ESTIMA: Extrapolating ScalabiliTy of In-Memory Applications

Georgios Chatzopoulos
EPFL
georgios.chatzopoulos@epfl.ch

Aleksandar Dragojević
Microsoft Research
alekd@microsoft.com

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Abstract

This paper 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 various events, such as cache misses or waiting on a lock). 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 instance, using measurements of `memcached` and `SQLite` on a desktop machine, we obtain accurate predictions of their scalability on a server. Our extensive evaluation on a large number of in-memory benchmarks shows that ESTIMA has generally low prediction errors.

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 hold their whole datasets in memory. This way, applications avoid overheads of slow secondary storage and network, leaving the CPU as the main performance bottleneck [9, 12, 26, 29]. 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.

Consequently, the crucial question about performance of in-memory applications is that of their scalability with the increasing number of cores. Answering this question is very hard. Typical approaches include performing extensive performance evaluation or developing detailed models of the application [15, 27], which are time-consuming, error-prone, and require detailed knowledge of the application and the machine it executes on.

This paper presents ESTIMA[1], 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, 30, 44].

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 more cores 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 in order 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 and potential bottleneck identification.

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 an `SQLite` workload on a server machine based on measurements on a desktop. We then extensively evaluate ESTIMA using 21 benchmark 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 be used to identify their bottlenecks. More specifically:

[1] ESTIMA and accompanying files are available for download at *http://lpd.epfl.ch/site/estima*.

PPoPP '16, March 12 - 16, 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2851141.2851159

- 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.
- 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 helps identify bottlenecks as we illustrate through two parallel applications that exhibit poor scalability, `intruder` and `streamcluster` from the STAMP and PARSEC benchmark suites respectively.

The rest of the paper is organized as follows. We present the insights behind ESTIMA in Section 2. We explain how ESTIMA works in Section 3. We present its implementation in Section 4. We report on its evaluation in Section 5. We discuss the related work in Section 6 and conclude the paper in Section 7.

2. Insights Behind ESTIMA

In this section, we present the insights upon which ESTIMA is built. We first recall what stalled cycles are and then present their main sources and how they affect the scalability of an application. We discuss why ESTIMA uses stalled cycles, in contrast to the straightforward approach of extrapolating time. Finally, we present some of the decisions we have made and how they reflect on the capabilities of ESTIMA.

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 almost 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 represent 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 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 gains one can expect when scaling up an application and could be the reason behind even slowdown for higher core counts.

Hardware stalled cycles. Measuring hardware stalls is at the core of ESTIMA. They can be divided into two big categories, depending on the stalled execution stage. *Frontend stalls* are the stalled cycles in the fetch and the decoding phase of instructions in the pipeline, while *backend stalls* are the stalled cycles due to instructions being stalled while being executed. 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, frontend stalled cycles do not change significantly for increasing core counts.

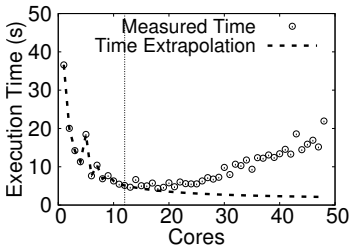


Figure 1: Time extrapolation for `kmeans`.

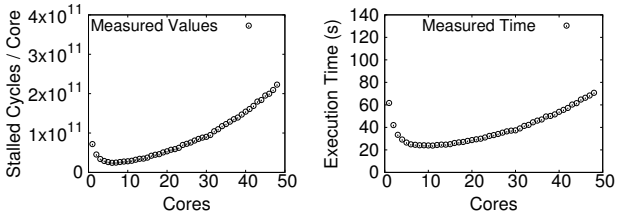
In contrast, backend stalled cycles have a direct impact on the scalability of an application, as they significantly increase when adding more cores. We have found no benefits in prediction accuracy when using frontend stalls. For this reason, and given that modern processors can measure up to 4 events concurrently, ESTIMA uses only backend stalls at the hardware level.

Software stalled cycles. Previous research has shown that 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. This can lead to prediction inaccuracies. A typical source of software stalls is synchronization (e.g. spinning on a busy lock). Another interesting scenario is the use of an optimistic concurrency control mechanism, such as *software transactional memory (STM)* [18, 36]. In applications that use STM libraries, aborted transactions discard all work done inside the transaction.

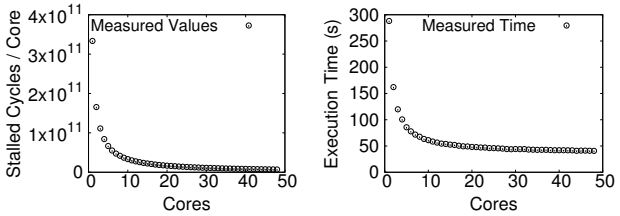
ESTIMA solves this problem by enabling the use of *software stalled cycles*. These represent cycles during which the application is executing instructions, but which are not producing useful work. Use of software stalled cycles is optional: users can decide to use a runtime that reports software-level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

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 extrapolating the measurements to higher core counts. An important drawback of this approach is that extrapolation requires the behavior to be evident 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. As we show in Section 5.3, ESTIMA does not suffer from these problems, improving the accuracy of the predictions. It does so by using lower-level information, in the form of stalled cycles.

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 evaluate the applications used in this paper and find that for all of them, the number of stalled cycles per core have a high correlation with execution time. Two such examples are shown in Figure 2. They are the `intruder` and `blackscholes` benchmarks from the STAMP [28] and PAR-



(a) `intruder` stalled cycles per core. (b) `intruder` execution time.



(c) `blackscholes` stalled cycles per core. (d) `blackscholes` execution time.

Figure 2: Stalled cycles and execution time correlation.

SEC [5] suites respectively. For both applications, there is over 98% correlation between the number of stalled cycles per core and their execution time. Similar high correlations are present in the rest of the applications we evaluate.

ESTIMA relies on stalled cycles for its predictions. By default it 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 a high-level image of the scalability of the application (Figure 2).

The insight behind using fine-grain stall events is that 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. Using aggregate events would not capture significant changes in the scalability of applications, which is the main goal of ESTIMA. By using fine-grain 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 (e.g. 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 bottleneck. By using the detailed events, ESTIMA can help identify the area that prohibits scalability and guide the developer to fix the bottlenecks, as we show in Section 5.5.

Other performance counters. Prior work [38, 41] 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 in order to translate the misses captured to time and quantify their effect to the scalability of the application. Using cache misses without a detailed model would cause the predictions to be pessimistic. The reason for this is that an increase in cache misses does not always result in poor scalability. Identifying the complex 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 process 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

A. 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 execution of the application, ESTIMA also measures the memory resident set size, as well as 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 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 for each machine.

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*, where deciding on the cycles that are not producing useful work is straightforward, and an STM runtime can report these measurements directly.

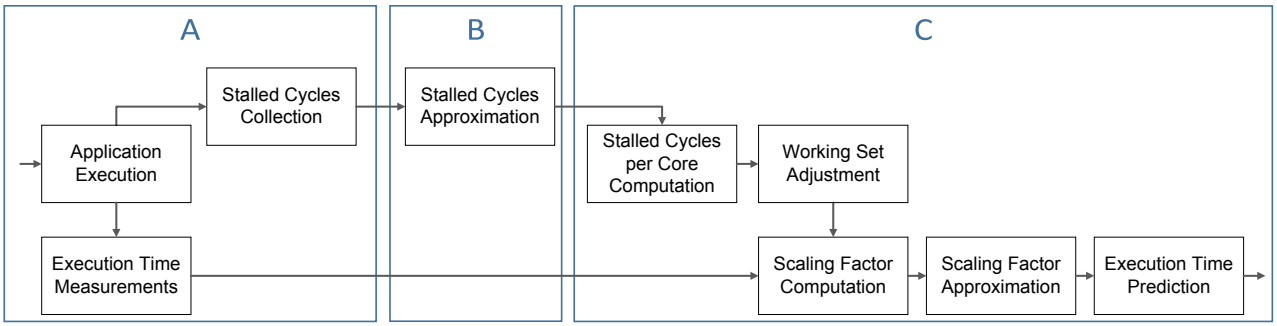


Figure 3: Constructing extrapolations with ESTIMA.

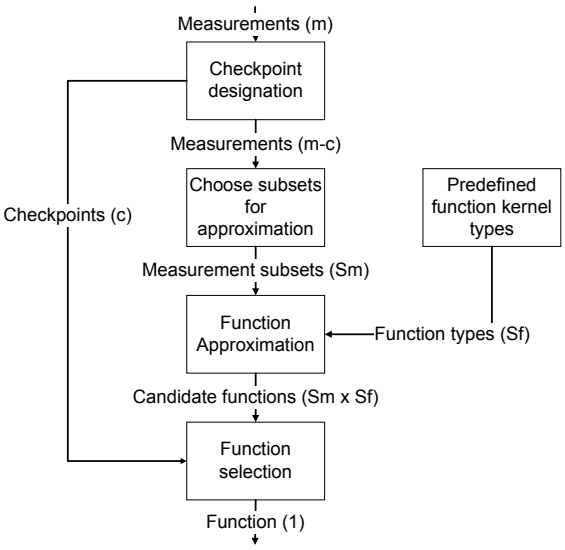


Figure 4: The regression analysis of ESTIMA.

B. 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 stalled cycle category measured. 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:

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 a function from the predefined kernels in Table 1. These functions are used based on the approximation library used (see Section 4), 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 also considered.
4) ESTIMA chooses the function that minimizes the error and uses it to approximate the stalled cycle values.

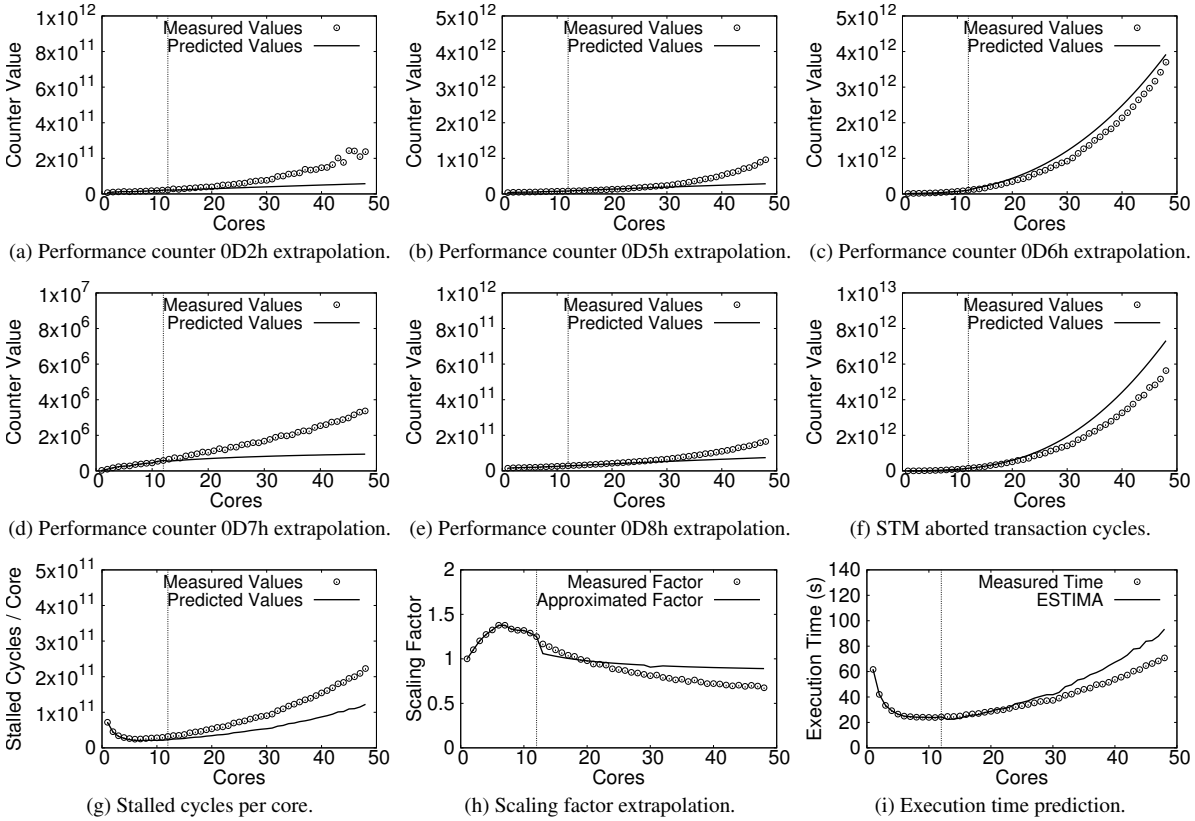
Name	Function
Rat22	$\frac{a_0 + a_1 n + a_2 n^2}{1 + b_1 n + b_2 n^2}$
Rat23	$\frac{a_0 + a_1 n + a_2 n^2}{1 + b_1 n + b_2 n^2 + b_3 n^3}$
Rat33	$\frac{a_0 + a_1 n + a_2 n^2 + a_3 n^3}{1 + b_1 n + b_2 n^2 + b_3 n^3}$
CubicLn	$a + b\ln(n) + c\ln(n)^2 + d\ln(n)^3$
ExpRat	$e^{\frac{a+bn}{c+dn}}$
Poly25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

Table 1: Extrapolation function types.

At this step, ESTIMA has created functions that approximate the values of the hardware and software stalled cycles and can use them to calculate the stalls for higher core counts.

C. 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 of hardware and software stalled cycles. In order to adjust predictions to different workload sizes, ESTIMA adjusts the stalled cycles calculated using the ratio of the measures and target dataset sizes.

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 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 with the total stalled cycles per core. The reason is the following: we argue that execution time and stalled cycles have a very high correlation. As such, the produced execution time values should retain high correlation with the total stalled cycles per core that were calculated in the previous step.



(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.

(g) Stalled cycles per core. (h) Scaling factor extrapolation. (i) Execution time prediction.

Figure 5: `intruder` prediction example.

After the factor function has been created, ESTIMA uses the stalled cycles per core, as approximated in step (B), and the scaling 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 [28] 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 Design 5 of the NIDS described by Haagdorens et al. in [14]. Network packets are processed in parallel and go through three phases: capture, reassembly, and detection. In the version that uses transactional memory included in STAMP, the capture and reassembly phases are each enclosed in 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). We take measurements using only one processor of the machine (12 cores) and target all four processors with our predictions. Hence, the measurements machine is a 12-core machine, while the target machine is a 48-core machine. We also execute the application on the target machine and measure the stalled cycles and execution time for up to 48 cores. We present these measurements alongside our extrapolations 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 for the existing measurements of each stall. With one function for each stalled cycle category, ESTIMA can extrapolate the stall values for higher core counts. In Figures 5a-f we present the result of this process. ESTIMA uses the measurements 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. It is important to notice here that even though execution time and stalled cycles per core are decreasing for fewer than 12 cores, the fine-grain stalled cycles are increasing. Thus, the result of the extrapolation of these stalls helps predict the slowdown of the application for higher core counts. `intruder` also showcases why ESTIMA uses fine-grain stalls instead of an aggregate event. By examining the aggregate backend stall values in Figure 5g, we notice that for measurements with up to 12 cores, the

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

Table 2: Hardware performance counters used for the *Opteron* machine.

slowdown of the application is not visible. As a result, if ESTIMA simply extrapolated these values, it would fail to capture the behavior of `intruder` for higher core counts, similarly to the time extrapolation method.

After the performance counter values have been approximated, ESTIMA computes the stalled cycles per core, shown in Figure 5g. 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 with the stalled cycles per core. The approximation of the scaling factor is presented in Figure 5h. 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 5i. ESTIMA successfully predicts the scalability of the application and the slowdown it exhibits for higher core counts.

4. Implementation

We implement ESTIMA in Python. We integrate the functionality in a single, easy-to-use tool. For the function approximation, we use the `pythonequation` library from [33] 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, but extending it to support additional families of processors is straightforward. By default, the user needs to specify only the input of the application.

In order to improve the accuracy of the predictions, ESTIMA enables the use of *plugin* components. The user can specify additional categories of stalled cycles at the software level that can be used for the predictions. ESTIMA takes a configuration file that includes the path to the file the stalls are reported to (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. The way ESTIMA collects software stalls can vary between applications. In our evaluation, for the collection of synchronization overheads and spinning times we use a thin wrapper around the `pthread` library. For the applications that use transactional memory, we use SwissTM [11] with detailed statistics enabled, which reports the duration of committed and aborted transactions.

5. Evaluation

5.1 Evaluation setup

We evaluate ESTIMA using four 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 4-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 system as *Opteron*. We also use two *Intel Xeon* machines. The first has 2 Intel Xeon E5-2680 v2 processors with 10 cores each, clocked at 2.80GHz (40 threads in total). The second machine has 2 Intel Xeon E5-2680 v3 processors, with 24 cores each, clocked at 2.5GHz (48 threads in total). We refer to them as *Xeon20* and *Xeon24* respectively.

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. The STM workloads use SwissTM [10, 11].

We start with two production applications and cross-machine predictions using ESTIMA. We then conduct experiments using three benchmark suites. We conduct both strong and weak scaling experiments. We show how we use ESTIMA to pinpoint the bottlenecks in two applications and guide us towards fixing them. Finally, we discuss ESTIMA's strengths and shortcomings.

5.2 Extrapolating to different machines

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

In our first experiment, we use ESTIMA to predict the scalability of a `memcached` server. We use ESTIMA on the desktop Haswell machine and target the *Xeon20* server machine with our predictions. We run the clients on the same machine as the `memcached` server, as we do not want to take network performance into account. The client and dataset are the ones provided by `cloudsuite` [13], scaled to 10x the original size. 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.

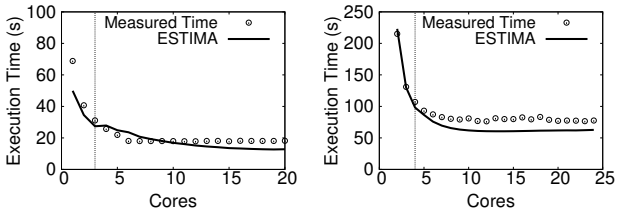
We use all 8 threads of the machine, letting the operating system do the scheduling of the threads. ESTIMA collects stalled cycles and execution time from the `memcached` server using up to 3 cores of the desktop machine and extrapolates its performance to a machine 8 times its size. The performance 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 40 hardware threads. We keep the threads on the same processor when possible. The result of the prediction produced by ESTIMA, as well as the time measured from the *Xeon20* machine are presented in Figure 6a. ESTIMA successfully predicts the scalability of the application. The absolute errors are 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 the *Xeon24* server with our predictions. We use *tmpfs* to avoid IO bottlenecks for logging and dedicate threads on the desktop machine to the SQLite. We pin the client threads to the rest of the hardware threads. ESTIMA collects stalled cycles and execution time from the `SQLite` process for up to 4 threads of the desktop machine and extrapolates its performance to

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

Table 3: Hardware performance counters used for the latest *Intel* processors.



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

Figure 6: Predictions for `memcached` and `SQLite`.

a machine 6 times its size. The hardware stalls are the same as in the previous experiment. Similarly, we scale the execution frequencies of the two machines for the predictions.

We then measure the execution time of the workload on the server machine, using all 48 hardware threads. 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 6b. 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, while predicting for a machine with 6 times more cores.

ESTIMA is successful in predicting the scalability of both applications, using measurements on a significantly smaller machine, from a different family of processors.

5.3 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 3 suites of in-memory benchmarks: STAMP [28], Parsec [5] and standard STM 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 4 and 2 CPUs respectively (the full machines).

In Table 4, we present the summary of our prediction errors for both machines. The errors presented are the maximum errors observed when using one processor from the respective machine and predicting for 2, 3 and 4 processors of the *Opteron* machine (13-24, 25-36 and 37-48 cores) and 2 processors of the *Xeon20* machine (11-20 cores). For brevity, we discuss the strengths of ESTIMA as well as its limitations, using predictions for the *Opteron* machine, for which we have predictions for higher core counts. For comparison purposes, we also implement a version of the time extrapolation method. We collect execution time measurements and use the same function kernels to approximate these measurements. We include the results of this process in the prediction figures and call it *time extrapolation*. 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 most workloads with small prediction errors. Out of 19 workloads used for the evaluation of ESTIMA:

	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
intruder	9.2	22.1	31.9	30
kmeans	50.3	50.9	17.0	30
labyrinth	15.4	15.0	18.4	10
ssca2	2.8	4.6	8.1	21.4
vacation-high	14.7	14.3	10.3	17
vacation-low	18.9	18.5	25.0	10
yada	8.1	23.0	15.1	40
parsec:				
blackscholes	3.7	4.4	2.9	14
bodytrack	1.3	3.0	5.9	8
canneal	10.7	12.4	8.3	6
raytrace	2.7	3.6	4.6	1
streamcluster	15.6	59.0	88.8	20
swaptions	10.6	14.7	20.3	9
K-NN	11.5	22.5	32.0	13

Table 4: Maximum prediction errors with measurements on 1 processor of each machine (12 cores for *Opteron* and 10 cores for *Xeon20*).

- For all extrapolations performed, ESTIMA was successful in predicting the scalability of the applications. There were no cases where ESTIMA would incorrectly predict that an application would or would not scale.
- When extrapolating to the *Xeon20* machine with double the number of cores than used for measurements, 15 workloads have execution time prediction errors lower than 25% and 9 of them have errors lower than 10%.
- When extrapolating to the *Opteron* machine with four times the number of cores than used for measurements, 16 workloads have execution time prediction errors lower than 25% and 9 of them have errors lower than 10%.

In Figure 8a we present an example of a prediction result, using `raytrace` from the PARSEC benchmark suite. `raytrace` is an Intel RMS application which 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 ex-

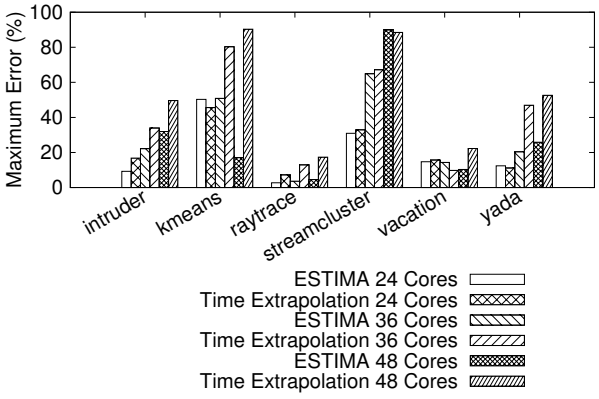
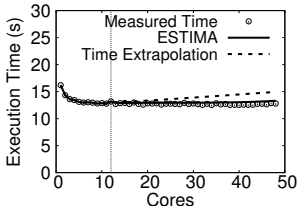
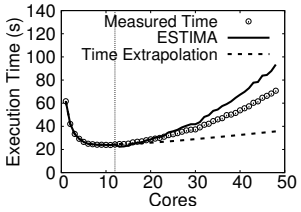


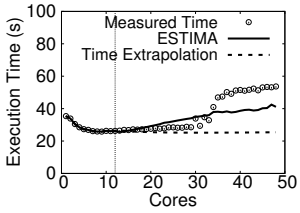
Figure 7: Comparison of errors between `ESTIMA` and time extrapolation.



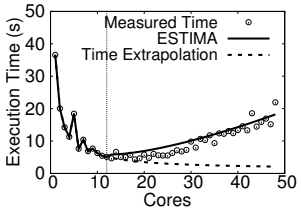
(a) `raytrace` execution time prediction.



(b) `intruder` execution time prediction.



(c) `yada` execution time prediction.



(d) `kmeans` execution time prediction.

Figure 8: Predictions using `ESTIMA`.

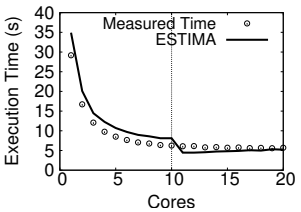
ecution time prediction error observed for predictions for up to 48 cores being 4,6%. This is in contrast to the time extrapolation method, which produces errors up to 17,3%.

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 8b and 8c. `intruder` has already been introduced in Section 3.2. `yada` implements Ruppert's algorithm for Delaunay mesh refinement [35]. 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 limits of 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 their scalability trends and has significantly higher prediction errors (up to 81% and 130% higher for `intruder` and `yada` respectively).

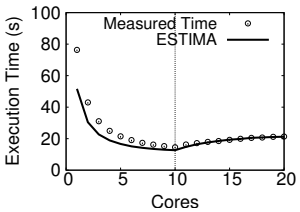
Another interesting example is `kmeans` from the STAMP suite. `kmeans` is a partition-based clustering benchmark that represents a cluster by the mean value of all objects contained in it. We present a scalability prediction for `kmeans` on the *Opteron* machine in Figure 8d. Although the absolute maximum error for `kmeans` is higher in absolute numbers in Table 4, the prediction is a successful one. The high error value is the result of the fluctuations in `kmeans`' execution time for different core counts, which the prediction does not follow. Nevertheless, ESTIMA successfully predicts the performance of the application. As with `intruder` and `yada`, the scalability degradation of `kmeans` is not evident in the original measurements. ESTIMA can successfully capture the trends in stalled cycles and predict the performance degradation, something time extrapolation is unable to do.

5.4 Weak scaling

Using a larger machine commonly means running bigger workloads. This is due to the larger amount of memory usually installed in larger machines. We evaluate how we can use ESTIMA in such a scenario. We use measurements on a machine and predict the scalability of the application on a machine with double the number of



(a) `genome` execution time prediction



(b) `intruder` execution time prediction

Figure 9: Predictions with changing workload sizes.

cores and with a twice as large dataset. We use two applications: `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.

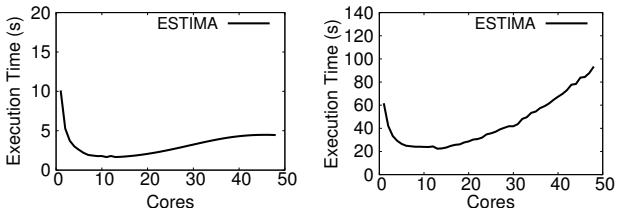
We use ESTIMA on one processor of the machine for each application, targeting a machine with double the number of cores (the full machine). We configure ESTIMA with a target workload size that is two times the size used for measurements. By measuring the memory footprint of the application and scaling its predictions accordingly, ESTIMA produces predictions for both applications on the target machine and for the target workloads.

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 most significant errors appear for single core performance of `intruder` on the bigger machine. The maximum errors (excluding single core performance) are 28% for `intruder` and 29% for `genome`. Although with higher errors than their strong scaling counterparts, these predictions show that with a simple technique ESTIMA is capable of accounting for changes in workload sizes.

5.5 Identifying future bottlenecks

Throughout this paper, we present how ESTIMA can be used to extrapolate stalled cycles in order to predict the scalability of an application. A question that arises is the following: *can we use this knowledge to help developers identify bottlenecks in their applications, before they even appear?* We now show how we can use ESTIMA to identify bottlenecks in two applications, as well as how we fix these bottlenecks. We use two applications that use different concurrency control mechanisms: `streamcluster` and `intruder` from the PARSEC [5] and STAMP [28] suites respectively. For both applications we collect both hardware and software stalls. 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.

We use ESTIMA to extrapolate the scalability of the two applications to the *Opteron* machine. We use measurements on one processor of the machine (12 threads) and extrapolate to all four processors (48 threads). We show the results of these extrapolations in Figure 10. Both application exhibit slowdown for high core counts. We configure ESTIMA to report the intermediate extrapolations of individual stalled cycle categories and report the ones that contribute most to stalls. We then use the *perf* linux tool to pinpoint the most significant sources of the reported stalls. For `streamcluster`, we identify the source of a significant part of hardware stalled cycles in the *pthread_mutex_trylock* function used for the custom PARSEC barriers. This leads us to identify the mutexes used as a potential scalability bottleneck. For `intruder`, we similarly identify



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

Figure 10: Predictions for `streamcluster` and `intruder` using ESTIMA.

the main source of stalled cycles to be aborted STM transactions in the *processPackets* function, and more specifically in the *TMDECODER_PROCESS* call. By examining the code of the function, we understand that aborted transactions are the result of contention for a shared data structure.

For `streamcluster`, we replace the standard pthread mutexes used by PARSEC with test-and-set spinlocks, based on our findings. For `intruder`, we modify the application to decode more elements in every step, contrary to the original implementation. The measurements on the full *Opteron* machine validate our findings. The performance of both modified applications significantly improves. We show the original and modified applications' performance in Figure 11. For `streamcluster`, ESTIMA helps us improve its execution time by up to 74%. Similarly, we improve `intruder`'s performance by up to 70%.

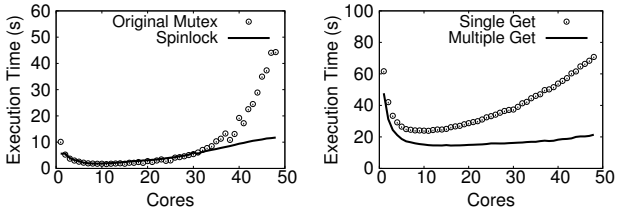
5.6 Discussion

Software stalled cycles. ESTIMA by default uses hardware stalls for its extrapolations. However, it can be configured to include software stalls to further improve the accuracy of its predictions.

For our experiments, 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, this effect of software stalls is expected: contention for shared resources leads to aborted transactions, which are transparent to the hardware. Useful work at the hardware level is discarded when a transaction is aborted.

Moreover, we experiment with measurements of synchronization cycles using a thin wrapper around the `pthread` library. This wrapper measures cycles that threads spend spinning on 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 12 we present five applications for which software cycles significantly improve ESTIMA's predictions. We show the prediction errors for the applications with and without software cycles. Using software cycles for these applications improves prediction



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

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

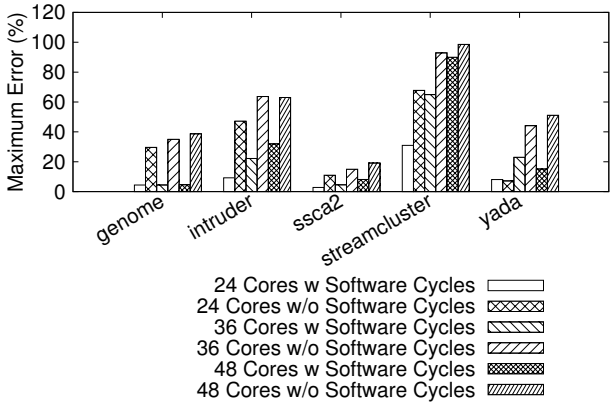
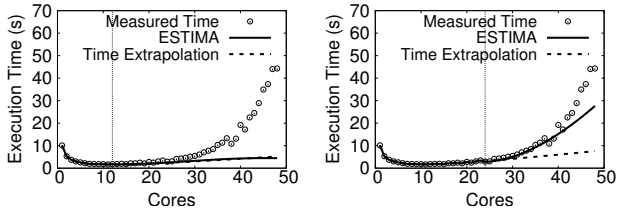


Figure 12: Comparison of prediction errors with and without software stalled cycles.

accuracy by 57% on average, and for `genome` by up to 87% for a machine with four times the number of cores.

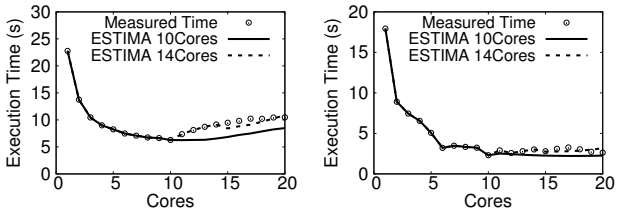
Limitations. Table 4 shows that predictions for `streamcluster` from the PARSEC benchmark suite exhibit high absolute prediction errors. `streamcluster` is a clustering benchmark, which, for a stream of input points, finds a predetermined number of medians, so that each point is assigned to its nearest center. We show both the extrapolated and measured execution time of the application in Figure 13a. 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 (2 sockets of *Opteron*), the prediction is significantly better, as seen in Figure 13b. 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.

Similarly, due to its nature, ESTIMA does not capture effects that are not present in the measurements machine, such as NUMA effects. This is the reason behind the higher prediction errors for the *Xeon20* machine. In contrast, the architecture of the *Opteron* machine enables ESTIMA to account for NUMA effects. Each processor of the *Opteron* machine contains two chips, introducing NUMA effects even for measurements using only one processor. By accounting for these effects, ESTIMA has higher accuracy for extrapolations to significantly higher core counts. The prediction results



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

Figure 13: Predictions for `streamcluster`.



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

Figure 14: Predictions with NUMA effects captured.

are similar for the *Xeon20* machine when using more than 10 cores for the measurements. We show two such examples in Figure 14. By capturing the discontinuities in the measured cycles, ESTIMA achieves higher accuracy in its predictions.

It is important to note that ESTIMA's main use case involves predictions for machines with similar architectures. ESTIMA successfully predicts the performance of an application across such machines. 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 in general result in less accurate predictions. 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.

6. Related work

The work that is most closely related to ESTIMA is that of Crovella et al [8], in which the authors use productive and stalled cycles collected at the software level to predict the performance of an application. In contrast, ESTIMA relies primarily on hardware stalls, using performance counters offered by modern hardware. ESTIMA can leverage software-level stalls to further improve its extrapolations, applying 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. The use of linear logarithmic functions results in accurate predictions as the workloads scale almost linearly, unlike the workloads that ESTIMA targets. ESTIMA targets applications that use shared data, and utilizes hardware and software stalls to extrapolate the scalability of parallel in-memory applications.

In [25], statistical techniques and regression are used to build piecewise polynomial and neural network black-box models of scientific programs. Neural networks are used as well in [21] to build models of SMG2000 applications on two different large-scale machines. Unlike ESTIMA, these models do not address the question of application's scalability with more CPUs than used in the measurements.

In [31], the author extrapolates address stream profiles 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 itself, identifying bottlenecks in the process.

Several systems combine the predictions of sequential performance of single-node tasks performed by distributed cores with the model of communication between them [6, 27, 44]. 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 [42]. 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 [32] and Queueing theory [32]. 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. [39] use a simple cost-benefit analysis to choose between locking and transactions. The performance model from [34] focuses on modeling transactional conflict behavior. Unlike ESTIMA, this approach requires heavy instrumentation of the applications in order to collect the statistics of memory accesses.

Performance counter research has mainly focused on profiling of applications and identifying performance bottlenecks. In [37] and [40], the authors use performance counters to increase power efficiency, through thread scheduling and placement. In [41], the authors use performance counters to capture performance impacts as a function of resource usage. In [43], 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. [38] use hardware performance counters to identify scalability bottlenecks in parallel applications running on Distributed Shared-Memory multiprocessors. Finally, in [23], the authors 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 used for scalability predictions, as in ESTIMA.

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 visualize the scalability of applications with minimum effort. To achieve that, ESTIMA uses stalled cycle measurements in hardware and optionally in software, and regression analysis 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 its accuracy.

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 four times larger than the one available for measurements were lower than 15% for more than half of the applications, and ESTIMA successfully captured the scalability of all the applications used. Finally, we used ESTIMA to identify bottlenecks in parallel applications, showing how ESTIMA can be useful to developers.

Acknowledgments

We wish to thank our shepherd, Alexandro Baldassin, and the anonymous reviewers for their helpful comments on improving the paper. We would also like to thank the members of the Distributed Programming Laboratory at EPFL for the insightful discussions. Part of this work was supported by the European Research Council (ERC) Grant 339539 (AOC).

References

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

[2] A. R. Alameldeen and D. A. Wood. Ipc considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4), 2006.

[3] AMD. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors, 2010.

[4] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. ICS '08. ACM, 2008.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*, 2008.

[6] L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3), 2006.

[7] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of spmd codes using expectations. ICS '07. ACM, 2007.

[8] M. E. Crovella and T. J. LeBlanc. Parallel performance prediction using lost cycles analysis. IEEE Computer Society Press, 1994.

[9] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. SIGMOD '13. ACM, 2013.

[10] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why stm can be more than a research toy. *Commun. ACM*, 54(4), Apr. 2011.

[11] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI '09*, New York, NY, USA, 2009. ACM.

[12] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. NSDI'13. USENIX Association, 2013.

[13] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ASPLOS '12*. ACM, 2012.

[14] B. Haagdorens, T. Vermeiren, and M. Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *WISA'04*, 2005.

[15] A. Heindl and G. Pokam. An analytic framework for performance modeling of software transactional memory. *Computer Networks*, 53(8), 2009.

[16] A. Heindl and G. Pokam. An analytic model for optimistic stm with lazy locking. In *Analytical and Stochastic Modeling Techniques and Applications*. Springer, 2009.

[17] A. Heindl, G. Pokam, and A.-R. Adl-Tabatabai. An analytic model of optimistic software transactional memory. In *ISPASS 2009*. IEEE, 2009.

[18] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93*. ACM, 1993.

[19] 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(4), 2000.

[20] I. Intel. and ia-32 architectures software developers manual volume 3b: System programming guide. *Part*, 1, 2007.

[21] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Euro-Par 2005 Parallel Processing*. Springer, 2005.

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

[23] V. Jiménez, F. J. Cazorla, R. Gioiosa, M. Valero, C. Boneti, E. Kursun, C.-Y. Cher, C. Isci, A. Buyuktosunoglu, and P. Bose. Power and thermal characterization of power6 system. In *PACT '10*. ACM, 2010.

[24] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *SC2001*. ACM, 2001.

[25] 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 *PPoPP '07*. ACM, 2007.

[26] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. NSDI'14. USENIX Association, 2014.

[27] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32. ACM, 2004.

[28] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[29] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. NSDI'13. USENIX Association, 2013.

[30] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. PACEa toolset for the performance prediction of parallel and distributed systems. *International Journal of High Performance Computing Applications*, 14(3), 2000.

[31] C. R. M. Olschanowsky. *Hpc Application Address Stream Compression, Replay and Scaling*. PhD thesis, La Jolla, CA, USA, 2011.

[32] C. A. Petri. Communication with automata, new york: Griffiss air force base. Technical report, Tech. Rep. RADC-TR-65-377, 1966.

[33] J. R. Phillips. Zunzun.com. `http://www.zunzun.com`.

[34] D. E. Porter and E. Witchel. Understanding transactional memory performance. In *ISPASS 2010*. IEEE, 2010.

[35] J. Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of algorithms*, 18(3), 1995.

[36] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95*. ACM, 1995.

[37] K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, 37(2), July 2009.

[38] J. Torrellas, Y. Solihin, and V. Lam. Scal-tool: Pinpointing and quantifying scalability bottlenecks in dsm multiprocessors. In *Supercomputing, ACM/IEEE 1999 Conference*, Nov 1999.

[39] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. In *PACT '09*. IEEE Computer Society, 2009.

[40] A. Vega, A. Buyuktosunoglu, and P. Bose. Smt-centric power-aware thread placement in chip multiprocessors. In *PACT '13*. IEEE Press, 2013.

[41] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. In *PACT '10*. ACM, 2010.

[42] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *SC2005*. IEEE, 2005.

[43] A. Yasin. A top-down method for performance analysis and counters architecture. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, March 2014.

[44] J. Zhai, W. Chen, and W. Zheng. PHANTOM: predicting performance of parallel applications on large-scale parallel machines using a single node. In *ACM Sigplan Notices*, volume 45. ACM, 2010.