ESTIMA: Extrapolating ScalabiliTy of In-Memory Applications

GEORGIOS CHATZOPOULOS, École Polytechnique Fédérale de Lausanne
ALEKSANDAR DRAGOJEVIĆ, Microsoft Research
RACHID GUERRAOUI, École Polytechnique Fédérale de Lausanne

This article presents ESTIMA, an easy-to-use tool for extrapolating the scalability of in-memory applications. ESTIMA is designed to perform a simple yet important task: Given the performance of an application on a small machine with a handful of cores, ESTIMA extrapolates its scalability to a larger machine with more cores, while requiring minimum input from the user. The key idea underlying ESTIMA is the use of stalled cycles (e.g., cycles that the processor spends waiting for missed cache line fetches or busy locks). ESTIMA measures stalled cycles on a few cores and extrapolates them to more cores, estimating the amount of *waiting* in the system. ESTIMA can be effectively used to predict the scalability of in-memory applications for bigger execution machines. For instance, using measurements of `memcached` and `SQLite` on a desktop machine, we obtain accurate predictions of their scalability on a server. Our extensive evaluation shows the effectiveness of ESTIMA on a large number of in-memory benchmarks.

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**; **Concurrent algorithms**; • **Computer systems organization** → **Multicore architectures**;

Additional Key Words and Phrases: ESTIMA, scalability, extrapolation, prediction, in-memory, stalled cycles

ACM Reference format:
Georgios Chatzopoulos, Aleksandar Dragojević, and Rachid Guerraoui. 2017. ESTIMA: Extrapolating ScalabiliTy of In-Memory Applications. *ACM Trans. Parallel Comput.* 4, 2, Article 10 (August 2017), 28 pages.
https://doi.org/10.1145/3108137

1 INTRODUCTION

Commodity machines nowadays have hundreds of gigabytes of memory. This enables building performance-critical parallel applications, such as databases and key-value stores, that keep their datasets in main memory. This way, applications avoid slow secondary storage and networks, leaving the CPU as the main performance bottleneck [9, 12, 26, 30]. Understanding the performance of these applications proves to be hard, since the number of CPU cores available during the deployment of a parallel application can be significantly higher than that during its development and testing. Applications developed today can be tested on machines with 16 or 24 cores, but in a few years the same applications are likely to be run on machines with 64 or even more cores.

Part of this work was supported by the European Research Council (ERC) Grant 339539 (AOC).
Authors' addresses: G. Chatzopoulos and R. Guerraoui, EPFL IC IINFCOM LPD, Station 14, 1015 Lausanne, Vaud, Switzerland; emails: {georgios.chatzopoulos, rachid.guerraoui}@epfl.ch; A. Dragojević, Microsoft Research UK Ltd, 21 Station Road, Cambridge, CB1 2FB, UK; email: alekd@microsoft.com.

ACM 2329-4949/2017/08-ART10 $15.00
https://doi.org/10.1145/3108137

Consequently, the crucial question about performance of in-memory applications is that of their scalability on an increasing number of cores. Answering this question is very hard. Typical approaches include performing extensive performance evaluation or developing detailed models of applications [15, 28], which are time-consuming, error-prone, and require detailed knowledge of each application and the machine it executes on.

This article presents ESTIMA, a practical tool that enables developers and users to predict the scalability of parallel in-memory applications in a simple way, without having to understand in detail the internals of the application or the machine it will run on. ESTIMA enables developers to visualize the scalability of their applications, as well as to discover bottlenecks that might not be evident during initial performance benchmarking. ESTIMA can be applied with little effort to *any* parallel in-memory application, in contrast to other approaches that heavily rely on application-specific information [4, 6, 22, 24, 25, 31, 46].

Instead, ESTIMA leverages *stalled cycles* to extrapolate the scalability of an application. These are cycles the application spends on non-useful work, such as waiting for a cache line to be fetched from memory or waiting on a busy lock. Contention for shared resources typically increases with the number of cores used by an application, resulting in an increase in stalled cycles that directly impact the application's scalability. The application's performance keeps improving as long as adding cores mainly increases the number of useful cycles. As soon as adding more cores mostly results in stalls, performance stops improving, or even degrades: The application stops scaling.

ESTIMA measures stalled cycles in both hardware and software and extrapolates them (using analytic functions) to higher core counts to predict the overheads of using more cores. Then, ESTIMA correlates stalls to execution time to produce predictions of the execution time of the application at higher core counts. In addition to predicting scalability, analyzing the dominating stalled cycle categories reported by ESTIMA can reveal bottlenecks that will appear for higher core counts and guide developers' optimization efforts. To the best of our knowledge, ESTIMA is the first system to use stalled cycles for scalability extrapolations.

By default, ESTIMA uses hardware performance counters to measure hardware stalls. These are counters offered by modern hardware that can collect the values of events that stall the execution of an application with low overhead. Measuring software stalls requires configuring or instrumenting runtime libraries, such as `pthreads` or a transactional memory library. In our experience, software stalls can be exposed with minimal changes to the runtime libraries, but because they are not always available, ESTIMA does not require software stalled cycles to function.

Our evaluation shows that ESTIMA's simple approach yields accurate predictions. We illustrate the use of ESTIMA to successfully predict the performance of a `memcached` and a `SQLite` workload on a server machine based on measurements on a desktop. We then extensively evaluate ESTIMA using 21 workloads that span a wide range of application characteristics and synchronization techniques on two different platforms: a 4-socket, 48-core AMD *Opteron* machine and a 2-socket, 20-core Intel *Xeon* machine. We conduct both strong scaling and weak scaling experiments. Finally, we pick two applications that exhibit poor scalability (`streamcluster` and `intruder` from our benchmark workloads) and show how stalled cycles can help identify bottlenecks. More specifically:

- ESTIMA accurately extrapolates scalability between different machines with similar architectures on real-world workloads. For our `memcached` and `SQLite` workloads measured on a desktop machine, ESTIMA predicts their scalability on a server machine with errors lower than 30% and 26%, respectively.
- ESTIMA successfully captures the scalability of all applications we consider for its evaluation, correctly identifying the number of cores for which the applications stop scaling, for both strong and weak scaling predictions. The predictions are fairly accurate in absolute terms,

too. ESTIMA can predict performance when doubling the number of cores with errors lower than 15% on more than half of the workloads.
- Prediction errors do not result in prediction of a different behavior. In other words, there are no cases where ESTIMA predicts that an application will scale, when in fact it does not.
- ESTIMA can help identify bottlenecks as we illustrate through two parallel applications that exhibit poor scalability, `intruder` and `streamcluster` from the STAMP and PARSEC benchmark suites, respectively.

In summary, the main contribution of this work is ESTIMA, a practical tool that uses stalled cycles from both hardware and software sources to extrapolate the scalability of in-memory applications. ESTIMA can help users make better decisions regarding the deployment of their application. To the best of our knowledge, ESTIMA is the first system to use stalled cycles for scalability extrapolations. It showcases the power of stalled cycles in extrapolating the scalability of an application, while being applicable to a wider range of application than existing tools and methodologies.

The rest of this article is organized as follows. We present the insights behind ESTIMA in Section 2. We explain how ESTIMA works in Section 3. We present the implementation, as well as our extensive evaluation of ESTIMA in Section 4. We discuss some interesting findings of our work in Section 5. We present related work in Section 6 and conclude this article in Section 7.

2 ESTIMA: INSIGHTS

In this section, we present the insights behind ESTIMA. We first recall what stalled cycles are, their main sources, and how they affect scalability. We discuss why we use stalled cycles in ESTIMA, in contrast to the straightforward approach of extrapolating time. Finally, we present some of the decisions we have made when designing ESTIMA and how they affect its capabilities.

2.1 Stalled Cycles

During the execution of an application, CPU cycles are spent to produce useful work, while part of the cycles is spent stalling (called *stalled cycles*), either at the hardware level (i.e., waiting on a cache line miss) or at the software level (i.e., spinning on a busy lock). In an ideal scenario, stalled cycles would not constitute a significant part of the execution time and cycles that produce useful work would be evenly distributed across processors, resulting in linear speedup for the application (assuming that the instructions executed do not change significantly when running on more cores).

However, this is rarely the case. Stalled cycles can comprise a significant part of execution time, increasing with the number of cores. Stalled cycles are present both in hardware and software. At the hardware level, stalls are the result of the unavailability of processing units or data, which typically degrades the performance of an application as the number of cores increases. What further aggravates the problem is that parallel applications need synchronization, which causes further increases in stalled cycles at the software level. These stalls minimize the benefits from scaling up an application and could be the reason behind even slowdown for higher core counts.

2.2 Hardware Stalled Cycles

Measuring hardware stalls is at the core of ESTIMA. These can be divided into two big categories, depending on the stalled execution stage. *Frontend stalls* are stalled cycles in the fetch and decoding phase of an instruction execution, while *backend stalls* are stalled cycles during the rest of the execution of an instruction. Frontend stalls can typically be attributed to waiting on an instruction fetch that missed in the instruction cache or a target fetch after a branch misprediction. Backend stalls are typically the result of resources or data not being available during execution. Both categories of stalled cycles have a negative effect on the performance of an application. However,

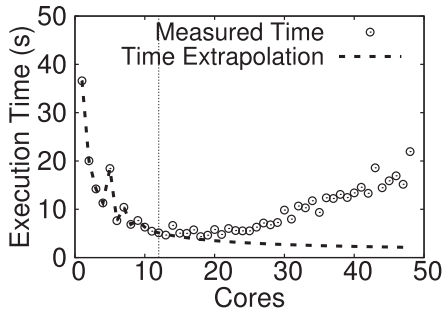


Fig. 1. Time extrapolation for kmeans.

frontend stalled cycles do not change significantly for increasing core counts. In contrast, backend stalled cycles have a direct impact on the scalability of an application, as they significantly increase when adding more cores. We have found no benefits in prediction accuracy when using frontend stalls. For this reason, and given that modern processors can measure up to four events concurrently, ESTIMA uses only backend stalls at the hardware level.

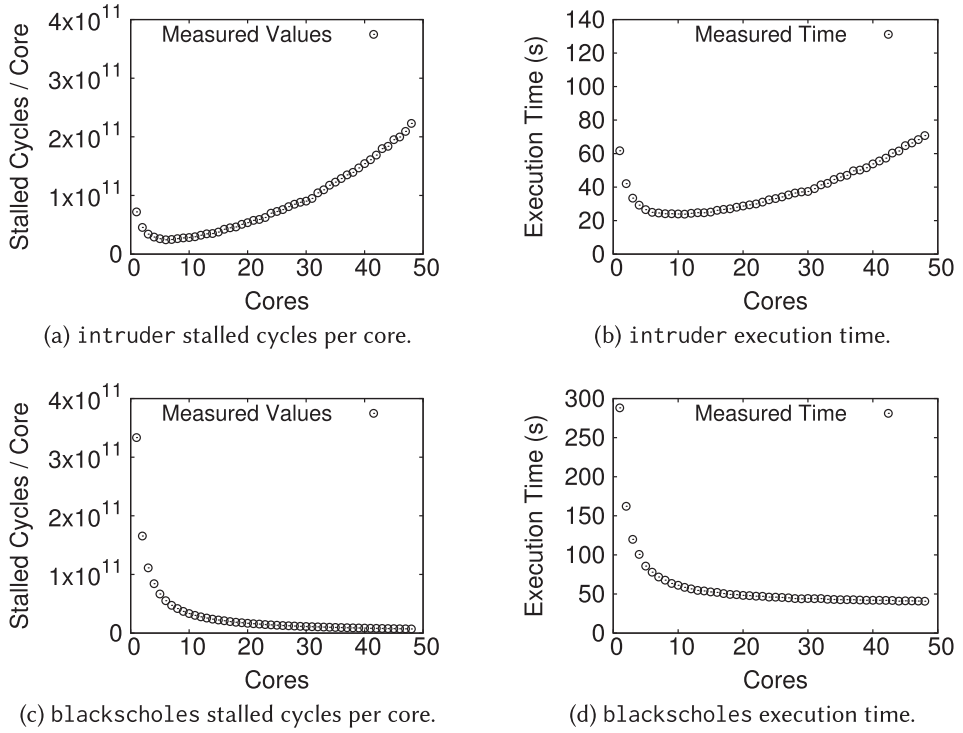
2.3 Software Stalled Cycles

Previous research has shown that Instructions-Per-Cycle (IPC) related metrics are not always useful in predicting the performance of an application [2]. Stalled cycles are closely related to IPC calculations and can thus face the same problem. A processor can spend time executing instructions that do not contribute to useful work but are also not considered stalled cycles at the hardware level. If only hardware stalls are considered, then this choice can be the source of prediction inaccuracies. Stalls at the software level can typically be found in synchronization of threads. This includes both lock-based synchronization (e.g., spinning on a busy lock), as well as optimistic concurrency control mechanisms, such as *software transactional memory (STM)* [18, 37]. In applications that use STM libraries, aborted transactions discard all work done inside the transaction.

ESTIMA solves this problem by enabling the use of *software stalled cycles*. These represent cycles during which the application is executing instructions that do not produce useful work. Software stalls are optional: users can use a runtime that reports software-level stalled cycles, or modify their applications to provide such information, to improve the accuracy of predictions.

2.4 Extrapolating Time

A straightforward approach for scalability predictions is to extrapolate the execution time of an application, measured for low core counts. Indeed, such approaches already exist [4] and provide high accuracy for the workloads they target. They typically function as follows: Initially, they take measurements of the execution time of the application for different core counts. The next step is to use analytic functions to approximate the measurements, and extrapolate the measurements to higher core counts. An important drawback of this approach is that extrapolation requires scalability trends to be present in the existing measurements. When that is not the case, such as in the case of `kmeans`, shown in Figure 1, directly extrapolating time can lead to erroneous conclusions. In this case, the time extrapolation method predicts that the application will continue scaling for up to 48 cores, which is not the case. Similar cases can appear when small changes in the execution time steer the extrapolation towards wrong predictions. In our evaluation (Section 4), we show how ESTIMA compares against time extrapolation, as well as cases where ESTIMA is able to capture the scalability of applications for which trends are not visible in measurements.



(a) `intruder` stalled cycles per core.

(b) `intruder` execution time.

(c) `blackscholes` stalled cycles per core.

(d) `blackscholes` execution time.

Fig. 2. Stalled cycles and execution time correlation.

2.5 Stalled Cycles for Scalability Predictions

ESTIMA uses the number of stalled cycles per core to predict the scalability of an application as the number of cores increases. We show two examples of applications, presenting the stalled cycles per core and the execution time in Figure 2. They are the `intruder` and `blackscholes` benchmarks from the STAMP [29] and PARSEC [5] suites, respectively. For both applications, the correlation of the number of stalled cycles per core to execution time is 1.00. We evaluate the correlation of stalled cycles to execution time more extensively in Section 5.1.

ESTIMA leverages stalled cycles for its predictions. By default, ESTIMA uses *hardware performance counters*, dedicated CPU registers, used to monitor performance events, such as the number of instructions executed, cache misses per cache level, stalled cycles, as well as I/O requests and memory accesses. Different architectures offer various performance counter events that measure a wide range of backend stalled cycles. There usually exist aggregate events that measure the total backend hardware stalled cycles, as well as more detailed events that measure different types of backend stalled cycles individually [3, 20]. ESTIMA does not use the aggregate events. Instead, it uses the performance counters that measure fine-grain backend stalled cycles on each architecture. These counters are low-level enough to provide insights into the behavior of the application for higher core counts. In addition, when combined, they give the same high-level image of the scalability of the application as aggregate events.

The insight behind using fine-grain stall events is the following: Using an aggregate event would be similar to extrapolating the execution time itself, since the two follow the same trends. For example, from the aggregate backend stalls shown in Figure 2, with measurements up to 12 cores, we do not see trends that show the poor scalability of the applications for higher core counts.

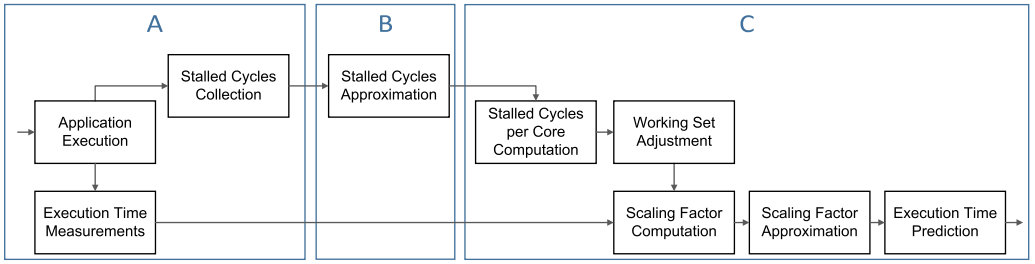


Fig. 3. Constructing extrapolations with ESTIMA.

Using aggregate events would not capture significant changes in the scalability of applications, which is the main goal of ESTIMA. By using fine-grained stalls, trends appear in their values for lower core counts, before the effect on the scalability of the application is significant. These trends are helpful in predicting scalability changes that are otherwise difficult to capture (we give more details in the prediction example in Section 3.2). A second reason for using the individual stalled cycles is that aggregate events do not provide any information on the scalability bottlenecks. By using the detailed events and pinpointing the parts of the code they come from, ESTIMA can help identify the area that inhibits scalability, as we show in Section 4.6.

2.6 Other Performance Counters

Prior work [43] has used performance counters to measure events such as cache misses and branch mispredictions to identify bottlenecks in applications. A similar approach in ESTIMA would involve extrapolating counters such as cache misses for the different levels of cache and incorporating them in the prediction process. The problem with this approach is that cache misses require a very detailed model that takes into account the memory access patterns of the application. This is necessary to translate the misses captured to time and quantify their effect to the scalability of the application. Quantifying the interactions between misses and scalability requires detailed knowledge of the application, which is against the generic purpose of ESTIMA. For this reason, we chose not to use cache misses in our predictions. However, their effect is captured by the stalled cycles that ESTIMA uses. In this case, their actual effect on scalability is captured through its manifestation in stalls in the pipeline.

3 ESTIMA

The prediction scheme of ESTIMA is depicted in Figure 3. It involves three main steps: (A) first, ESTIMA executes the application on the *measurements machine*, collecting different types of stalled cycles from hardware and optionally from software. (B) Then, ESTIMA extrapolates the values of these stalls to higher core counts, using regression analysis and a set of pre-defined function kernels. (C) Finally, ESTIMA combines the extrapolated values and calculates the stalled cycles per core. By correlating stalled cycles to execution time, ESTIMA predicts the execution time of the application for higher core counts. In the next sections, we provide a detailed description of the internals of this process and present a step-by-step execution example of ESTIMA.

3.1 Prediction Process

3.1.1 Stalled Cycles Collection. The first step of the prediction process of ESTIMA is to execute the application for different core counts, up to the number of cores available on the measurements machine, collecting hardware performance counters and software-reported stalls. During the

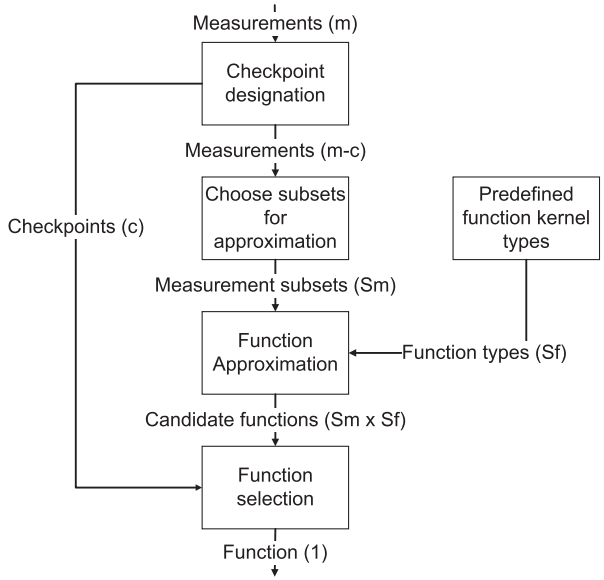


Fig. 4. The regression analysis of ESTIMA.

execution of the application, ESTIMA also measures the application's execution time. ESTIMA uses these measurements for the execution time predictions in the last step of the process.

For the hardware stalls, ESTIMA collects the backend stalled cycles (as available by the architecture). Choosing the backend stalls for an architecture involves identifying the counters that measure stalled cycles in the pipeline. From this set, we discard the events that refer to instruction fetching, keeping only the stalls in the execution phase of an instruction. We also discard events that significantly overlap, such as aggregate events for backend stalls. Intuitively, stalls that overlap can make predictions pessimistic, depending on the extent to which they overlap.

Processor families typically share the same set of counters, and using ESTIMA with different processors of the same family requires no changes in its configuration. Adding support for a new processor family requires consulting the developer's manual of the processor and identifying these backend stalls. For the machines that we had available, identifying the stalls to be used was a simple task that was necessary only once for each manufacturer. The same counters were then used for both desktop and server machines, without the need to choose different counters.

When choosing the software stalls to be (optionally) collected, the developer needs to consider the parts of (mainly synchronization) code that, when executed, produce no useful work for the workload. Such cases include (but are not limited to) spinning on locks and looping on *trylock* operations. An interesting case is *Software Transactional Memory* (STM), where deciding on the cycles that are not producing useful work is straightforward (aborted transactions), and an STM runtime can report these measurements directly.

3.1.2 Stalled Cycles Regression Analysis. This step consists of regressing the stalled cycles measurements. ESTIMA uses function approximation [1] to construct a set of functions for each stall category. Function approximation takes the measurements, a function type (e.g., a polynomial function of degree d), and constructs a function that closely fits the measurements (e.g., calculates the coefficients of the polynomial). ESTIMA chooses one function for each stall category and uses it to extrapolate the measured values. The approximation process, shown in Figure 4, consists of the following steps, assuming m measured values for a specific category of stalled cycles:

Table 1. Extrapolation Function Types

Name	Function
Rat22	$\frac{a_0+a_1 n+a_2 n^2}{1+b_1 n+b_2 n^2}$
Rat23	$\frac{a_0+a_1 n+a_2 n^2}{1+b_1 n+b_2 n^2+b_3 n^3}$
Rat33	$\frac{a_0+a_1 n+a_2 n^2+a_3 n^3}{1+b_1 n+b_2 n^2+b_3 n^3}$
CubicLn	$a + b\ln(n) + c\ln(n)^2 + d\ln(n)^3$
ExpRat	$e^{\frac{a+bn}{c+dn}}$
Poly25	$y = a + bx + cx^2 + dx^{2.5}$

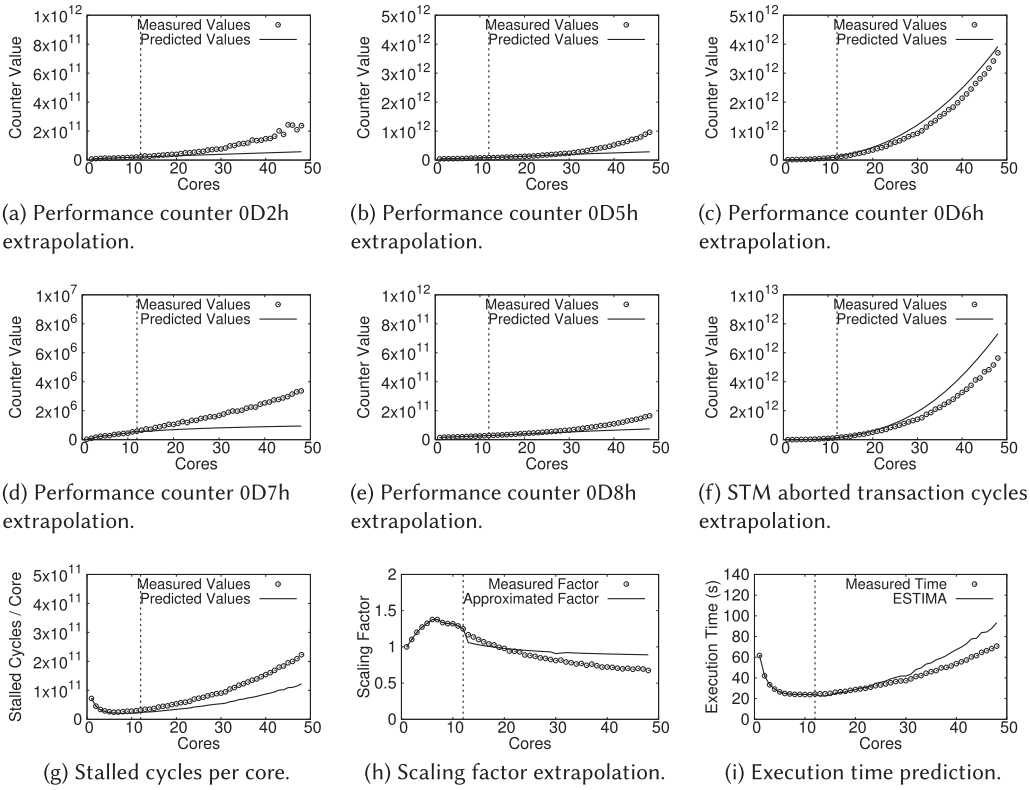
(1) From the m available measurements, ESTIMA designates the c measurements with the highest core counts as checkpoints. In our experiments, we set c to 2 and 4.
(2) Using the first n measurements ($n = m - c$), ESTIMA creates functions from the predefined kernels in Table 1. These functions are used based on the approximation library used (see Section 4.1), discarding the function types that produce functions that are not realistic for this approximation. The process is repeated for i in $3..n$, to avoid over-fitting the function to the available measurements. Intuitively, small deviations in the measurements sometimes steer the function in the wrong direction, resulting in less accurate predictions.
(3) For each of the constructed functions, ESTIMA calculates the root mean square error (RMSE) at the checkpoints. By using only the checkpoints, functions that have deviations for low core counts but approximate performance counter values accurately for higher core counts are considered as possible choices.
(4) ESTIMA chooses the function that minimizes the error and subsequently uses it to approximate the stalled cycle values for higher core counts.

3.1.3 Translating Stalled Cycles to Execution Time. After all the stalled cycle events have been approximated, ESTIMA calculates the total stalled cycles per core, using the approximated values. The total stalled cycles per core and the execution time have similar curves, including minima and maxima points. However, they represent different quantities. An example has already been introduced in Figure 2, where execution time and stalled cycles are shown for the `intruder` and `blackscholes` applications. The two quantities are not *similar*, in the sense that there is no constant number that connects them. Their similarity factor is rather a function of the number of cores.

ESTIMA uses the stalled cycles and the execution time measurements, collected during the execution of the application, to calculate the values of the scaling factor function for the available core counts. It then extrapolates this function using the same kernels as before (Table 1). In this case, ESTIMA no longer chooses the function that best fits the points. In contrast, it chooses the function that produces execution time predictions that have the highest correlation to the total stalled cycles per core. The reason is that we have already argued that execution time and stalled cycles have a very high correlation. As such, the produced execution time values should retain high correlation to the total stalled cycles per core that were calculated in the previous step. After the factor function has been created, ESTIMA uses the stalled cycles per core (from step B) and the factor function to calculate the execution time of the application for higher core counts.

3.2 Prediction Example

To better explain the prediction process, we use `intruder` from the STAMP benchmark suite as an example. `intruder` is a signature-based network intrusion detection system (NIDS) benchmark that scans network packets and matches them against a set of intrusion signatures. It emulates



(a) Performance counter 0D2h extrapolation.
(b) Performance counter 0D5h extrapolation.
(c) Performance counter 0D6h extrapolation.
(d) Performance counter 0D7h extrapolation.
(e) Performance counter 0D8h extrapolation.
(f) STM aborted transaction cycles extrapolation.
(g) Stalled cycles per core.
(h) Scaling factor extrapolation.
(i) Execution time prediction.

Fig. 5. `intruder` prediction example.

Design 5 of the NIDS described by Haagdorens et al. [14]. Network packets are processed in parallel and go through three phases: capture, reassembly, and detection. In the version included in STAMP, the capture and reassembly phases are each enclosed in STM transactions.

For this example, we use a machine with four AMD Opteron 6172 processors, each containing two chips with 6 cores each, clocked at 2.1GHz (48 cores in total). ESTIMA uses only one processor of the machine (12 cores) for measurements and targets a machine with four such processors. Hence, the measurements machine is a 12-core machine, while the target machine is a 48-core machine. We then execute the application on the target machine and measure the stalled cycles and execution time for up to 48 cores. We present the extrapolated and measured values in Figure 5. The vertical lines in the figures represent the maximum number of cores used for the measurements.

The first step of the process is to collect performance counters for executions of the application when using up to 12 cores (the application is configured to use as many threads as cores available). For this AMD Opteron machine, the performance counters that measure backend stalled cycles are the ones presented in Table 2 [3]. `intruder` uses software transactional memory as a concurrency control mechanism. We configure the SwissTM software transactional memory runtime [11] to report aborted transaction cycles for the application and configure ESTIMA to use these cycles.

ESTIMA then approximates each stalled cycle category individually. It creates multiple functions for each category and chooses the function that minimizes the RMSE at the checkpoints. With one function for each stalled cycle category, ESTIMA can extrapolate the stall values for higher core counts. In Figures 5(a)–(f), we present the result of this process. ESTIMA uses the measurements

Table 2. Hardware Performance Counters used for the *Opteron* machine

Event Code	Event Description
0D2h	Dispatch Stall for Branch Abort to Retire
0D5h	Dispatch Stall for Reorder Buffer Full
0D6h	Dispatch Stall for Reservation Station Full
0D7h	Dispatch Stall for FPU Full
0D8h	Dispatch Stall for LS Full

left of the vertical line and produces the functions presented. In each figure, we also show the measured values on all 48 cores of the *Opteron* machine.

After the performance counter values have been approximated, ESTIMA computes the stalled cycles per core, shown in Figure 5(g). It is important to note here that although the stalled cycles for each category increase, the total number of stalled cycles divided by the number of cores decreases for up to 12 cores. This quantity starts increasing for more cores, hinting at a slowdown for the application for higher core counts. Intuitively, as the application runs on more cores, stalled cycles increase *"faster."* That means that by scaling up the application, we introduce overheads that surpass the benefits in terms of performance. This observation validates the reasons behind ESTIMA's use of fine-grain stalls in the place of an aggregate event (as discussed in Section 2.5). By only examining the total number of backend stalls (which is what an aggregate performance counter would report) in Figure 5(g), we notice that the slowdown of the application is not visible for measurements with up to 12 cores. As a result, if ESTIMA simply extrapolated the values of such an aggregate counter, it would fail to predict the behavior of `intruder` for higher core counts, similarly to the time extrapolation method. In contrast, by using fine-grain stalls, ESTIMA captures the behavior of each stall cycle category and extrapolates them individually. The resulting values, when combined, are able to predict the fast increase in the total number of stalls, resulting in accurate predictions.

The correlation of stalled cycles to execution time involves a scaling factor that connects the two quantities. ESTIMA computes the values of this factor for up to 12 cores using the stalled cycles per core and the execution time values collected. It then approximates this factor. For this approximation, it chooses the function that produces execution time predictions that have the highest correlation to stalled cycles per core. The approximation of the scaling factor is presented in Figure 5(h). Finally, using this scaling factor, ESTIMA predicts the execution time of the application. We then measure the execution time of `intruder` for up to 48 cores of the machine and use the measurements to evaluate our prediction. Both the predicted and measured execution times are presented in Figure 5(i). ESTIMA successfully predicts the scalability of the application and the slowdown it exhibits for higher core counts.

4 EVALUATION

4.1 Implementation

We implement ESTIMA in Python. We integrate the functionality in an easy-to-use tool. For the function approximation, we use the `pythonequation` library from [34] to create functions based on specific kernels and fit them to collected values of stalled cycles. ESTIMA offers a variety of options for different prediction scenarios. It can either discover the number of cores of the machine it runs on or take the number of cores to use as an input parameter. ESTIMA discovers the topology of the cores and uses cores within the same socket first. It supports current x86 processor families by both Intel and AMD, and uses all the available events for backend stalls on each architecture.

Extending ESTIMA to support additional families of processors is straightforward, by adding the necessary performance counters that need to be collected.

To improve the accuracy of predictions, ESTIMA enables the use of *plugin* components. The user can specify additional categories of stalled cycles that can be used for the predictions, either at the hardware or the software level. ESTIMA takes a configuration file that includes the path to the file the stalls are reported in (including special files like *stdout* or *stderr*), as well as the expression that is used to report the cycles. ESTIMA can apply a function to the collected values (e.g., *min*, *max*, *sum*, *average*) and use the resulting values for its predictions. For example, extending ESTIMA to take into account Intel's RTM [20] would require including additional stalled cycle events (currently there are no available events that measure aborted RTM transaction cycles but rather only aborted transactions). The way ESTIMA collects additional stalls can vary between applications. In our evaluation, for the collection of synchronization overheads we use a thin wrapper around the `pthread` library. For the applications that use transactional memory, we use SwissTM [10, 11] with detailed statistics enabled, which reports the duration of committed and aborted transactions.

4.2 Evaluation Setup

We evaluate ESTIMA using three different machines. The first is a desktop *Intel Core i7* Haswell machine with 4 cores clocked at 3.4GHz (8 hardware threads in total). The second machine is a four-processor AMD Opteron 6172 one, with each CPU containing two chips with 6 cores each, clocked at 2.1GHz (48 cores in total). In the remainder of the text, we refer to this machine as *Opteron*. We also use a machine that has two Intel Xeon E5-2680 v2 processors with 10 cores each, clocked at 2.80GHz (40 threads in total). We refer to this machine as *Xeon20*.

We use several applications to evaluate ESTIMA. These span a variety of workloads, with different lengths of critical sections, levels of contention and synchronization techniques. In total, we use 21 different workloads, among which 8 are STM based.

4.3 Extrapolating to Different Machines

We start our evaluation with two production applications, in a realistic setting. We use `memcached` and a `SQLite` application. We use a desktop machine for our measurements and predict the scalability of the applications on a server machine. We then execute the applications on the server machine and evaluate the accuracy of our predictions.

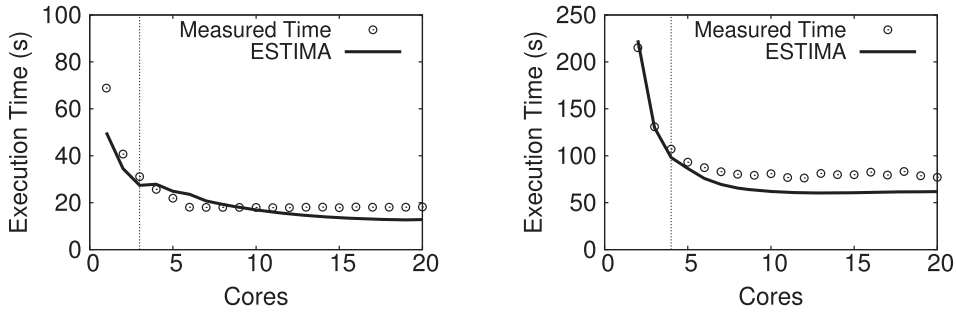
In our first experiment, we use ESTIMA to predict the scalability of `memcached`. We use ESTIMA on the desktop Haswell machine and target *Xeon20* with our predictions. We run clients on the same machine as the `memcached` server to remove any network effects. The client and dataset are from `cloudsuite` [13], with a scaling factor of 10×. We use the number of workers and connections that produces the highest throughput. The workload is read-mostly and objects have a size of 550 bytes.

ESTIMA collects stalled cycles and execution time from the `memcached` server using up to three hardware threads of the desktop machine (clients run on all other hardware contexts) and extrapolates its performance to a machine 7 times its size. The performance counters that measure backend stalled cycles for our *Intel* machines are presented in Table 3 [20]. As presented in Section 3.1, ESTIMA uses measured execution time to correlate stalled cycles to the execution time of the application for higher thread counts. In this experiment, because the machines have processors with different frequencies, the measured execution time is also scaled using the ratio of execution frequencies.

We then measure the execution time of the workload on the *Xeon20* machine, using all 20 cores. We execute the `memcached` server on one socket of the machine and the clients on the other socket (20 hardware contexts each). We present ESTIMA's prediction and the measured execution time in Figure 6(a). ESTIMA successfully predicts the scalability of the application. The absolute errors are

Table 3. Hardware Performance Counters Used for the Latest *Intel* Processors

Event Code	Event Description
0487h	Stalled cycles due to IQ full
01A2h	Cycles allocation stalled due to resource-related reasons
04A2h	No eligible RS entry available
08A2h	No store buffers available
10A2h	Re-order buffer full



(a) memcached execution time prediction. (b) SQLite execution time prediction.

Fig. 6. Predictions for memcached and SQLite.

below 30% for all core counts. ESTIMA successfully predicts that the server will stop scaling, using only three cores for the measurements, while predicting for up to 7 times more cores.

The second experiment uses the SQLite in-memory DBMS, and a TPC-C workload with 10GB of data. We use the same Haswell desktop machine and target *Xeon20* again. We use *tmpfs* to avoid IO bottlenecks for logging. ESTIMA collects stalled cycles and execution time from the SQLite process for up to four cores of the desktop machine and extrapolates its performance to a machine 5 times its size. We measure the same stalled cycles and scale the execution frequencies as before.

We then measure the execution time of the workload on the server machine, using all 20 cores. We keep the threads on the same processor when possible. The result of the prediction produced by ESTIMA, as well as the actual time measurements from the server machine are presented in Figure 6(b). ESTIMA successfully predicts the scalability of the application on the server machine. The execution time errors are below 26% for all core counts. ESTIMA successfully predicts that the server will stop scaling, as well as the number of cores for which this will happen. It does so using only four cores for the measurements and predicting for a machine with 5 times more cores.

4.4 Scaling-up Applications

In our previous experiments, we show how we use ESTIMA to extrapolate the scalability of production applications. We now evaluate the extent to which our tool can predict the scalability of applications by using three suites of in-memory benchmarks: STAMP [29], Parsec [5], and standard data structure micro-benchmarks (used in [10]). We also use a modified k-nearest neighbors (*KNN*) calculation kernel, commonly used in recommender systems. The benchmark suites we use are written in C/C++ and compiled using GCC, while the KNN calculation kernel is written in Java and compiled using GCJ. We use one CPU of the *Opteron* and *Xeon20* machines for our measurements (12 and 10 cores, respectively) and then predict for up to four and two CPUs, respectively (the full machines).

Table 4. Maximum Prediction Errors with Measurements on One Processor of Each Machine (12 Cores for *Opteron* and 10 Cores for *Xeon20*)

	Opteron Errors (%)			*Xeon20* Errors (%)
Benchmark	2 CPUs	3 CPUs	4 CPUs	2 CPUs
lock-based HT	7.8	8.3	8.9	41.7
lock-based SL	27.7	24.3	21.4	16.1
lock-free HT	3.3	3.4	3.7	15.8
lock-free SL	13.2	10.4	9.9	24.8
stamp:				
genome	4.4	4.4	4.6	6.3
intruder	9.2	22.1	31.9	30.0
kmeans	50.3	50.9	17.0	30.2
labyrinth	15.4	15.0	18.4	9.9
ssca2	2.8	4.6	8.1	21.4
vacation-high	14.7	14.3	10.3	16.8
vacation-low	18.9	18.5	25.0	10.0
yada	8.1	23.0	15.1	40.3
parsec:				
blackscholes	3.7	4.4	2.9	13.9
bodytrack	1.3	3.0	5.9	8.5
canneal	10.7	12.4	8.3	6.4
raytrace	2.7	3.6	4.6	1.7
streamcluster	15.6	59.0	88.8	20.1
swaptions	10.6	14.7	20.3	9.3
K-NN	11.5	22.5	32.0	13.1
Average	11.3	16.8	17.7	17.7
Std. Dev.	11.2	15.0	18.9	11.0
Max.	50.3	59.0	88.8	41.7

In Table 4, we present the summary of the predictions produced by ESTIMA for *Opteron* and *Xeon20*. The errors presented are the maximum errors observed when using one processor and targeting up to the full machines with our predictions (two, three, and four processors of *Opteron* and two processors of *Xeon20*). For brevity, we present examples using *Opteron*, for which we have predictions for higher core counts. In our examples, we also present the results of the time extrapolation method presented in Section 2.5 (*time extrapolation*). This method involves directly extrapolating time measurements of an application, using the function kernels that ESTIMA uses to extrapolate stalled cycle measurements. This approach is similar to using aggregate events for our measurements. It fails to predict changes in the application behavior that are not evident from the measurements, which explains the significant differences in accuracy when compared to ESTIMA. We highlight the biggest of these differences in accuracy between *time extrapolation* and ESTIMA in Figure 7.

Our benchmark evaluation shows that ESTIMA is successful in predicting the scalability of most benchmarks with small prediction errors. Of 19 workloads used for the evaluation of ESTIMA:

- ESTIMA is successful in predicting the scalability of the applications used. There are no cases where ESTIMA incorrectly predicts that an application will or will not scale. ESTIMA also successfully predicts the core counts for which an application will stop scaling,

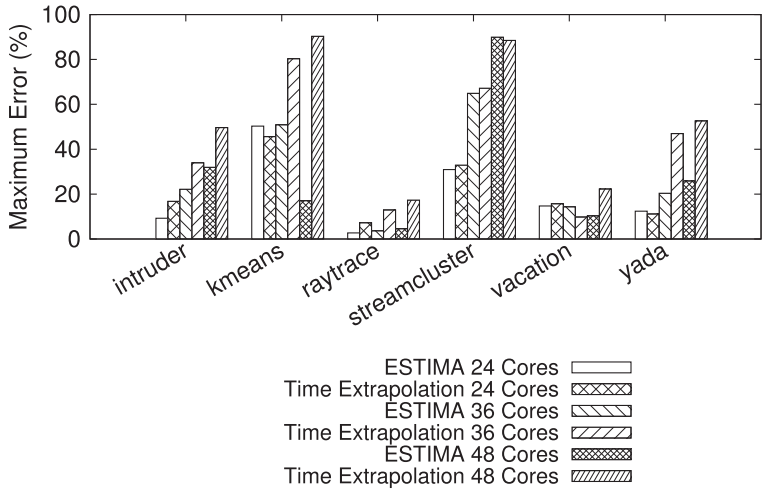


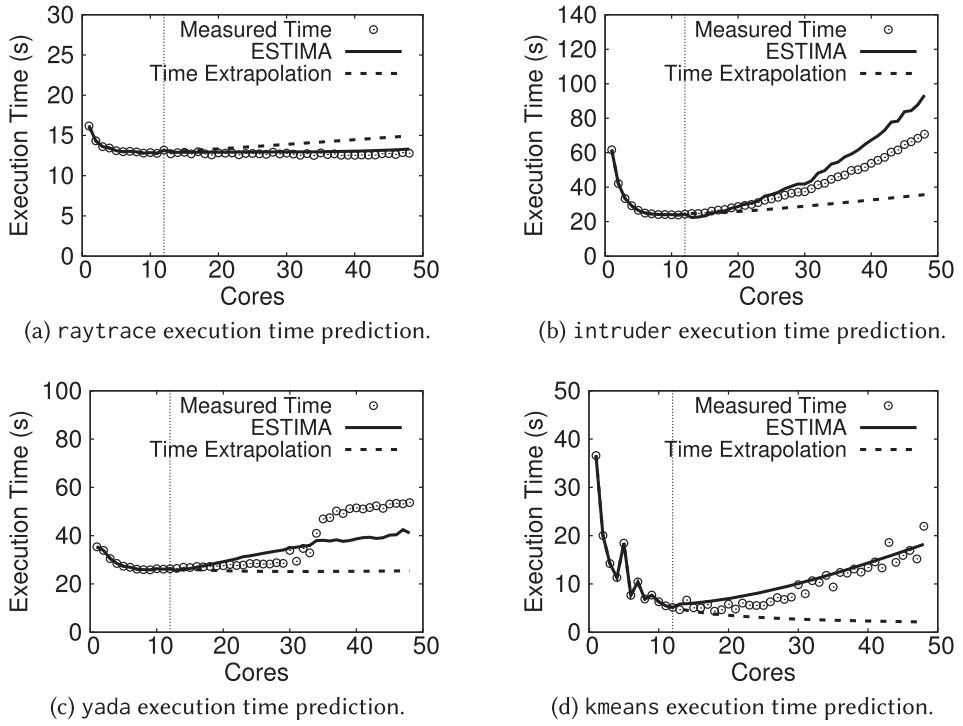
Fig. 7. Comparison of errors between ESTIMA and time extrapolation.

- When extrapolating from one socket of *Xeon20* to the full machine (double the number of cores used for the measurements), 15 workloads have execution time prediction errors lower than 25% and 9 of them have errors lower than 10%.
- When extrapolating from one socket of *Opteron* to the full machine (4 times the number of cores used for the measurements), 16 workloads have execution time prediction errors lower than 25% and 9 of them have errors lower than 10%.

In Figure 8(a), we present an example of a prediction result, using `raytrace` from the PARSEC benchmark suite. `raytrace` is an Intel RMS application that uses a version of the raytracing method that would typically be employed for real-time animations such as computer games, optimized for speed rather than realism. ESTIMA accurately predicts its scalability, with the maximum execution time prediction error observed for predictions for up to 48 cores being 4.6%. This is in contrast to the time extrapolation method, which produces errors up to 17.3%.

The main advantages of ESTIMA appear when predicting changes in the behavior of an application, such as the ones seen in `intruder` and `yada` from the STAMP benchmark suite, presented in Figures 8(b) and (c). `intruder` has already been introduced in Section 3.2. `yada` implements Ruppert's algorithm for Delaunay mesh refinement [36]. The input consists of an initial mesh and threads identify the triangles of the mesh for which the minimum angle is below some threshold. Once such triangles are found, new points are added to the mesh and the process continues with new triangulations. For both workloads ESTIMA successfully predicts the changes that appear, as well as the scalability of the applications. These cases demonstrate the advantage of using stalled cycles for our predictions, as the trends in stalled cycles appear before their effect in performance is significant enough. This is unlike time extrapolation, which fails to predict the scalability trends and has higher prediction errors (up to 81% and 130% higher for `intruder` and `yada`, respectively).

Another interesting example is `kmeans` from the STAMP suite. `kmeans` is a partition-based clustering benchmark that represents a cluster by the mean value of all objects contained in it. We present a scalability prediction for `kmeans` on the *Opteron* machine in Figure 8(d). Although the maximum prediction error for `kmeans` is 50.9% in Table 4, the prediction is accurate. The high error value is the result of fluctuations in `kmeans`' execution time for different core counts, which the prediction does not follow. Nevertheless, ESTIMA successfully predicts the scalability of the



(a) raytrace execution time prediction.

(b) intruder execution time prediction.

(c) yada execution time prediction.

(d) kmeans execution time prediction.

Fig. 8. Predictions using ESTIMA.

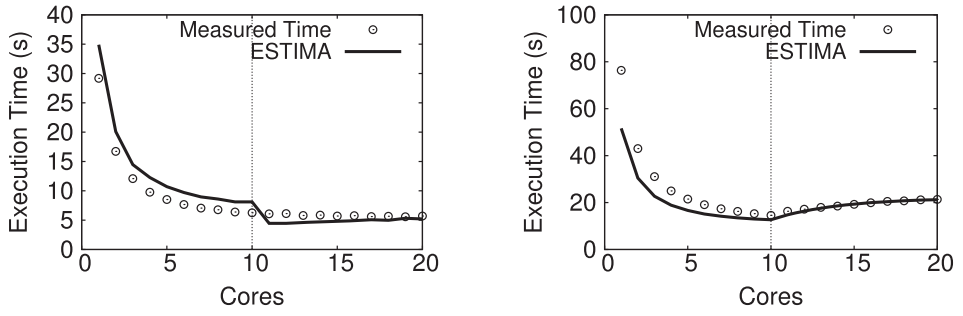
application. As with intruder and yada, the scalability degradation of kmeans is not evident in the performance on the measurement machine. However, ESTIMA successfully captures the trends in stalled cycles and predicts the performance degradation accurately.

4.5 Weak Scaling

Using a larger machine enables users to also use bigger datasets, due to the bigger amount of memory typically available. To evaluate how we can use ESTIMA with changing workload sizes, we use measurements of an application on a machine and predict its scalability on a machine with double the number of cores but also with twice as large a dataset. We use genome and intruder from the STAMP benchmark suite, introduced earlier in this section. We run our experiments on the *Xeon20* machine. We use both applications with the default datasets from the STAMP suite.

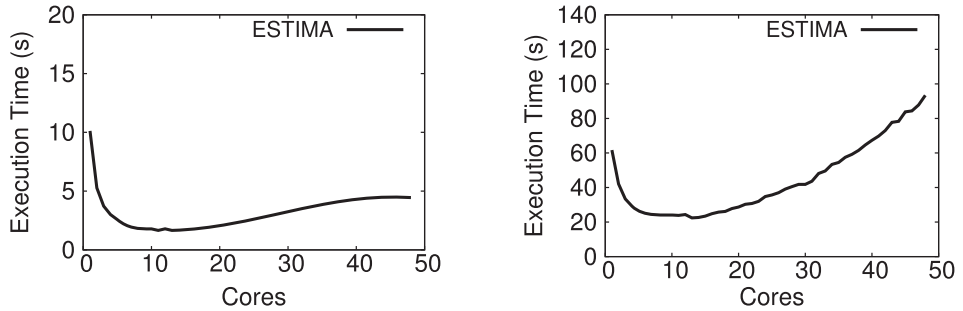
ESTIMA executes measurements on one processor of the *Xeon20* machine and targets the full machine. We configure ESTIMA with a target dataset size of 2×. ESTIMA uses the same technique for extrapolating the scalability of the applications but also measures the memory footprint of the execution. During the extrapolation process, ESTIMA scales the extrapolated values accordingly to produce predictions for the target machine and dataset for both applications.

We then execute the applications on the full *Xeon20* using the bigger dataset. We present both ESTIMA's prediction, as well as the measured execution time in Figure 9. ESTIMA successfully predicts the scalability for both applications. The predictions are accurate in absolute terms, too. The maximum errors (excluding single core performance) are 28% for intruder and 29% for genome. The most significant errors appear for single core performance of intruder on the bigger machine. This can be explained due to the simple scaling that we use to target the bigger dataset, which does not accurately connect the performance on a single core.



(a) genome execution time prediction. (b) intruder execution time prediction.

Fig. 9. Predictions with changing workload sizes.



(a) Prediction for streamcluster. (b) Prediction for intruder.

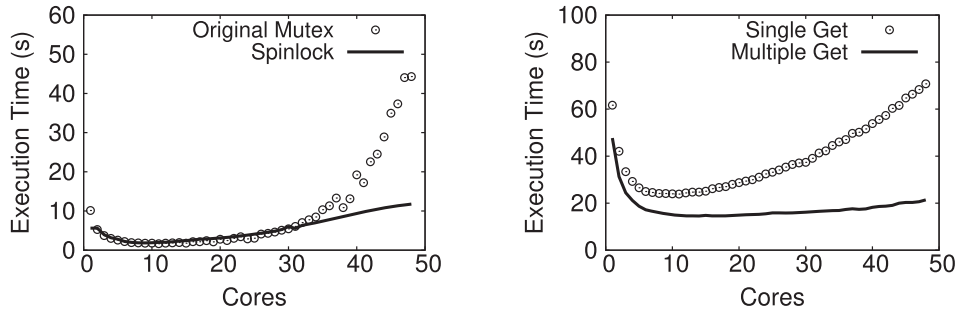
Fig. 10. Predictions for streamcluster and intruder using ESTIMA.

Although with higher errors than their strong scaling counterparts, these predictions show how, with a simple technique, ESTIMA can scale both the workload size and together the core counts at the same time, while maintaining the generality and wide applicability of ESTIMA, that is, without using complex models for the interaction between the workload and the different cache levels. We expect more complex methods, such as scaling different stalled cycle categories differently based on measurements, to be able to produce even more accurate predictions for different workload sizes.

4.6 Identifying Future Bottlenecks

In our evaluation so far, we present how ESTIMA can be used to extrapolate stalled cycles to predict the scalability of an application. A question that arises is the following: *Can* ESTIMA *help developers identify the sources of poor scalability in their applications, before such poor scalability even appears?* We now show how we can use ESTIMA to identify bottlenecks in two examples. We use two applications that use different concurrency control mechanisms: `streamcluster` and `intruder`. For both applications, ESTIMA collects stalled cycles from both hardware and software sources. For `streamcluster`, we create a thin wrapper around the pthread library calls. For `intruder`, we simply configure the SwissTM runtime library to report aborted transaction cycles.

ESTIMA extrapolates the scalability of the two applications on *Opteron*. It uses measurements on one processor of the machine (12 threads) and extrapolates to all four processors (48 threads). We show the results of these extrapolations in Figure 10. Both applications exhibit slowdown for high core counts. We observe the individual stalled cycle category extrapolations and identify the ones that contribute most to stalls for higher core counts. We then use the *perf* linux tool to pinpoint



(a) `streamcluster` scalability. (b) `intruder` scalability.

Fig. 11. Improving the scalability of `streamcluster` and `intruder` using ESTIMA's predictions.

the most significant sources of the reported stalls in our measurements. For `streamcluster`, we notice a high number of stalled cycles in the *pthread_mutex_trylock* function of the PARSEC barriers. This leads us to identify the mutexes used as a potential scalability bottleneck. For `intruder`, we similarly notice a high number of stalled cycles from aborted STM transactions in the *processPackets* function and more specifically in the *TMDECODER_PROCESS* call. By examining the function code, we understand that aborted transactions are the result of contention for a shared data structure.

To fix the problems identified, we modify the two applications. For `streamcluster`, we replace the standard pthread mutexes used by PARSEC with test-and-set spinlocks, which incur lower synchronization overheads. For `intruder`, we modify the application to decode more elements in every step. The measurements on the full *Opteron* machine validate our findings for the scalability bottlenecks. We show the original and modified applications' performance in Figure 11. For `streamcluster`, we improve its execution time by up to 74%. Similarly, our optimization improves `intruder`'s performance by up to 70%. Evidently, the applications continue to scale poorly. There do exist more bottlenecks that we could identify and try to remove. Nevertheless, the goal of this example is to showcase how ESTIMA can be used to identify future scalability bottlenecks. It is important to note here that identifying bottlenecks is not the main goal of ESTIMA, and as such, it cannot replace tools specifically designed for this purpose.

5 DISCUSSION

5.1 Stalled Cycles and Scalability

The main insight behind ESTIMA is the usage of backend stalled cycles for scalability extrapolations. This assumes that stalled cycles contain the necessary information about the scalability of an application. Our evaluation shows that, indeed, by using backend stalls from both hardware and software, ESTIMA successfully extrapolates the scalability of a wide range of applications.

We notice that for some workloads and platforms, prediction errors are higher. Although these cases are limited, it is crucial to understand the cause of such errors. To do so, we need to first consider *where* errors can occur in ESTIMA. The two main components of the extrapolations ESTIMA performs are the following:

(1) The extrapolation of stalled cycles from hardware (and optionally software).
(2) The correlation of the extrapolated stalled cycles to execution time, and the extrapolation of the scaling factor that connect the two quantities.

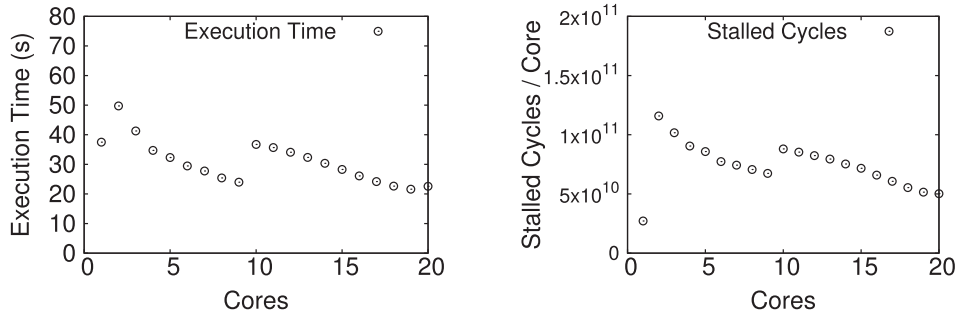
Table 5. Correlation of Stalled Cycles per Core with Execution Time for the Full Machines

Benchmark	*Opteron*	*Xeon20*	*Xeon48*
lock-based HT	0.71	0.66	0.93
lock-based SL	1.00	1.00	1.00
lock-free HT	1.00	1.00	1.00
lock-free SL	0.83	0.81	0.70
stamp:			
genome	1.00	1.00	1.00
intruder	1.00	0.97	0.92
kmeans	0.88	0.98	0.96
labyrinth	0.99	1.00	1.00
ssca2	0.99	1.00	1.00
vacation-high	1.00	1.00	0.99
vacation-low	0.99	1.00	0.98
yada	0.62	0.95	0.77
parsec:			
blackscholes	1.00	1.00	1.00
bodytrack	1.00	1.00	1.00
canneal	0.97	0.99	0.95
raytrace	0.94	0.98	0.96
streamcluster	0.84	1.00	0.81
swaptions	1.00	1.00	1.00
K-NN	1.00	1.00	0.99
Average	0.93	0.97	0.94
Std. Dev.	0.11	0.08	0.09
Min.	0.62	0.66	0.70

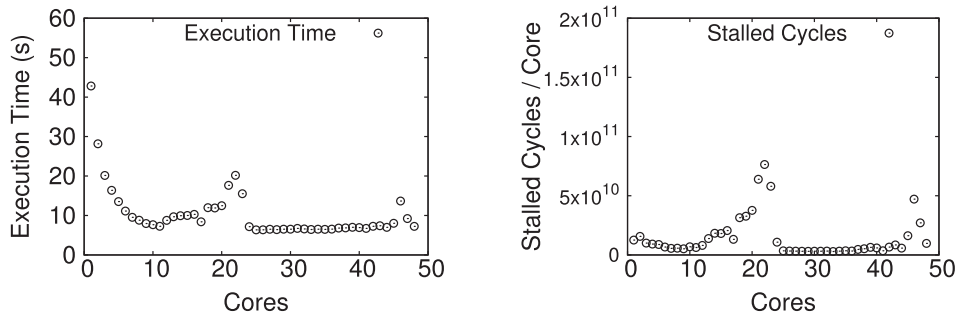
Both of these steps are potential sources of errors. Function approximation can produce functions that do not extrapolate accurately either a category of stalled cycles or the scaling factor that connects stalled cycles to execution time. In this last step, inaccuracies can result in significant errors in the prediction of the scalability of an application. Nevertheless, ESTIMA assumes that stalled cycles can tell the full story of scalability, and errors are due to the limitations of function approximation. If this does not hold true, then extrapolation errors can be significantly higher. Even worse, ESTIMA could predict that an application will continue scaling, while in fact it will not.

To evaluate this assumption, we conduct the following experiment, using an additional Intel machine with 4 E7-4830 v3 processors (48 cores, *Xeon48*). We execute all benchmarks for all core counts and measure both stalled cycles and execution time. We then calculate the correlation between stalled cycles per core and execution time. Ideally, this correlation should be 1.0, so errors are mainly due to function approximation. More importantly, such a high correlation would indicate that stalled cycles follow the changes of execution time and will correctly predict whether an application will stop scaling, as well as the core count for which this will happen.

Correlations across applications and machines (Table 5) are higher than 0.95 for the vast majority of cases, supporting ESTIMA's use of stalled cycles. What this implies is that errors in predictions are mainly due to function approximation errors. These numbers include software stalls for the applications of the STAMP benchmark suite, as well as `streamcluster` from the PARSEC suite. In the presented results, we notice cases with lower correlation, namely the lock-based hash



(a) `lock-based hash table` on *Xeon20*.



(b) `lock-free skip list` on *Xeon48*.

Fig. 12. Execution time and stalled cycles for two data structure microbenchmarks.

table microbenchmark on the *Xeon20* machine and the lock-free skip list on the *Xeon48* machine. We present the execution time and stalled cycle measurements for both workloads in Figure 12. Execution time and stalled cycles per core have similar curves. The correlation is lower because of small changes in core to core measurements that are not in sync for the two curves. However, in both cases, ESTIMA accurately extrapolates scalability, as we show in our evaluation (Table 4).

5.2 Backend vs. Frontend Stalled Cycles

When designing ESTIMA, we opted to use backend stalled cycles as our main tool. As discussed in Section 2, there also exist frontend stalls, which mainly refer to instruction fetch stalls (e.g., misses in the instruction cache). We chose to not use these stalls for two reasons: (a) Bottlenecks that hinder scalability do not typically stem from instruction fetching but rather the execution, and (b) there is a limit to the number of hardware counters that modern platforms can accurately monitor at the same time without imposing additional overheads to an application.

To verify the extent to which frontend stalls indeed do not contribute to the accuracy of ESTIMA, we extend the experiment of the previous subsection. We measure both frontend and backend stalls and calculate the correlation between the number of stalled cycles (including frontend stalls) per core and the execution time of a workload. We present the differences in correlation to execution time between with and without including frontend stalled cycles in Table 6. Positive values in the table signify that using frontend stalls improves the correlation, while negative values signify that using frontend stalled cycles degrades the correlation between the two quantities.

As seen from the table, the average improvement in the results is either close to zero, or negative. This indicates that frontend stalls do not contribute additional information about the scalability of

Table 6. Frontend+backend Stalled Cycles Improvement over Backend-only Stalls (%)

Benchmark	*Opteron*	*Xeon20*	*Xeon48*
lock-based HT	2.09	−6.93	−2.63
lock-based SL	0.00	−0.52	0.00
lock-free HT	0.01	0.00	−0.16
lock-free SL	7.52	6.76	3.53
stamp:			
genome	0.02	0.02	−0.10
intruder	0.08	1.31	−9.98
kmeans	2.86	−12.07	5.67
labyrinth	0.37	−0.28	−0.02
ssca2	−0.71	−0.05	−0.46
vacation-high	0.00	−0.02	−0.03
vacation-low	−0.04	−0.01	−0.10
yada	3.43	−0.10	5.57
parsec:			
blackscholes	0.00	0.00	−0.01
bodytrack	0.00	−0.01	−0.04
canneal	−0.03	0.01	−0.11
raytrace	0.13	−0.03	0.13
streamcluster	0.80	−14.79	−2.67
swaptions	0.00	0.00	0.00
K-NN	0.05	0.52	−0.12
Average	0.87	−1.38	−0.08
Std. Dev.	1.89	4.73	3.16
Max.	7.52	6.76	5.67
Min.	−0.71	−14.79	−9.98

the applications. In some cases using frontend cycles can decrease correlation to execution time by more than 10%, which can result in significant prediction errors for ESTIMA. These findings confirm our decision to not include frontend cycles in ESTIMA.

Similarly to frontend stalls, we could omit stalls that have an insignificant effect. Indeed, not all backend stall categories contribute equally to the total. However, the most important stalled cycle category varies per application. For example, while the *0D7h* event (Table 2) contributes less than 0.01% of the total number of stalls for `vacation-high` and less than 0.1% for `intruder` on *Opteron*, it contributes more than 30% of the total stalls for `blackscholes` on the same machine. As such, ESTIMA uses all the available backend stall events of a processor.

5.3 Software Stalled Cycles

ESTIMA by default uses only hardware stalls for its extrapolations. However, as presented in Section 4.1, users can configure ESTIMA via plugins to also include software stalls in its extrapolations. This way the accuracy of its predictions can be improved.

We measure software stalled cycles for all applications from the STAMP suite by configuring the STM runtime to report aborted transaction cycles. Because of the nature of STM, the effect of software stalls is expected to be high: contention for resources and data memory locations leads to

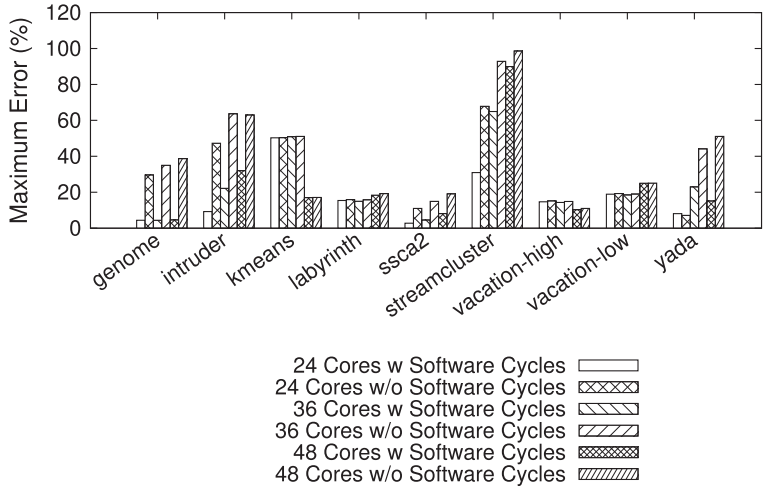
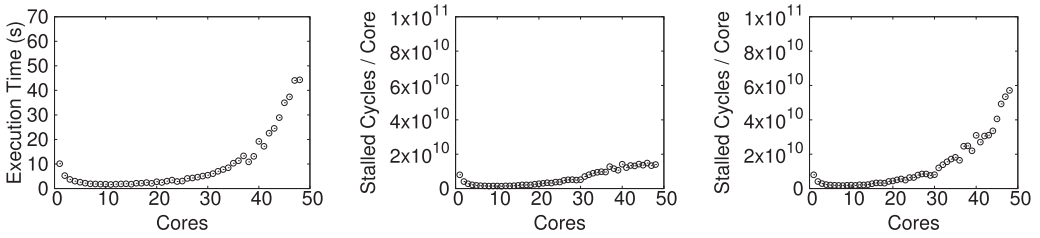


Fig. 13. Comparison of prediction errors with and without software stalled cycles.



(a) Execution time.

(b) Stalled cycles per core without synchronization cycles.

(c) Stalled cycles per core with synchronization cycles.

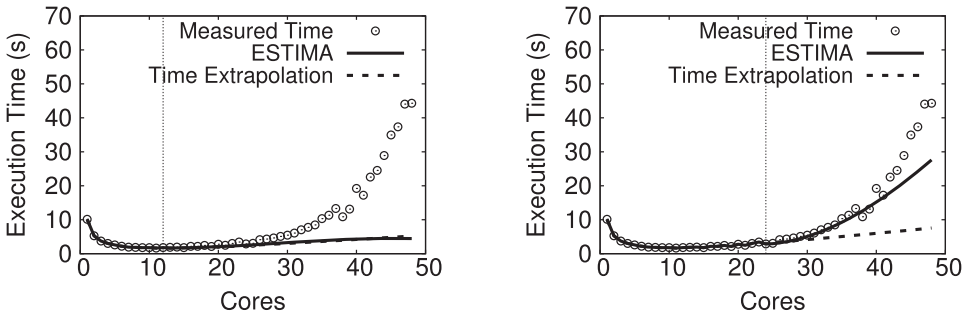
Fig. 14. Effect of software stalled cycles for `streamcluster`.

aborted transactions. Thus, although at the hardware level instructions are executed, all the work is discarded when a transaction is aborted, not contributing to useful work at the application level.

We additionally measure software stalled cycles due to synchronization for applications that use the `pthread` library. We do so by writing a thin wrapper around the library that measures the cycles each threads spends spinning on locks and barriers. We measure these synchronization cycles for `streamcluster` from the PARSEC suite, as well as for `genome` and `ssca2` from the STAMP suite.

In Figure 13, we present all the applications for which we configure ESTIMA to additionally use software stalled cycles. We show the prediction errors for the applications with and without software cycles. We observe that using software cycles can significantly improve the prediction accuracy: by 57% on average, and for certain applications (e.g., `genome`) by up to 87% when targeting a machine with four times the number of cores.

An application for which hardware stalls alone do not accurately capture its scalability is `streamcluster` from the PARSEC benchmark suite. For `streamcluster`, synchronization overheads introduce a significant bottleneck (one that we identify using ESTIMA in Section 4.6). The execution time of `streamcluster` on our *Opteron* machine can be seen in Figure 14(a). When using only hardware stalls (Figure 14(b)), stalled cycles per core do not show the synchronization bottleneck, resulting in lower correlation with execution time (0.86). When incorporating the cycles



(a) `streamcluster` execution time prediction.

(b) `streamcluster` execution time prediction with measurements for up to 24 cores.

Fig. 15. Predictions for `streamcluster`.

spent on synchronization, stalled cycles per core give a more complete image of the scalability of the application (Figure 14(c)), resulting in very high correlation to execution time (0.98).

5.4 Limitations

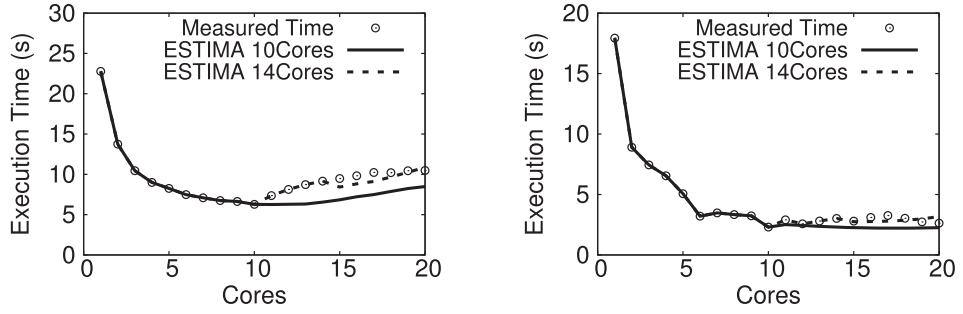
Table 4 shows that predictions for `streamcluster` from the PARSEC benchmark suite exhibit high absolute prediction errors. `streamcluster` is a clustering benchmark that, for a stream of input points, finds a predetermined number of medians, so each point is assigned to its nearest center. We show both the extrapolated and measured execution time of the application in Figure 15(a).

The behavior of `streamcluster` changes significantly for more than 30 cores. ESTIMA successfully captures the slowdown of the application, but with higher absolute errors, because there is no hint of this performance change in the measured stalls. When using 24 cores for the measurement (two sockets of *Opteron*), the prediction is significantly better, as seen in Figure 15(b). This shows the main limitation of ESTIMA. Although stalled cycles show trends before they have an impact on performance, as discussed in Section 2, there are cases where significant changes in the application happen for higher core counts. In this example, the synchronization overheads, together with memory bandwidth saturation, cause slowdown for core counts greater than 36. This behavior is not captured by stalled cycles when using measurements for up to 12 cores.

Moreover, ESTIMA is not a silver bullet for predicting the scalability of parallel applications. ESTIMA's main use case involves predictions for machines with similar architectures. ESTIMA successfully predicts the performance of an application across such machines, as we show in our evaluation. However, cross-platform predictions with significant differences between the measurements and target architectures (e.g., using measurements from an Intel machine to predict performance on a SPARC machine), will typically result in higher errors. Similarly, ESTIMA is not meant to predict the performance of an application for very different workload configurations, as it does not rely on static or dynamic analysis of the target application. We believe that these shortcomings are outweighed by the simplicity and generality of ESTIMA.

5.5 NUMA Effects

Because of its nature, ESTIMA does not capture effects that are not present in the measurements machine. As an application scales to more sockets, effectively changing the underlying architecture (e.g., introducing Non-Uniform Memory Access effects), prediction accuracy will be lower. The effect of NUMA accesses on the scalability of an application varies with the application, the target machine, and the dataset. Capturing the interaction of such factors would require a detailed



(a) `intruder` execution time prediction. (b) `kmeans` execution time prediction.

Fig. 16. Predictions with NUMA effects captured.

model of the application (e.g., memory access patterns), which is against ESTIMA's goal of wide applicability and ease of use.

However, this is not the case on our *Opteron* machine. The reason lies in its architecture, which enables ESTIMA to account for NUMA effects. Each processor of *Opteron* consists of 2 chips, each containing 6 cores (12 in total per processor). Thus, memory accesses on one processor introduce remote accesses between the 2 chips. With these effects present in measurements, ESTIMA accurately extrapolates their impact, resulting in predictions with high accuracy.

When observing the results for *Xeon20*, we see the average prediction errors are higher. Contrary to *Opteron*, *Xeon20* is a typical NUMA machine with two sockets. As such, single-socket execution measurements do not include the effects of NUMA accesses. Thus, ESTIMA fails to capture the trends that they cause, resulting in higher prediction errors for high core counts. Including these effects significantly improves the accuracy of predictions. For example, using measurements with more than 10 cores involves cores from the second socket, capturing non-local accesses and improving the prediction accuracy. We present two examples of such extrapolations in Figure 16.

To further demonstrate how non-uniformity can be accounted for in measurements, we conduct an experiment with our two Intel platforms. We use ESTIMA to execute and measure stalled cycles for all of the benchmarks used in our evaluation on both sockets of *Xeon20*, targeting the *Xeon48* machine introduced in Section 5.1 (2.5× the number of cores). We then execute the application on the *Xeon48* machine and measure the execution time for the same workload. Table 7 presents the maximum prediction errors for each application. In the same table, we have also copied the errors observed when extrapolating the scalability of one socket of *Xeon20* to the full machine (from Table 4). The results clearly show that prediction accuracy is significantly improved. The average prediction error falls from 17.7% to 13.9% (an improvement of 21.47%). Additionally, and more importantly, the prediction errors have a lower standard deviation (6.5 instead of 11). What this implies is that errors are better clustered around the average, rather than having multiple very small values, and a few very large ones. This is also evident in the absence of significant errors in the results: The maximum error observed is 30%, down from 41.7%, (an improvement of 28.06%).

6 RELATED WORK

The work that is most closely related to ESTIMA is that of Crovella et al. [8], in which the authors identify two categories of stalls at the software level: productive cycles and stalled cycles. They collect measurements of the two categories of cycles and use them to predict the performance of an application. In contrast, ESTIMA relies primarily on hardware stalls, using low-overhead performance counters offered by modern hardware. ESTIMA can, however, leverage software-level stalls

Table 7. Maximum Prediction Errors for Predictions Targeting Our *Xeon48* Machine (*Xeon20* Prediction Errors from Table 4 for Comparison)

Benchmark	*Xeon20* Errors (%)	*Xeon20* to *Xeon48* Errors (%)
lock-based HT	41.7	19.8
lock-based SL	16.1	18.4
lock-free HT	15.8	6.8
lock-free SL	24.8	23.1
stamp:		
genome	6.3	8.2
intruder	30.0	5.2
kmeans	30.2	30.0
labyrinth	9.9	18.0
ssca2	21.4	15.3
vacation-high	16.8	12.8
vacation-low	10.0	10.9
yada	40.3	15.2
parsec:		
blackscholes	13.9	13.4
bodytrack	8.5	11.4
canneal	6.4	8.9
raytrace	1.7	4.0
streamcluster	20.1	21.1
swaptions	9.3	11.6
K-NN	13.1	9.2
Average	17.7	13.9
Std. Dev.	11.0	6.5
Max.	41.7	30.0

in addition to hardware stalls to further improve its extrapolations, making it applicable to a wider variety of workloads.

Barnes et al. [4] use linear logarithmic functions to predict the scalability of message-passing scientific applications on large-scale parallel systems. The number and the configuration of CPUs are the inputs and the process uses linear logarithmic functions. ESTIMA targets in-memory applications that use shared data and leverages hardware and software stalls to extrapolate their scalability.

In [25], statistical techniques and regression analysis are used to build piecewise black-box polynomial and neural network models of scientific programs. Neural networks are also used in [21] to build models of SMG2000 applications executing on two different large-scale machines. Unlike ESTIMA, such models do not address the question of an application's scalability on a machine with significantly higher number of CPUs available.

In [32], the author extrapolates address stream profiles, using low-level metrics collected on memory access patterns, to study the memory performance of an application under strong scaling. In [7], the authors use call path profiles and expectations on the cost differences between executions to estimate the scalability costs incurred by different parts of the program. ESTIMA uses both hardware and software cycles to extrapolate the scalability of the application as a whole.

Several systems combine predictions of the sequential performance of single-node tasks performed by distributed cores with models of communication between them [6, 28, 46]. Similar

cross-platform performance predictions for large-scale machines using a combination of known relative performance of the two systems and partial execution of the workload are described in [44]. These systems use different, more detailed models than ESTIMA.

Various formal modeling techniques for distributed and concurrent systems have been proposed [22], including petri nets and queuing theory [33]. They were used to develop detailed analytic models for several applications [19, 24]. These models are very accurate. They require however in-depth understanding of the applications and the system. In contrast, ESTIMA can be used with little effort on *any* parallel in-memory application.

Models based on discrete-time Markov chains were developed for several STM algorithms [15–17] to compare different STM designs. Usui et al. [41] use a simple cost-benefit analysis to choose between locking and transactions. The performance model from [35] focuses on modeling transactional conflict behavior. Unlike ESTIMA, this approach requires heavy instrumentation of the application memory accesses.

Performance counter research has mainly focused on profiling of applications. In [38] and [42], the authors use performance counters to increase power efficiency, through thread scheduling and placement. In [43], the authors use performance counters to capture performance impacts as a function of resource usage. Srikanthan et al. [39] use performance counters for online workload scheduling in multiprogrammed environments. Jiménez et al. [23] devise a model based on performance counters to predict total power consumption. While performance counters have been used for many different goals, the low-level information they provide has not yet been exploited for scalability predictions, as in ESTIMA.

Finally, numerous projects have focused on identifying bottlenecks in parallel applications. Liu et al. [27] develop ScaAnalyzer, a tool that uses event-based sampling to identify memory-related bottlenecks in parallel applications and suggest optimizations that improve scaling. In [45], the author proposes a set of new performance counters, which, together with a new analysis method, can identify performance bottlenecks in out-of-order processors. Torrellas et al. [40] use hardware performance counters to identify scalability bottlenecks in parallel applications running on distributed shared-memory multiprocessors. The main goal of ESTIMA is to predict the scalability of an application for larger machines. By pinpointing the sources of stalled cycles predicted by ESTIMA, developers can also identify bottlenecks that will appear in their applications. This, however, does not replace tools specifically built for performance tuning and bottleneck identification.

7 CONCLUSION

We presented ESTIMA, a practical tool for extrapolating the scalability of in-memory parallel applications. ESTIMA is designed to help developers and users predict the scalability of applications with minimum effort, without the need for detailed models. To achieve that, ESTIMA uses stalled cycle measurements at the hardware and optionally at the software level, and function approximation on the collected values. ESTIMA is general and easy to use. It can be applied to *any* in-memory parallel application with minimum effort. It can also take advantage of application-specific user input to further improve the accuracy of its predictions.

ESTIMA produces accurate predictions, as conveyed by our extensive evaluation. We successfully used ESTIMA to predict the scalability of production applications, as well as a wide range of benchmarks. The errors when predicting for a machine up to 4 times larger than the one available for measurements were lower than 15% for more than half of the applications. Moreover, ESTIMA successfully captured the scalability behavior of all the applications used. Finally, we used ESTIMA to identify scalability bottlenecks in two applications, showing how developers can further benefit from ESTIMA during the development phase.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers of both PPoPP '16 and ACM TOPC, as well as the shepherd of the paper for the PPoPP '16 conference, Alexandro Baldassin, for their helpful comments on improving this article.

REFERENCES

[1] Naum I Achieser. 1992. *Theory of Approximation*. Dover Publications.
[2] Alaa R. Alameldeen and David A. Wood. 2006. IPC considered harmful for multiprocessor workloads. *IEEE Micro* 26, 4 (2006), 8–17. DOI: http://dx.doi.org/10.1109/MM.2006.73
[3] AMD. 2010. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors. (2010).
[4] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis R. de Supinski, and Martin Schulz. 2008. A regression-based approach to scalability prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS'08)*. ACM, 368–377. DOI: http://dx.doi.org/10.1145/1375527.1375580
[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architecture and Compilation Techniques (PACT'08)*. ACM, 72–81. DOI: http://dx.doi.org/10.1145/1454115.1454128
[6] Laura Carrington, Allan Snavely, and Nicole Wolter. 2006. A performance prediction framework for scientific applications. *Future Gener. Comp. Syst.* 22, 3 (2006), 336–346. DOI: http://dx.doi.org/10.1016/j.future.2004.11.019
[7] Cristian Coarfa, John M. Mellor-Crummey, Nathan Froyd, and Yuri Dotsenko. 2007. Scalability analysis of SPMD codes using expectations. In *Proceedings of the 21th Annual International Conference on Supercomputing (ICS'07)*. ACM, 13–22. DOI: http://dx.doi.org/10.1145/1274971.1274976
[8] Mark E. Crovella and Thomas J. LeBlanc. 1994. Parallel performance prediction using lost cycles analysis. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (Supercomputing'94)*. IEEE Computer Society Press, Los Alamitos, CA, 600–609.
[9] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, 1243–1254. DOI: http://dx.doi.org/10.1145/2463676.2463710
[10] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. 2011. Why STM can be more than a research toy. *Commun. ACM* 54, 4 (2011), 70–77. DOI: http://dx.doi.org/10.1145/1924421.1924440
[11] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. 2009. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, 155–165. DOI: http://dx.doi.org/10.1145/1542476.1542494
[12] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, 371–384. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan.
[13] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, 37–48. DOI: http://dx.doi.org/10.1145/2150976.2150982
[14] Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. 2004. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *Proceedings of the 5th International Workshop on Information Security Applications (WISA'04)*. Springer, 188–203. DOI: http://dx.doi.org/10.1007/978-3-540-31815-6_16
[15] Armin Heindl and Gilles Pokam. 2009. An analytic framework for performance modeling of software transactional memory. *Comput. Netw.* 53, 8 (2009), 1202–1214. DOI: http://dx.doi.org/10.1016/j.comnet.2009.02.006
[16] Armin Heindl and Gilles Pokam. 2009. An analytic model for optimistic STM with lazy locking. In *Proceedings of the 16th International Conference Analytical and Stochastic Modeling Techniques and Applications (ASMTA'09)*. Springer, 339–353. DOI: http://dx.doi.org/10.1007/978-3-642-02205-0_24
[17] Armin Heindl, Gilles Pokam, and Ali-Reza Adl-Tabatabai. 2009. An analytic model of optimistic software transactional Memory. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. IEEE Computer Society, 153–162. DOI: http://dx.doi.org/10.1109/ISPASS.2009.4919647
[18] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ACM, 289–300. DOI: http://dx.doi.org/10.1145/165123.165164

[19] Adolfy Hoisie, Olaf M. Lubeck, and Harvey J. Wasserman. 2000. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *Int. J. High Perf. Comput. Appl.* 14, 4 (2000), 330–346. DOI: http://dx.doi.org/10.1177/109434200001400405

[20] Intel. 2016. Intel 64 and IA-32 architectures software developer's manual, volume 3B: System programming guide. *Part* 1 (2016), 64.

[21] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. 2005. An approach to performance prediction for parallel applications. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing (Euro-Par'05)*. Springer, 196–205. DOI: http://dx.doi.org/10.1007/11549468_24

[22] Raj Jain. 1991. *The Art of Computer Systems Performance Analysis—Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley.

[23] Víctor Jiménez, Francisco J. Cazorla, Roberto Gioiosa, Mateo Valero, Carlos Boneti, Eren Kursun, Chen-Yong Cher, Canturk Isci, Alper Buyuktosunoglu, and Pradip Bose. 2010. Power and thermal characterization of POWER6 system. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT'10)*. ACM, 7–18. DOI: http://dx.doi.org/10.1145/1854273.1854281

[24] Darren J. Kerbyson, Henry J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman, and Michael L. Gittings. 2001. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. ACM, 37. DOI: http://dx.doi.org/10.1145/582034.582071

[25] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. 2007. 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)*. ACM, 249–258. DOI: http://dx.doi.org/10.1145/1229428.1229479

[26] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[27] Xu Liu and Bo Wu. 2015. ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2015)*. ACM, 47:1–47:12. DOI: http://dx.doi.org/10.1145/2807591.2807648

[28] Gabriel Marin and John M. Mellor-Crummey. 2004. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'04)*. ACM, 2–13. DOI: http://dx.doi.org/10.1145/1005686.1005691

[29] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the 4th International Symposium on Workload Characterization (IISWC'08)*. IEEE Computer Society, 35–46. DOI: http://dx.doi.org/10.1109/IISWC.2008.4636089

[30] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[31] Graham R. Nudd, Darren J. Kerbyson, Efstathios Papaefstathiou, S. C. Perry, John S. Harper, and Daniel V. Wilcox. 2000. Pace—A toolset for the performance prediction of parallel and distributed systems. *Int. J. High Perf. Comput Appl.* 14, 3 (2000), 228–251. DOI: http://dx.doi.org/10.1177/109434200001400306

[32] Catherine Rose Mills Olschanowsky. 2011. *Hpc Application Address Stream Compression, Replay and Scaling*. Ph.D. Dissertation. University of California at San Diego, La Jolla, CA.

[33] Carl Adam Petri. 1966. *Communication with Automata, New York: Griffiss Air Force Base*. Technical Report. Technical Report RADC-TR-65-377.

[34] James R. Phillips. 2013. ZunZun.com. Retrieved from http://www.zunzun.com.

[35] Donald E. Porter and Emmett Witchel. 2010. Understanding transactional memory performance. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'10)*. IEEE Computer Society, 97–108. DOI: http://dx.doi.org/10.1109/ISPASS.2010.5452061

[36] Jim Ruppert. 1995. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algor.* 18, 3 (1995), 548–585. DOI: http://dx.doi.org/10.1006/jagm.1995.1021

[37] Nir Shavit and Dan Touitou. 1995. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*. ACM, 204–213. DOI: http://dx.doi.org/10.1145/224964.224987

[38] Karan Singh, Major Bhadauria, and Sally A. McKee. 2009. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Arch. News* 37, 2 (2009), 46–55. DOI: http://dx.doi.org/10.1145/1577129.1577137

[39] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. 2016. Coherence stalls or latency tolerance: Informed CPU scheduling for socket and core sharing. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 2016)*. USENIX Association, 323–336. https://www.usenix.org/conference/atc16/technical-sessions/presentation/srikanthan.

[40] Josep Torrellas, Yan Solihin, and Vinh Vi Lam. 1999. Scal-tool: Pinpointing and quantifying scalability bottlenecks in DSM multiprocessors. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'99)*. ACM, 17. DOI: http://dx.doi.org/10.1145/331532.331549

[41] Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. 2009. Adaptive locks: Combining transactions and locks for efficient concurrency. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. IEEE Computer Society, 3–14. DOI: http://dx.doi.org/10.1109/PACT.2009.20

[42] Augusto Vega, Alper Buyuktosunoglu, and Pradip Bose. 2013. SMT-centric power-aware thread placement in chip multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 167–176. DOI: http://dx.doi.org/10.1109/PACT.2013.6618814

[43] Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. 2010. Online cache modeling for commodity multicore processors. In *19th International Conference on Parallel Architecture and Compilation Techniques (PACT'10)*. ACM, 563–564. DOI: http://dx.doi.org/10.1145/1854273.1854353

[44] Leo T. Yang, Xiaosong Ma, and Frank Mueller. 2005. Cross-platform performance prediction of parallel applications using partial execution. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing*. IEEE Computer Society, 40. DOI: http://dx.doi.org/10.1109/SC.2005.20

[45] Ahmad Yasin. 2014. A top-down method for performance analysis and counters architecture. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. IEEE Computer Society, 35–44. DOI: http://dx.doi.org/10.1109/ISPASS.2014.6844459

[46] Jidong Zhai, Wenguang Chen, and Weimin Zheng. 2010. 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)*. ACM, 305–314. DOI: http://dx.doi.org/10.1145/1693453.1693493

Received November 2016; revised May 2017; accepted May 2017