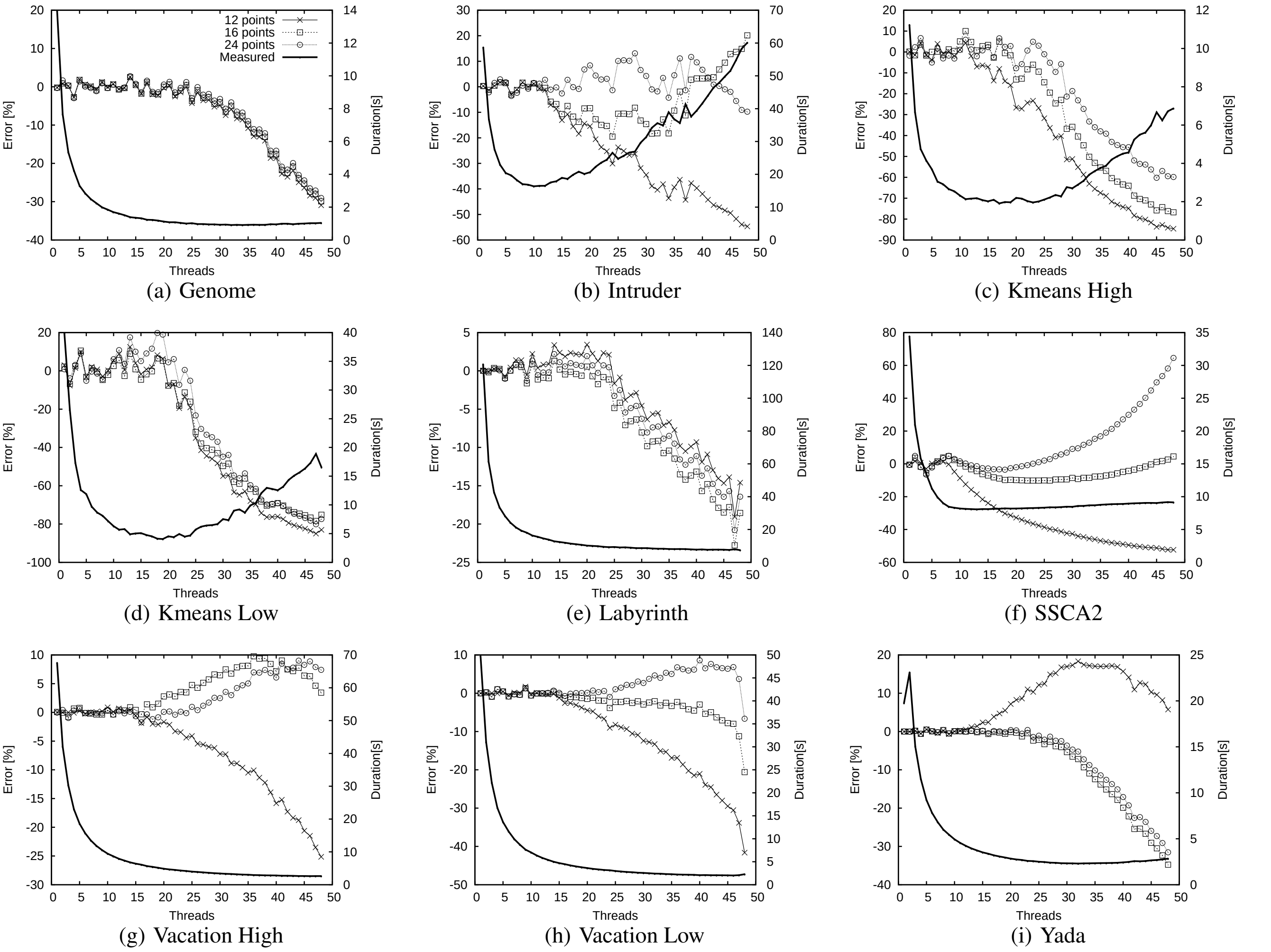
(e) Hashtable, low

(f) Hashtable, high

(g) Red-black tree, low

(h) Red-black tree, high

Figure 13: Predictions for STM micro-benchmarks on *Sparc* using 4 checkpoints



(a) Genome (b) Intruder (c) Kmeans High

(d) Kmeans Low (e) Labyrinth (f) SSCA2

(g) Vacation High (h) Vacation Low (i) Yada

Figure 14: Predictions for STAMP on *x86* using 4 checkpoints

In general, *PreSca* cannot be used to predict the changes for which there is no indication in the available measurements. Also, basing the predictions on the measurements with high variance can result in inaccurate predictions, although, as the `kmeans high` and the `kmeans low` predictions in Figure 10 illustrate, the impact is not necessarily high.

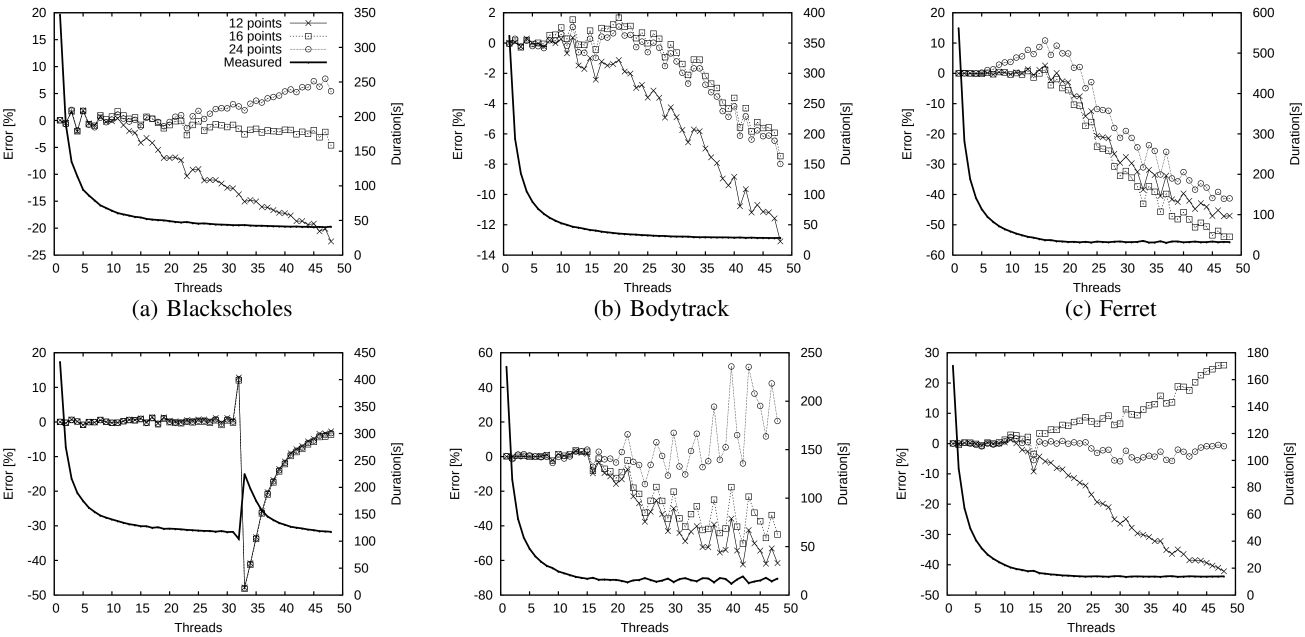
Figure 10 also shows that in some cases *PreSca*'s approach for selecting the best performance function from a set of candidates is not ideal. For example, the predictions for `intruder` based on 12 measurements are more accurate than the predictions based on more than 12 measurements. This happens because the performance function that most closely fits the measurements at the checkpoints is not the most accurate at higher thread counts. Despite this, *PreSca*'s function selection method makes the right choice in most cases.

Parsec. Figure 11 depicts *PreSca*'s prediction errors for Parsec lock-based workloads. The predictions are roughly as accurate as the STAMP predictions.

PreSca predicts the performance of `blackscholes` very accurately, with errors of less than 3.5% for all thread counts and any number of measurements used for the predictions. *PreSca* is very accurate for `raytrace` as well, with errors of less than 10% for all thread counts, except around 32 threads where the execution times suddenly increase. With $m = 12$ measurements, the errors are less than 20% with up to 21 threads for all workloads and are only marginally higher at higher thread counts for `ferret` and `x264`. Similarly, with $m = 16$ and $m = 24$ measurements, the errors are less than 20% up to $2m$ threads for all workloads except `x264`. With $m = 32$, the predictions are less than 20% off with up to 48 threads for all workloads. At higher thread counts, the prediction errors are higher for `ferret` and `x264`. Better predictions for `ferret` than the one chosen exist (e.g. predictions for $m = 16$), but *PreSca* fails to select them.

Overall, based on m measurements *PreSca* predicts the performance of the Parsec workloads with up to $2m$ threads with errors of less than 20% in 20 out of 24 cases. Furthermore, only in 2 workloads the errors are higher than 35%.

STMBench7. Figure 12 depicts the prediction errors for STMBench7 locking and STM workloads. The variance in the collected data for STMBench7 is the highest of all the used workloads, but, despite the high variance, *PreSca* is very accurate. The errors for the predictions based on measurements for m threads are lower than 20% up to $2m$ threads for all workloads and all values of m we used except for the STM-based read-write workload and $m = 16$. In this case the error is slightly higher than 20% with more than 30 threads. In other words, the predictions based on m measurements



(a) Blackscholes (b) Bodytrack (c) Ferret

(d) Raytrace (e) Vips (f) x264

Figure 15: Prediction for Parsec on *x86* using 4 checkpoints

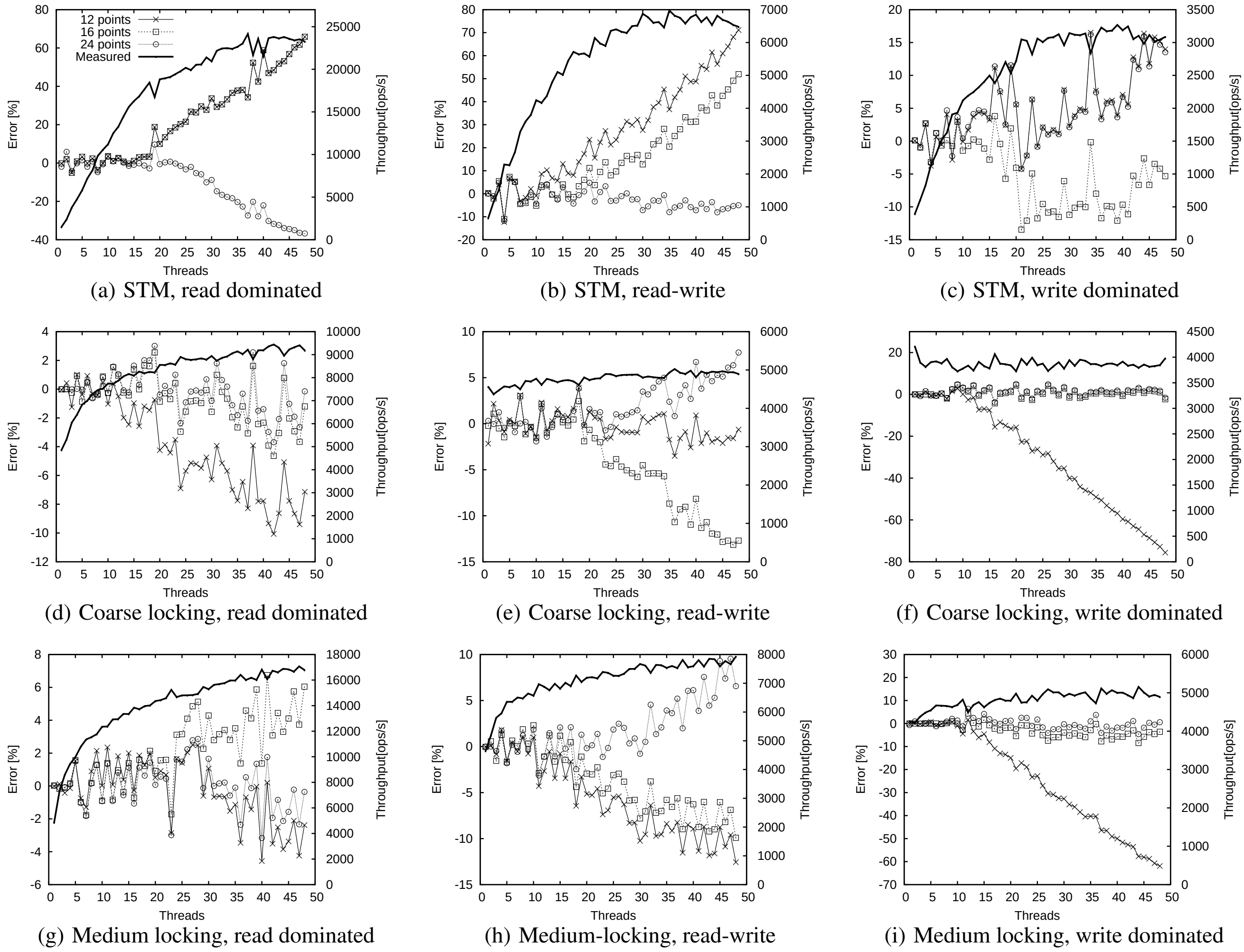
have errors lower than 20% with up to $2m$ threads in 35 out of 36 cases and they are always lower than 30%.

STM micro-benchmarks. Figure 13 depicts the prediction errors for STM micro-benchmarks. The micro-benchmarks turn out to be the hardest target for *PreSca*, mostly because their performance suddenly and significantly changes in several cases. Despite this, *PreSca* quite accurately predicts the performance of STM micro-benchmarks. With $m = 12$ measurements, the prediction errors are lower than 20% with up to 24 threads for all workloads. With $m = 16$ and $m = 24$ measurements, the prediction errors are lower than 20% with up to $2m$ threads for all workloads except `linked list low`, where the performance trends change around 24 threads. This makes it difficult for *PreSca* to accurately predict the performance using only measurements with fewer than 24 threads. With $m = 32$ measurements *PreSca* predicts the performance of the `linked list low` workload accurately. Also, with $m = 32$ measurements, the errors are lower than 20% with up to 60 threads for all workloads except `skiplist low`. The errors are only slightly higher with more than 60 threads for `linked list high` and `red-black tree low`. As the performance trends of the `skiplist low` workload change significantly at around 32 threads, the prediction errors with more than 48 threads become higher than 20%. In this workload, *PreSca* would require more measurements in order to produce more accurate predictions. Overall, the predictions based on m measurements have errors lower than 20% up to $2m$ threads in 29 out of 32 cases.

4.3.2 x86

STAMP. Figure 14 depicts the prediction errors for STAMP workloads on *x86* system. The figure shows that the predictions are roughly as accurate as the predictions for STAMP on *Sparc*. With $m = 12$, *PreSca*'s prediction errors are lower than 20% with up to 24 threads for all workloads except for `intruder`, `kmeans high` and `ssca2`. The reason for the higher errors in these cases is the significant change in the performance trends with more than 12 threads. With $m = 16$, the predictions for the `ssca2` and `intruder` workloads are more accurate and the errors are lower than 20% with up to 32 threads because the change in the performance trends is captured with the available measurements. The predictions for all other workloads except the two `kmeans` workloads are also less than 20% off. *PreSca* is less accurate for the two `kmeans` workloads because their performance significantly degrades with more than 24 threads. To accurately predict the performance of these two workloads, *PreSca* would require more than 24 measurements. With $m = 24$ measurements, the prediction errors are lower than 20% with up to 40 threads for all workloads except for the two `kmeans` workloads and `ssca2`. More accurate performance function exists for `ssca2` (e.g. the one used for performance predictions with 16 measurements), but *PreSca* fails to select it because it is not the most accurate at the checkpoints. With higher thread counts, the prediction errors for `genome` and `yada` become slightly higher than 20%, but they still remain below 30%. Overall, the predictions based on m measurements have errors lower than 20% up to $2m$ threads in 18 out of 27 cases. The errors lower than 30% in 22 cases.

Parsec. Figure 15 shows that *PreSca* produces accurate predictions for Parsec lock-based workloads on *x86* system. With $m = 12$ measurements, the prediction errors are lower than 20 with up to 24 threads for all workloads except `vips`, where they are only slightly higher with 23 and 24 threads. The reason for *PreSca*'s lower accuracy with `vips` is high variance in the measured data. With $m = 16$ and $m = 24$, the prediction errors for all workloads except `ferret` and `vips` are lower than 20% with up to $2m$ threads. Similarly to the experiments on *Sparc*, the execution times for `raytrace` suddenly increase around 32 threads, which causes higher prediction errors around 32 threads. For other thread counts, the prediction errors are below 10%. The reason for lower accuracy with `vips` is again high variance in the measured data. Overall, the predictions based on m measurements have errors lower than 20% up to $2m$ threads in 13 out of 18 cases. The



(a) STM, read dominated

(b) STM, read-write

(c) STM, write dominated

(d) Coarse locking, read dominated

(e) Coarse locking, read-write

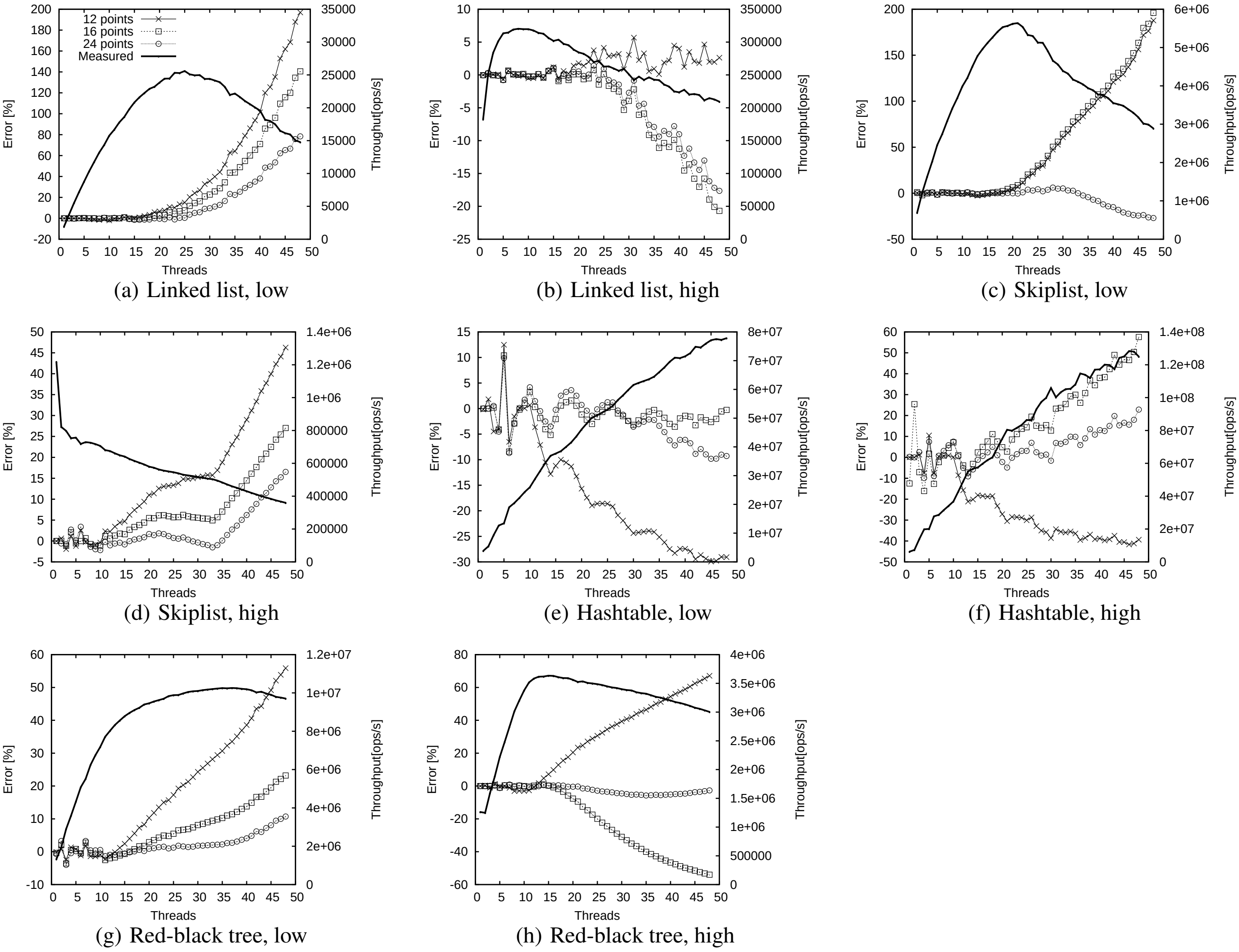
(f) Coarse locking, write dominated

(g) Medium locking, read dominated

(h) Medium-locking, read-write

(i) Medium locking, write dominated

Figure 16: Predictions for STMBench7 on *x86* using 4 checkpoints



(a) Linked list, low

(b) Linked list, high

(c) Skiplist, low

(d) Skiplist, high

(e) Hashtable, low

(f) Hashtable, high

(g) Red-black tree, low

(h) Red-black tree, high

Figure 17: Predictions for STM micro-benchmarks on *x86* using 4 checkpoints

errors are higher than 30% only in 2 cases.

STMBench7. Figure 16 depicts the prediction errors for the STM and locking STMBench7 workloads on *x86*. Despite the high variance in the measurements *PreSca* produces accurate predictions. With $m = 12$ measurements, the errors are lower than 20% with up to 24 threads for all workloads except the coarse-grained locking write-dominated workload and the STM read-write workload. In these cases the errors are slightly higher due to high variations in the measured performance. Similarly, with $m = 16$ and $m = 24$ measurements, the errors are lower than 20% with up to 48 threads for all workloads except the STM read-dominated workload where the errors are slightly higher at higher thread counts. Overall, the predictions based on m measurements have errors lower than 20% up to $2m$ threads in 23 out of 27 cases. The errors are higher than 35% only in 1 case.

STM micro-benchmarks. Figure 17 depicts the predictions for the STM micro-benchmarks on *x86*. Predicting the performance of the micro-benchmarks is very challenging, as in many cases the performance trends change suddenly without any indication of the change at the lower thread counts. Furthermore, the performance often exhibits high variation and is thus be difficult to predict. Despite this, *PreSca* is still reasonable accurate. With $m = 12$ measurements, the errors are lower than 20% with up to 24 threads for all workloads except `hashtable low` and `red-black tree high`, where the errors are only slightly higher. In these cases, the prediction errors are higher than 20% because the throughput suddenly changes around 12 threads, which *PreSca*'s cannot accurately predict based on the measurements with only up to 12 threads. With $m = 16$ measurements, *PreSca* predicts the performance of `hashtable low` more accurately as the measurements include the change in the performance trends. The errors are below 20% with up to 32 threads for all workloads except `red-black tree high` and `skiplist low`. In these cases the measurements do not capture the changes in the performance trends around 12 and 24 threads respectively. Using $m = 24$ measurements helps with the predictions for both of these workloads and *PreSca* produces the predictions with errors lower than 20% when using 24 measurments. With $m = 24$, the errors are lower than 20% for all workloads except the `linked list low`. *PreSca* requires more measurements to correctly predict the performance of `linked list low` due to the sudden drop in throughput at around 24 threads. Overall, the predictions based on m measurements have errors lower than 20% up to $2m$ threads in 19 out of 24 cases.

4.3.3 Summary

Our evaluation demonstrates that *PreSca* produces accurate extrapolations in most cases, regardless of the system architecture, workload and synchronization technique used. Overall, we used *PreSca* to perform 224 extrapolations and in 185 (or 82.5%) cases the errors were less than 20% with up to $2m$ threads. In most other cases, the errors were only slightly higher—in less than 10% the errors are higher than 35%. Inaccurate predictions are mostly the result of changes in trends in the workload's performance that are not captured in the measurements used for making the predictions and high variance in the measurements.

4.4 Using the extrapolations

We illustrate how the extrapolations presented in the previous section can be used to gain useful insight into the behavior of parallel workloads.

Predicting scalability. Maybe surprisingly, we can sometimes gain useful information about workload's scalability even with only a handful of measurements and with relatively high predictions errors. Figure 18 depicts predicted (solid line) and measured (circles) performance for several STAMP workloads based on the measurements with up to $m = 12$ threads, using the measurements for the highest 4 thread counts as the checkpoints. The plots in the figure are zoomed in where necessary to improve readability for higher thread counts.

For `intruder` on *Sparc* (Figure 18(a)) and `labyrinth` on *x86* (Figure 18(f)) the predicted performance follows the measured performance closely, with errors of less than 15% for all thread counts. Highly accurate predictions, such as these, provide great insight into the scalability of the workload. They can be used to accurately answer questions about the performance with a particular number of threads, performance improvement when using n instead of m threads or the number of threads for which the workload performs the best.

The prediction errors are higher for the other workloads in Figure 18. The predictions still follow the general shape of the measured performance providing valuable insight into trends with different number of threads, even if they cannot be used to accurately predict the execution times. For instance, the prediction error for `vacation high` on *x86* (Figure 18(c)) is up to 25%, but the predictions still allow us to make reasonably accurate estimates of the performance improvement when using 24 or 48 threads instead of 12. *PreSca* predicts that the execution time with 24 threads is 50% of the execution time with 12 threads, while the measurements show that it is actually 53%. Similarly, *PreSca* predicts that the execution time with 48 threads is 25% of the execution time with 12 threads, while it is actually 33%. In other words, *PreSca* predicts the decrease in execution time with errors of 6% and 13% respectively. Similarly, the performance function accurately predicts the good scalability of `vacation high` on *Sparc* (Figure 18(d)), but overestimates it for more than 48 threads, predicting the reduction in execution time when using 64 threads instead of 12 with 24% error.

The prediction error for `kmeans high` on *Sparc* (Figure 18(e)) varies a lot, reaching 30% at some points. Figure 18 reveals that the performance function in this case is actually quite useful, as most of the variance in the error is the result of the variance in the measurements, not in the predictions. The predictions with up to 48 threads are surprisingly accurate. *PreSca* predicts that the execution time with 32 threads is 87% of the execution time with 12 threads, while it is actually 78%. Similarly, the execution time with 48 threads is predicted to be 95% of the execution time with 12 threads, while it is really 80%.

Interestingly, even extrapolations with higher errors can provide very important insights. For example, the predictions for `intruder` on *x86* (Figure 18(f)) are off by up to 60% with higher thread counts, but *PreSca* correctly predicts that the maximum performance is achieved with around 12 threads and that using more threads negatively impacts the performance.

STM vs locking. It is sometimes interesting to speculate about the scalability of existing locking implementation and compare it to an STM alternative. We illustrate how *PreSca* can be used for this purpose by comparing STM and medium-grained locking implementations of STMBench7 on *Sparc*. The measured speedups and prediction errors are depicted in Figure 19. We calculate the speedup as $\frac{throughput_{STM}}{throughput_{lock}}$. The figure for instance shows that for the read-dominated workload, STM has lower performance than locking with a few threads, but is almost 3x faster at higher thread counts. In this case it is indeed beneficial to replace the locking with STM if additional CPU cores are available. However, the benefits of using STM instead of locking might not be completely clear when simply observing the throughput measurements with a

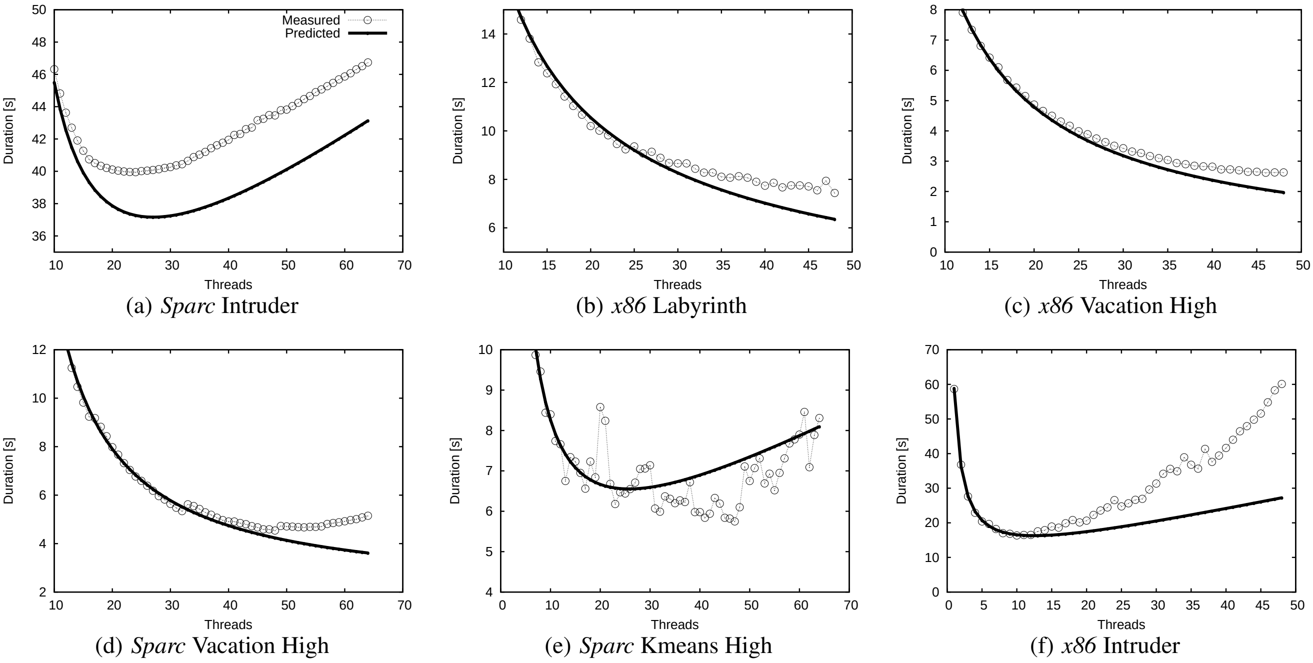


Figure 18: Measured and predicted performance (using 12 measurements with 4 checkpoints)

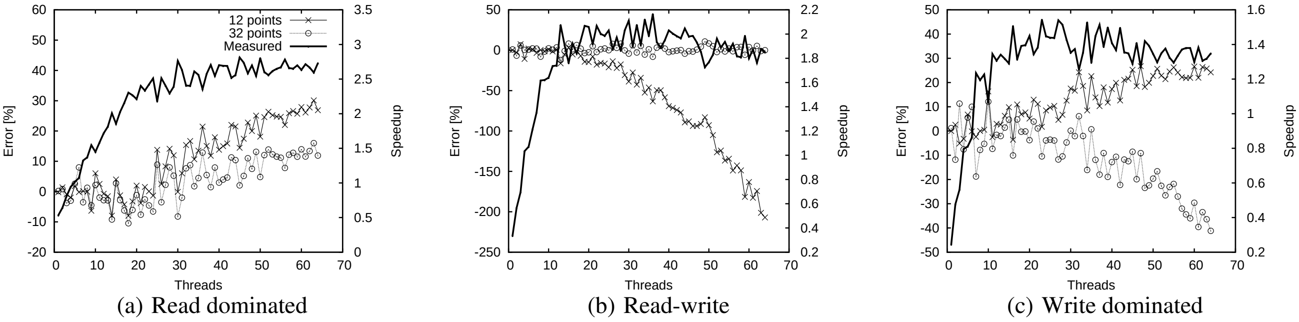


Figure 19: Predicting speedup of STM over medium-grained locking for STMBench7 on *Sparc*

few threads. We use *PreSca* to predict the performance of both locking and STM based versions of STMBench7. We compare the predicted performance to gain insight into how the two compare with higher thread counts. Figure 19 shows that we can accurately predict the speedups of STM over locking by composing two performance predictions.

4.5 OPreSca

We simulated *OPreSca* using the measurements for all workloads on both *Sparc* and *x86*. In the simulations, we assume that the workload does not change before *OPreSca* converges. We measure the accuracy of *OPreSca* (how close to the best performance *OPreSca* converges) and its convergence speed (how many intervals it takes *OPreSca* to converge).

Tables 2 and 3 contain the summary of our evaluation for both *Sparc* and *x86* measurements. We used two sets of initial thread count assignments for *OPreSca*: one random (denoted with Rnd-Rnd-Rnd in the tables) and one where we used 16, 32 and 48 thread counts (denoted with 16-32-48 in the table) for *Sparc*, and 12, 24 and 36 thread counts (denoted with 12-24-36 in the table) for *x86*. The latter two are the best initial thread assignments for *Sparc* and *x86* of the ones we tested. For the random assignments, the table contains the averages of 10 different runs. We compare *OPreSca* to a variant of the Binsearch load-picking algorithm from [28] (denoted with Binsearch in the table).

Binsearch algorithm operates in two phases. In the first phase, it increases the number of threads until it finds the point at which the performance starts to degrade. Our version of Binsearch starts from a single thread and uses an initial increment of 4. It doubles the increment at each step (i.e. it uses thread counts 1, 5, 13,...). In the second phase, Binsearch performs a binary search of the interval at which the optimal thread counts lies, until it converges to it.

In the tables we present the difference between the performance with the optimal and the chosen thread count (column E) and the number of steps before the algorithm converges (column S). We also give the average difference between the performance at each step and the best performance (column E_S) and the number of steps for which the difference in the performance is higher than 10% (column $S_{E>10\%}$) to illustrate the overall cost of the algorithm.

The results show that *OPreSca* converges very closely to the op-

	Rnd-Rnd-Rnd				16-32-48				Binsearch			
	E	S	E_S	$S_{E>10\%}$	E	S	E_S	$S_{E>10\%}$	E	S	E_S	$S_{E>10\%}$
Genome	0.01	5.4	0.23	2.3	0	9	0.15	2	0	12	0.18	4
Intruder	0	6.7	0.08	1.4	0	4	0.03	0	0	12	0.13	3
Kmeans High	0.03	8.8	0.24	5.4	0.04	6	0.24	4	0.12	10	0.29	10
Kmeans Low	0.01	7.9	0.3	4.7	0.01	7	0.14	3	0	8	0.32	4
Labyrinth	0.01	8.4	0.16	2	0.01	4	0.07	1	0	13	0.17	3
SSCA2	0.12	7.5	0.39	5.5	0	8	0.27	4	0	12	0.35	8
Vacation High	0.02	8.4	0.25	4.1	0	6	0.13	3	0	13	0.24	6
Vacation Low	0.03	8.1	0.26	3.8	0	9	0.19	4	0	13	0.25	6
Yada	0.01	7.9	0.19	3.6	0.01	5	0.13	2	0.01	12	0.21	7
Blackscholes	0.01	4.4	0.12	1.3	0	6	0.25	3	0	7	0.22	3
x264	0	5.5	0.18	1.9	0	4	0.07	1	0	8	0.25	3
Bodytrack	0	6.8	0.14	2	0	4	0.07	1	0	13	0.15	3
Ferret	0.01	8	0.13	1.3	0	11	0.1	1	0	14	0.14	3
Raytrace	0.04	6.7	0.11	2.6	0	10	0.17	4	0	7	0.15	3
Vips	0.01	5.1	0.27	2.7	0.01	4	0.22	2	0	12	0.25	5
SB7 Coarse R	0.07	7.6	0.29	2.8	0.06	4	0.12	1	0.03	12	0.27	6
SB7 Coarse RW	0.03	8.5	0.14	5	0.03	7	0.12	5	0.02	11	0.12	8
SB7 Coarse W	0.06	7.6	0.13	3	0.03	9	0.11	4	0.21	2	0.22	0
SB7 Medium R	0.02	6.4	0.3	3.1	0.06	11	0.38	3	0	12	0.35	8
SB7 Medium RW	0.04	5.6	0.18	1.7	0.03	10	0.11	5	0.02	7	0.27	3
SB7 Medium W	0.03	9.4	0.13	4.8	0.03	8	0.17	2	0.19	8	0.26	0
SB7 STM R	0	5.7	1.04	2.4	0	4	0.16	2	0	12	1.83	8
SB7 STM RW	0.04	6.9	0.44	3.8	0.05	8	1.33	5	0.02	7	1.57	3
SB7 STM W	0.02	8.3	0.53	5.5	0.02	10	0.73	6	0.02	14	0.6	10
Hashtable High	0.01	4	2.32	1.4	0.01	7	0.63	1	0	7	5.96	3
Hashtable Low	0.01	4	1.76	1.4	0.01	5	1.49	1	0	7	6.62	3
Linked list High	0.05	7.9	0.37	2.6	0.04	11	0.29	4	0	8	0.13	6
Linked list Low	0	6.8	1.85	3.3	0	5	0.13	3	0	13	1.31	7
RB tree High	0	5.1	1.41	3.1	0	4	0.28	2	0.01	11	2.21	7
RB tree Low	0	4	1.95	2.1	0	4	0.39	2	0	7	5.71	3
Skiplist High	0	6.5	1.12	4.4	0	7	2.05	5	0	13	1.33	9
Skiplist Low	0.01	6.8	2.09	4.3	0	9	3.11	6	0	13	2.56	9
Average	0.02	6.77	0.6	1.39	0.01	6.87	0.43	2.87	0.02	10.31	1.08	5.125

Table 2: Convergence speed and error of *OPreSca* and Binsearchfor *Sparc* workloads. Column E represents the final error compared to the optimal thread count. S represents the total number of steps before the algorithm converges. E_S represents the average step error. $S_{E>10\%}$ represents the number of steps with errors of more than 10%.

timal thread count with the average error of less than 2.5%. It does so in less than 7 steps on average for both initial thread count assignments. On average *OPreSca* impacts the performance by more than 10% in fewer than 3 intervals on *Sparc* and fewer than 4 intervals on *x86*. Using the carefully chosen initial thread count assignment instead of a random one slightly improves *OPreSca*'s performance. Most notably, the total cost of using *OPreSca* is about 30% lower.

OPreSca is roughly as accurate as Binsearch, but it is about 35% faster. Also, the average per-step performance impact is smaller, which means that the total cost of using *OPreSca* is smaller than the cost of Binsearch. Overall, *OPreSca* with random initial thread counts incurs 2.5x smaller cost than Binsearch on *Sparc* and 1.4x on *x86*. Similarly, using *OPreSca* with the best fixed initial thread counts is more than 3.5x less expensive than using Binsearch on *Sparc* and about 2x on *x86*.

5. RELATED WORK

Scientific applications models. Several papers propose performance models for highly scalable scientific applications on large-scale machines, which typically address the question of how the performance changes depending on the applications' inputs. The models treat the number and the configuration of CPUs as one of the inputs. In contrast, *PreSca* considers a much wider range of workloads, typically with lower scalability. The application's inputs are considered a part of the workload and are not modeled explicitly.

The idea of predicting the scalability of the scientific applications using function approximation with linear logarithmic functions was explored in [3]. The use of linear logarithmic functions gave good results because the used workloads exhibit very good scalability, unlike some of the workloads *PreSca* targets. In contrast, *PreSca* is more general and therefore it uses a variety of function types, and selects the best function type based on the workload.

In [18], the authors use statistical techniques to build piecewise polynomial and neural network models of scientific programs. Unlike *PreSca*, the models do not address the question of application's scalability when increasing the number of CPUs beyond what is used during the measurements.

To predict the performance of the scientific applications, several papers combine the predictions of sequential performance of the tasks that nodes perform and the model of communication between the nodes. Phantom [31] uses a deterministic replay of single-node tasks and trace-driven network simulation. In [5] a convolution model is used to map the application signature to the machine characteristics in order to predict the performance of the individual tasks. The toolkit described in [19] semi-automatically models application characteristics in architecture-neutral way to predict the performance and low-level CPU events (such as cache and TLB misses) of the sequential task's executions. The toolkit uses a combination of static and dynamic analysis of program binaries to produce much more detailed models than *PreSca*, which it uses to predict the performance of the applications on different machines from the ones used during the dynamic analysis.

	Rnd-Rnd-Rnd				12-24-36				Binsearch			
	E	S	E_S	$S_{E>10\%}$	E	S	E_S	$S_{E>10\%}$	E	S	E_S	$S_{E>10\%}$
Genome	0.01	6.8	0.19	3.2	0	5	0.13	2	0	12	0.21	5
Intruder	0	8.1	0.29	5.1	0.01	6	0.29	3	0	10	0.19	5
Kmeans High	0.03	7.5	0.28	5.1	0.03	10	0.23	5	0.03	10	0.23	6
Kmeans Low	0.09	7.6	0.41	6.4	0.07	9	0.39	8	0	11	0.25	7
Labyrinth	0.04	7.1	0.3	3.7	0.08	6	0.28	3	0	9	0.27	4
SSCA2	0	7.7	0.13	2.2	0	8	0.12	1	0	11	0.11	2
Vacation High	0	5	0.28	2.8	0	8	0.33	5	0	10	0.28	4
Vacation Low	0.01	6.2	0.33	3.2	0.01	5	0.23	2	0	12	0.25	6
Yada	0.01	7.3	0.23	3.5	0	9	0.2	4	0	12	0.21	5
Blackscholes	0	4.6	0.22	2.4	0	4	0.12	2	0	10	0.19	3
x264	0.01	8.7	0.15	2.6	0.02	4	0.09	1	0.01	11	0.19 3	
Bodytrack	0	4.9	0.15	2.1	0	4	0.15	2	0	10	0.21	3
Ferret	0.03	9	0.21	3.8	0.03	5	0.13	2	0.05	11	0.24	5
Raytrace	0.11	7.1	0.26	7	0.11	8	0.25	8	0.11	9	0.25	9
Vips	0.09	7.5	0.35	6.9	0.15	6	0.4	6	0.11	10	0.39	10
SB7 Coarse R	0	4.6	0.1	2.8	0	7	0.1	5	0	11	0.16	8
SB7 Coarse RW	0.01	8.4	0.05	7.5	0	9	0.04	8	0.1	2	0.1	0
SB7 Coarse W	0.02	6.7	0.08	5.1	0	7	0.08	5	0	2	0.04	2
SB7 Medium R	0.03	4.2	0.3	1.9	0	7	0.44	4	0.01	11	0.35	7
SB7 Medium RW	0.02	8	0.15	5.2	0.03	4	0.08	3	0	9	0.14	6
SB7 Medium W	0.03	7.1	0.09	5	0.03	9	0.08	7	0	12	0.06	9
SB7 STM R	0.02	6.2	2.42	2.8	0.01	8	2.03	5	0.01	11	1.75	7
SB7 STM RW	0.02	8.5	1.09	5.7	0	7	1.78	4	0.02	12	1.05	9
SB7 STM W	0.13	7.2	0.86	4.1	0.01	6	0.36	4	0	13	0.75	8
Hashtable High	0.01	5.3	2.04	1.7	0	4	0.57	1	0	10	2.69	6
Hashtable Low	0.01	4.4	1.63	2.1	0.01	4	0.53	1	0	9	2.86	5
Linked list High	0	7.5	0.23	3.7	0	9	0.16	5	0	8	0.1	7
Linked list Low	0.07	6.7	1.5	3	0.01	4	0.16	2	0	12	1.32	7
RB tree High	0	7.3	0.23	4.9	0	6	0.34	4	0	10	0.28	7
RB tree Low	0	7.1	0.77	4.9	0	8	1.06	5	0	10	0.84	7
Skiplist High	0	4.2	0.83	1	0	4	0.9	1	0	2	0.31	1
Skiplist Low	0.01	6.7	1.24	2.7	0	8	1.16	4	0	11	0.83	7
Average	0.03	6.73	0.54	3.88	0.02	6.5	0.41	3.81	0.01	9.78	0.53	5.63

Table 3: Convergence speed and error of *OPreSca* and Binsearchfor *x86* workloads. Column E represents the final error compared to the optimal thread count. S represents the total number of steps before the algorithm converges. E_S represents the average step error. $S_{E>10\%}$ represents the number of steps with errors of more than 10%.

PACE [22] proposes an object-oriented language for describing performance characteristics of various system's components. A performance analysis and prediction framework that aims at predicting the performance of message-passing high performance applications is built around the language.

Detailed application models. Various formal modeling techniques for distributed and concurrent systems have been proposed [14], including Petri nets [23] and Queueing theory [17]. These techniques can be used to predict the performance and the scalability of parallel workloads, they are well-known and widely applicable. However, they require the users to create detailed models of the system, which is not always feasible or practical. They were used to develop detailed analytical models for several applications [13, 16]. These models are very accurate, but they require in-depth understanding of the applications and the system. In contrast, *PreSca* models the performance of the application without using any information about its internals.

STM models. Models based on discrete-time Markov chains were developed for several STM algorithmic design choices [10–12]. The models were used to compare choices using the aggregate transactional characteristics. This approach uses a much more complex model and more information about the executions than *PreSca*. The performance model from [25] focuses on modeling transactional conflict behavior. Unlike *PreSca*, it requires heavy instrumentation of the applications in order to collect the statistics about the memory accesses and conflict patterns. The focus of the model is on the conflict behavior and it mostly overlooks the other effects that also affect the performance (e.g. increased rate of cache misses). In contrast, our model describes all of the system's components using a simple function. A simple cost-benefit analysis that is used to choose locking or transactions is described in [29]. This analysis addresses a more limited question than *PreSca*.

6. CONCLUDING REMARKS

We presented *PreSca*, a pragmatic system for predicting the scalability of parallel applications. Despite its simplicity, *PreSca* provides accurate predictions, as conveyed by our extensive evaluation. We demonstrated how *PreSca* can be used to gain insight into the performance of parallel workloads, including the comparison of STM and locking versions of the same application, as well as how it could be deployed on-line to dynamically assign the optimal number of threads to a running application.

As we pointed out earlier in the paper, *PreSca* is by no means argued to be a silver bullet in determining the scalability of parallel applications in all contexts. We discuss in the following some of its advantages and limitations.

Indeed, *PreSca* can be applied to any parallel application, regardless of the synchronization technique it uses. Furthermore, *PreSca* does not use any knowledge of the workload or the computer system, which makes it very general and easy to use. In particular, *PreSca* does not require any instrumentation, which eliminates any side-effect commonly associated with profiling. It only uses some measure of performance, which needs to be provided by the application in any case, if we want to reason about its performance. The performance of the system can be measured in different ways. For example, it could be expressed as the time the application needs to execute some task, the number of committed transactions per second or the speedup over the sequential, single-threaded code. *PreSca* can be applied to any of these, because function approximation can be applied to any analytical function.

PreSca can help developers choose the synchronization technique for the application at the start of the application development. Naturally, the inclination to use STM can be very high, as using STM is simple. However, STM's performance can be disappointing which can lead the developers to choose other, more involved, techniques, such as locking or lock-free techniques. *PreSca* enables the developers to gain insight into the STM-based prototypes of the application. The prototype is relatively simple to produce—it requires defining atomic sections in the sequential code, which is necessary regardless of the synchronization technique used, so it does not pose additional burden to the developers. Then, if *PreSca*'s predictions of the performance are not satisfactory, the application can be improved by either using finer-grained transactions, or a completely different synchronization control technique. If the predictions are satisfactory, the developers can use STM to develop the application with some confidence that it will perform well. Such trial-and-error approach is not very well suited for the other concurrency control techniques because they require much more time and effort to produce the prototype.

Because *PreSca* models the whole workload as a black-box, the predictions are closely tied to the workload. Changing any underlying component (e.g. the number or the configuration of CPUs, or the synchronization technique) can significantly reduce the applicability, or even render useless, the performance function constructed for the old system. Despite the high sensitivity to the changes of the system, our experiments show that the performance function remains reasonably accurate in some very important cases. In particular, the performance function constructed using threads running on only a subset of CPUs in a machine, can accurately predict the performance when threads running on other CPUs in the same machine are used.

Also, the ability of *PreSca* to predict the performance for workloads with inherently highly varying performance is limited. In some cases, the performance varies so significantly between executions with the same number of threads that it truly cannot be predicted with any approach. Even when variations are less significant, like in `kmeans low` and `kmeans high` from STAMP (Figures 10 and 14), they can mislead the prediction and steer it in a wrong direction. This reduces the applicability of *PreSca* in cases when measurements are noisy, but it does not make it useless.

PreSca does not provide any insight about why the system performs the way it does even when the predictions are accurate. This impacts the usability of *PreSca* as a performance debugging tool. *PreSca* can still be used during performance debugging, but only to predict the impact of already implemented modifications and not to guide future optimizations.

References

[1] N. I. Akhiezer. *Theory of Approximation*. Dover Publications, June 1992.

[2] Anonymous. A pragmatic approach for predicting scalability of parallel applications, 2011. Technical report.

[3] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 368–377, New York, NY, USA, 2008. ACM.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[5] L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22:336–346, February 2006.

[6] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *ACM Queue*, 6:46–58, September 2008.

[7] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why stm can be more than a research toy. *Communications of the ACM*, 54:70–77, April 2011.

[8] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[9] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.

[10] A. Heindl and G. Pokam. An analytic framework for performance modeling of software transactional memory. *Computer Networks*, 53:1202–1214, June 2009.

[11] A. Heindl and G. Pokam. An analytic model for optimistic stm with lazy locking. In *Proceedings of the 16th International Conference on Analytical and Stochastic Modeling Techniques and Applications*, ASMTA '09, pages 339–353, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] A. Heindl, G. Pokam, and A.-R. Adl-Tabatabai. An analytic model of optimistic software transactional memory. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '09, pages 153 –162, 2009.

[13] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *International Journal of High Performance Computing Applications*, 14:330–346, November 2000.

[14] R. Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley, 1991.

[15] W. Karl. Uber die analytische darstellbarkeit sogenannter willk§rlicher functionen einer reellen veränderlichen. 1885.

[16] D. J. Kerbyson, A. H. J., A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, Supercomputing '01, pages 37–37, New York, NY, USA, 2001. ACM.

[17] E. Lazowska, Z. John, G. Scott, and S. Kenneth. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[18] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles*

and practice of parallel programming, PPoPP '07, pages 249–258, New York, NY, USA, 2007. ACM.

[19] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, pages 2–13, New York, NY, USA, 2004. ACM.

[20] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, IISWC '08, pages 35 –46, 2008.

[21] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, January 1965.

[22] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace–a toolset for the performance prediction of parallel and distributed systems. *International Journal of High Performance Computing Applications*, 14:228–251, August 2000.

[23] C. A. Petri. *Communication with automata*. DTIC Research Report AD0630125, 1966.

[24] J. R. Phillips. Zunzun.com online curve fitting and surface fitting. `http://www.zunzun.com`.

[25] D. Porter and E. Witchel. Understanding transactional memory performance. In *2010 IEEE International Symposium on Performance Analysis of Systems Software*, ISPASS '10, pages 97 –108, 2010.

[26] SciPy. Scientific tools for python. `http://www.scipy.org`.

[27] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[28] P. Shivam, V. Marupadi, J. Chase, T. Subramaniam, and S. Babu. Cutting corners: workbench automation for server benchmarking. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, USENIX '08, pages 241–254, 2008.

[29] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 3–14, Washington, DC, USA, 2009. IEEE Computer Society.

[30] N. Vasic, D. Novakovic, S. Miucin, D. Kostic, and R. Bianchini. DejaVu: Accelerating Resource Allocation in Virtualized Environments. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, 2012.

[31] J. Zhai, W. Chen, and W. Zheng. PHANTOM: predicting performance of parallel applications on large-scale parallel machines using a single node. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 305–314, New York, NY, USA, 2010. ACM.