Predicting the Cost of Lock Contention in Parallel Applications on Multicores using Analytic Modeling

Xiaoyue Pan
Department of Information Technology, Uppsala University
xiaoyue.pan@it.uu.se

Jonatan Lindén
Department of Information Technology, Uppsala University
jonatan.linden@it.uu.se

Bengt Jonsson
Department of Information Technology, Uppsala University
bengt.jonsson@it.uu.se

ABSTRACT

The scalability of a parallel program is limited by several factors, such as the size of its non-parallelizable fraction, the amount of contention for shared resources, the cost of synchronization, etc. We consider the problem of predicting the cost of contention for shared locks. We propose a way to address this problem with analytical techniques using queueing network models.

Analytical models allow to efficiently compute performance estimates over a wide range of parameter values. As a main contribution, we evaluate the accuracy of our analytical model in predicting the cost of lock contention. The evaluation is partly carried out on a set of benchmarks, some of which are intended to mimic common parallel programming patterns, and a realistic one, viz. the *dedup* benchmark in the PARSEC benchmark suite.

1. INTRODUCTION

With the introduction of multi-core machines, scalability becomes important for any parallel program. To study the scalability of a program, we need to understand the factors that prevents the program from scaling linearly; such as hardware contention (e.g., cache, memory, bus), software contention (e.g., for locks and buffers), parallelization overhead, and work load imbalance. In this paper, we focus on understanding lock contention.

In a parallel program, the use of shared data is typically protected by locks to guarantee exclusive access. If several threads try to acquire the same lock, only one thread at a time can succeed and the others must typically wait instead of doing useful work. Thus locks incur a contention cost.

As an intuitive explanation of how lock contention impacts parallel execution time and scalability, consider the example shown in Figure 1. This is an example of parallel execution with four threads running in parallel. All four threads contain an indefinitely parallelizable part (in light grey) and a critical section protected by a single lock. In Figure 1a, all threads try to acquire the shared lock at the same time, and three of them have to wait. In Figure 1b, the threads try to acquire the lock at different points in time, resulting in no lock contention. This example clearly shows how lock contention can affect parallel execution time.

We consider the problem of predicting the cost of lock contention in parallel programs. To understand if lock contention is a potential scalability problem, we must predict how much time is lost in lock contention when the program runs on a platform with an increasing number of cores. Often, we would like to predict the contention cost with modest

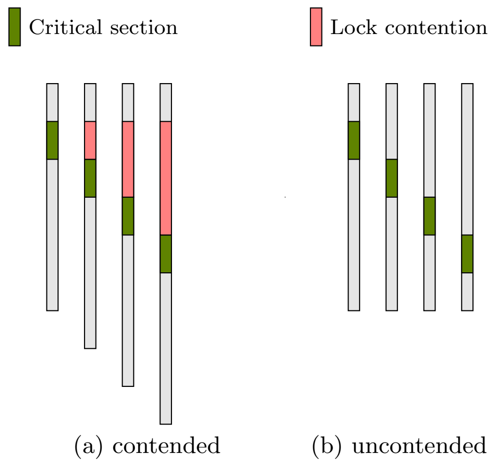

Figure 1: Serialized and parallel critical sections

effort (e.g., by inspection or a modest number of profiling executions) without having access to such a platform. In order to do this, we need an evaluation method which can provide reasonable answers with modest effort. Analytical models offer the possibility of efficiently computing performance estimates over a wide range of parameter values (e.g., number of cores). They allow to compute a range of performance indices that can shed light on various scalability issues. The key issue is to devise a model which is sufficiently accurate and still tractable for analysis.

In this paper, we propose a queueing network model and investigate how accurately such a model can predict the cost of lock contention. Our approach is to first extract the lock access behavior of each thread, in which the frequency and length of each lock access is represented. Such information can be obtained by profiling an execution of the program on a single (or small number of) cores with existing profiling tools. We thereafter construct an analytical queueing network model, which abstractly represents the lock access behavior with any number of threads and cores. The queueing network can then be analyzed with existing techniques to obtain a measure of the lock contention.

Our model can be used to predict if the lock contention will be a scalability bottleneck with more threads. It can also be used to evaluate the design of the lock usage in the program; for example, if a program does not suffer from lock contention with a big number of threads and coarse grained locks, there is no need to implement a finer-grained lock mechanism. Previous work evaluated their analytic models with simulation results. Sorin et al. [13] modeled the IPC of an application and compared their results only with simulation results. To our knowledge, this paper is the first to compare the result of lock contention of an analytical model to the real execution on real hardware. To summarize, we make the following contributions in this paper:

- we propose a method to model the structure of lock accesses in a parallel program using an analytical queueing network, and
- we evaluate the accuracy of our analytical model in predicting the cost of lock contention for parallel programs using a set of benchmarks.

Organization: The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 introduces our target programs. Section 4 discusses our analytical model. Section 5 presents the evaluation of our model and section 6 concludes the paper.

2. RELATED WORK

Eyerman et.al. [5] discuss how Amdahl's law can take critical sections into account. They present a probabilistic model for the critical sections in parallel programs, with simplifying assumptions, e.g., that all threads execute the same code and that locks are accessed uniformly. A main conclusion is that the impact of critical sections in a program can be included in Amdahl's law by splitting it into one fraction which is attributed to the sequential part, and one fraction which is attributed to the parallel part. The sequential part is proportional to the probability of entering the critical section multiplied by the contention probability. The aim is to give guidelines to future micro-architectural multi-core design, but the authors do not validate the probabilistic model with real benchmarks.

Tallent et al. [14] measure and quantify lock contention using sampling. They suggest three different strategies for giving insight to the programmer into which parts of the program should be "blamed" for observed lock contention, and should hence be modified to improve performance. The authors do not predict how lock contention may affect scalability to an increasing number of cores.

Sorin et al. [13] address the problem of predicting the IPC of an application given a small set of parameters about the architecture and program (which are obtained from fast simulation). They propose queueing networks models for describing the effects of lock contention and memory contention. The interdependence between these two effects is captured by coupling the two queueing network models and using an iterative approach when solving them. The results are validated against an existing detailed simulator for such a system, not against executions on actual multi-core platform as in our work. Although queueing theory is used, the purpose of their work is to evaluate different memory architectures on an SMP system while the purpose of ours is to predict the scalability of locks with more threads for a parallel application.

Navarro et al. [9] model the performance of pipeline parallelism programs with a queueing network model. Each pipeline stage is modeled as an n-server node where n is the number of threads assigned to the corresponding pipeline stage. Data items are jobs in the queueing system. This model describes the load imbalance between threads, and is not concerned with lock contention per se.

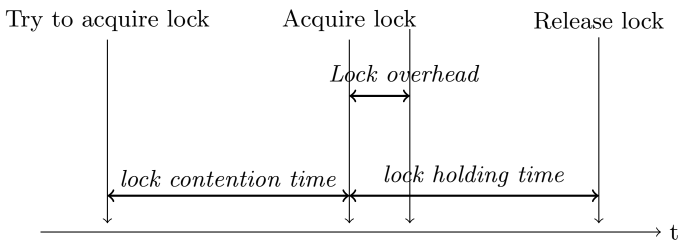

Figure 2: Lock holding time and lock contention

Adve et al. [1] mention the possibility of using a queueing network model for different kinds of overheads caused by shared resources as a sub-model for their task graph analysis. However, they do not present an evaluation of the model's accuracy.

3. PROGRAM STRUCTURE

We consider multi-threaded programs that synchronize accesses to shared data using locks, executing on a multi-core platform.

The terminology used in this paper is presented in Figure 2. When a thread tries to acquire a lock, it will succeed only if the lock is free, otherwise the thread must wait until the lock is eventually released. After finishing its operation, the thread releases the lock. The *lock holding time* is the length of the time period from *Acquire lock* to *Release lock*. The *lock contention time* is the length of the time period from *Try to acquire lock* to *Acquire lock*.

The lock holding time may vary between different accesses to the same lock by different threads, but we assume that it can be reasonably modeled by some probability distribution. Typically, some *lock overhead* is associated with acquiring the lock, which during a short period will prevent useful work from being performed. We measure this lock overhead for different kinds of locks and account for it in our analysis.

We make the following assumptions about program structure:

- The number of threads in the program is constant during each execution and the threads execute on dedicated cores.
- The set of threads can be partitioned into a finite number of classes, such that the threads within each class have the same lock access pattern. Threads of different classes may access different locks or access locks in different orders.
- Lock accesses are not nested. We investigated into the PARSEC benchmark, which is a state-of-the-art multi-threaded program benchmark suite, to check how common nested locks are. None of these programs use nested locks. Therefore we have reasons to believe that nested locks are not common in a large class of multi-threaded programs.
- The lock holding time does not change with the number of threads in the program. In reality, the lock holding time may sometimes change (typically increase)

with more threads due to increased competition for shared resources, such as memory bandwidth. In this paper, we assume that these effects and their influence on lock holding time are understood, and we focus on the effects of lock contention.

- The program execution is sufficiently repetitive to achieve and maintain a steady state in which the lock access patterns of the threads are stable.

Example 3.1.

As an illustration, consider the program skeleton in Figure 3. All threads in the program execute the same loop, in which a thread first performs local computation for t_P, and then acquires one of two locks to perform operations on some shared data: the choice of lock is decided by the test <*condition*>. It is assumed that the time periods spent in local computation, and in computing while holding the lock (i.e., t_P, t_1, and t_2, respectively) can be represented as stochastic variables with some probability distribution. We use $\overline{t_P}$, $\overline{t_1}$ and $\overline{t_2}$ to denote their mean values.

```
Skeleton Program
loop
          Compute for $t_P$
          if<condition>
                         Acquire $lock_1$
                         Compute for $t_1$
                         Release $lock_1$
          else
                         Acquire $lock_2$
                         Compute for $t_2$
                         Release $lock_2$
endloop
```

Figure 3: Example program skeleton: executed by all threads

When the program contains only one thread, there is no lock contention. With an increasing number of threads, the lock contention time is expected to increase. □

4. ANALYTIC MODELING

Queueing networks can model systems where jobs access resources according to some pattern. A queueing network consists of a set of *nodes* (aka. service stations). Jobs (aka. customers) travel between nodes at which they receive "service". A node can either serve one job at a time (*single-server node*) or some specified number of jobs concurrently. When a job arrives to a node which is currently serving a maximal number of jobs, it must wait in the queue of the node until currently served jobs have left the node. The flow of jobs between nodes is defined by a *routing matrix R*, in which the entry $R_{i,j}$ specifies the probability that a job visits node j after having been served at node i. We consider *closed* networks, in which the number of jobs is constant (i.e., jobs do not enter or leave the network). This is because the number of threads is constant in our target programs.

We model a parallel program as a closed queueing network, in which the number of jobs is the number of threads. Each shared lock is modeled as a single-server node, where "service" corresponds to the execution in the critical section, and where the service time is the time spent in the critical section. Local computation by a thread outside any critical section is modeled as a visit to an *infinite-server node* at which queueing is never needed: its service time is the time for local computation. We use the routing matrix R to describe the lock access patterns (the probability of visiting one node after leaving another node). In summary, our model takes these parameters:

- the routing matrix R: an $n \times n$ matrix to describe the probability of visiting one node after another, where n is the number of nodes in the network,
- the number of jobs (K) in the network, and
- the service time at each node: this may depend on the job classes.

The performance metrics of interest are the lock contention time, i.e., mean queueing time in the one-server nodes of the queueing network, and the average queue length, i.e., average number of threads contending for the lock.

Two important methods for solving closed queueing networks are the convolution algorithm [11] and Mean Value Analysis (MVA) [12]. When the service times are exponentially distributed, they both give exact solutions. The convolution algorithm and MVA are derivable from each other [6], and they have the same computational complexity. The main advantage of MVA is the numerical stability of its computations.

We remark that it is easier to use the Stochastic Petri Nets [8] to model synchronization. However, for the examples that we consider here, their state spaces would be so large as to make any analysis intractable; for example, for a simple program with 48 threads sharing two locks where each thread accesses the locks sequentially, a Stochastic Petri Net model has 20825 reachable states. To solve a markov chain with this number of states, we need to solve a linear equation system of 20825 states, which is too complex for such a small problem.

Example 3.1 (ctd.).

The example program can be modeled by the queueing network in Figure 4. We only need one class in this model. The nodes $node_1$ and $node_2$ are single-server nodes, corresponding to $lock_1$ and $lock_2$, respectively, whereas the node $node_0$ is an infinite-server node. The service times at node $node_1$ and $node_2$ are the lock holding times t_1 and t_2 (in milliseconds), respectively, and the service time at node $node_0$ is t_P. To define the routing matrix R, we assume that <*condition*> is satisfied in each evaluation with a probability of 40%. Ordering the nodes as $node_0, node_1, node_2$, the routing matrix R is as follows.

$$R = \begin{pmatrix} 0 & 0.4 & 0.6 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

If $\overline{t_P} = 3\ ms$, $\overline{t_1} = 1\ ms$ and $\overline{t_2} = 2\ ms$, we can calculate that the contention at $lock_1$ increases from 0 ms to 0.5 ms and the contention at $lock_2$ increases from 0 ms to 120 ms when the number of threads increases from 1 to 64 by solving the queueing network.

4.1 Modeling thread heterogeneity

In a parallel program, it is often the case that not all threads execute the same code. A typical example is a

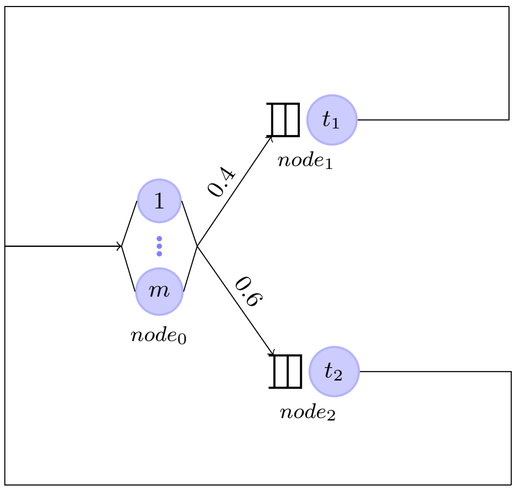

Figure 4: A simple closed queueing network

pipelined program with several parallel pipeline stages, to each of which a number of threads are assigned. Threads of different stages may access different locks or access the same locks in a different order. In this case, threads of the different stages have different routing matrices. The service time of jobs of different stages at the nodes may also differ, thus being *class-dependent*. We can model such a program with a multi-class queueing network where each class represents a pipeline stage. To solve such a network, we use the multi-class variant of the Mean Value Analysis [12].

4.2 Extracting model parameters from a run of the program

To apply our model to a parallel program, we need to extract the parameters for the queueing network, viz. the set of locks, corresponding to nodes, the routing matrix (or matrices), and the service time at each node for each class of threads.

We run the target program sequentially, on a single core, to extract the parameters for the queueing model. We used a modified version of Mutrace [10], which instruments the Pthreads library. It collects timestamps for *Try to acquire lock*, *Acquire lock* and *Release lock* for different locks and different threads. We then sort the timestamps for each thread. The lock holding time is the difference between the *Release lock* timestamp and the *Acquire lock* timestamp. The time spent in local computation is calculated as the difference between the *Try to acquire lock* timestamp of the second lock with the *Release lock* timestamp of the first lock. The extracted parameters serve as the input to the queueing network.

4.3 Potential sources of imprecision

There are several potential of sources of imprecision for comparing our model's results with those obtained in actual executions. Some are caused by a discrepancy between idealizations of the model and the properties of actual applications, including the following.

- In our model, lock holding times are typically assumed to be exponentially distributed, whereas actual lock holding times may follow arbitrary probability distributions. If the coefficient of variation (cv) of the lock holding time is less than 1, which is the cv of an exponential distribution, the model will typically overestimate the lock contention time. We show an example of how to obtain precise predictions for uniformly distributed service times for a simple network in Section 5.2. For other examples, we will use exponentially distributed lock holding times for simplicity.

- The lock holding time may vary as the number of threads increases, due to increased competition for shared resources, such as memory bandwidth. In cases where this effect is significant, we measure (rather than predict) the increased lock holding times, in order to isolate the effect of lock contention and understand how well the lock contention can be predicted.

- The analysis method (MVA) assumes a FCFS queueing discipline. In reality, this is not always the case; for example, the Pthreads implementation in glibc 2.12 (which is the version we used in the *dedup* experiment), has an unfair scheduling discipline. We observe that when two threads compete for the same lock, it is more likely for the thread accessing the lock more frequently to acquire the lock.

- In our model, the threads' choices of locks are assumed to be independent and randomly distributed in accordance with the routing matrix. In our experiments, this assumption holds.

- Exact solutions of queueing networks exist only for networks with class-independent service times. In actual applications, the service times may be class-dependent. There are methods to get approximate solutions for class-dependent service times [12].

- The model assumes that the lock overhead does not vary with the number of hardware cores. In actual implementations, the overhead may vary due to cache coherency traffic, for example.

These discrepancies lead us to conclude that the results of the model will be approximate, and the approximation error has to be evaluated. We are investigating into this.

5. EVALUATION

To evaluate our analytical model, we use three kinds of benchmarks: microbenchmarks with simple lock access patterns, a microbenchmark mimicking lock access behaviors of a real benchmark and a real benchmark (*dedup*) from the PARSEC benchmark suite [3] version 2.1, which is a state-of-the-art multi-threaded program benchmark suite.

5.1 Experimental setup

Two target multi-core machines have been used in our experiments:

- a small 8-core machine: a dual quad-core Intel Nehalem E5520 NUMA machine, with a QuickPath Interconnect (QPI) and

- a larger 64-core machine: a 64-core 8-socket NUMA machine with eight 8-core Intel Nehalem X6550 CPUs, with QPI.

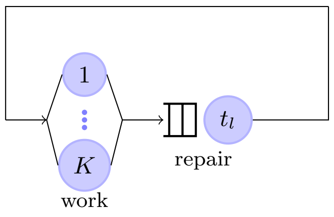

Figure 5: The machine repair model

Both of the machines run Linux 2.6.32 and all the binaries have been compiled with gcc version 4.4.6. We assign one thread to each core to avoid the cost of thread migration and context switch overhead.

In the microbenchmarks, an implementation of the CLH queue lock [7] has been used, to guarantee fairness in the lock scheduling policy. In the *dedup* benchmark, the original Pthread mutexes have been used.

5.1.1 Lock overhead

Each lock access incurs a certain overhead, caused by the execution of lock specific code at the moment of acquiring and releasing the lock. Such overhead should be accounted for in the lock holding times. Different lock implementations may have different overheads. In our experiments, two lock implementations are used, Pthread mutex lock (in glibc 2.12) and CLH locks.

Pthread mutex lock: When the mutex is not contended, acquiring the mutex is fast (under 100 cycles). When the mutex is contended, a thread (or threads) trying to acquire the lock is put into the sleep queue by the operating system. When the thread holding the mutex releases the mutex, the threads in the sleep queue are woken up and they may try to acquire the lock again. The measured overhead for waking up a thread is around 2400 cycles.

CLH lock: The CLH lock is a queue-based lock which guarantees fairness. When the CLH lock is not contended, the overhead is fast (around 84 cycles). When the lock is contended, the thread puts itself at the end of the queue and the thread before it points to the newly arrived thread. This is done through atomic swap. The measured overhead in the contended case is around 1600 cycles.

We account for the lock overhead in the lock holding time in our analytic models.

5.2 A simple microbenchmark

In this microbenchmark, the program consists of a number of threads. All threads execute the same loop. Within the loop, a local computation is done for a duration of t_c, before accessing a shared lock for a duration of t_l. To study the impact of the lock holding time on the lock contention, we set $\overline{t_c}$ to 10^7 cycles and vary the lock holding time (t_l). The number of threads increases from 1 to 64. Three sets of experiments are conducted using different distributions (exponential, deterministic and uniform) of the lock holding time and local computation time.

5.2.1 Exponential distributions (Figure 6)

In this set of experiments, both the local computation time and lock holding times are exponentially distributed. We used an $M/M/1/K/K/FCFS$ queueing network (aka. the machine repair model) to model this program (Figure 5). In the machine repair model, all the machines work, get repaired (only one machine can be repaired at a time) and are put back to work after being repaired. The *infinite-server* node in the network represents the machines working and the *one-server* node represents the machines being repaired. The *one-server* node has a FCFS policy and there are K machines in the network.

Figure 6 compares the measured lock contention with the modeled contention. We can see that the lock contention increases as the number of threads increases and as the lock holding time increases. Intuitively, more threads generate more contention. Increasing the lock holding time increases the probability of trying to acquire the lock when it is already taken, which creates more contention. The model is reasonably accurate with an average relative error of 3%.

5.2.2 Deterministic distribution (Figure 7)

To investigate the case of non-exponentially distributed lock holding times, we consider deterministic computation times and lock holding times. As a model, we used the $D/D/1/K/K/FCFS$ queueing network with one *infinite-server* node and one *one-server* node, which can be solved exactly [4]. According to the analytical model, there is lock contention only when $\rho > 1$ where

$$\rho = \frac{(n-1)\cdot\overline{t_l}}{\overline{t_c}}$$

and n is the number of threads. When the lock holding time is $2\cdot10^6$ cycles, it can be easily calculated that the lock contention only occurs with more than 6 threads. When $\rho > 1$, the average lock contention time is calculated as $lock\ contention = n\cdot\overline{t_s} - \overline{t_c}$. The model is reasonably accurate with an average relative error of 3.8% with $1-64$ threads.

5.2.3 Uniform distribution (Figure 8 and Figure 9)

In reality, lock holding times can be expected to follow other distributions. We conducted an experiment with uniformly distributed lock holding times, in which the coefficient of variation (*cv*) was varied between 0 (corresponding to a deterministic distribution) and 1 (the same *cv* as an exponential distribution).

This program is modeled with an $M/G/1/K/K/FCFS$ queueing network. The solution to this network is derived with the equations in [2]. The model is very accurate with an average relative error below 1% in Figure 8. In Figure 9, our model predicts the lock contention reasonably accurately with an average relative error of 4.8%.

Figure 8 shows that the lock contention increases when we switch from a uniform distribution to an exponential distribution with a small number of threads. In this experiment, the different *cv* of the uniform distribution does not affect the lock contention much. We can conclude that in such a simple network, assuming that the lock holding times are exponentially distributed gives an overestimation of the lock contention. Figure 9 shows that the effect of the distribution of lock holding times on lock contention is negligible with a larger number (16 - 64) of threads.

5.3 Emulating bodytrack in PARSEC

In this microbenchmark we have aimed at emulating the *bodytrack* benchmark locking behavior. In bodytrack, there are three locks and three classes of threads: one class for the

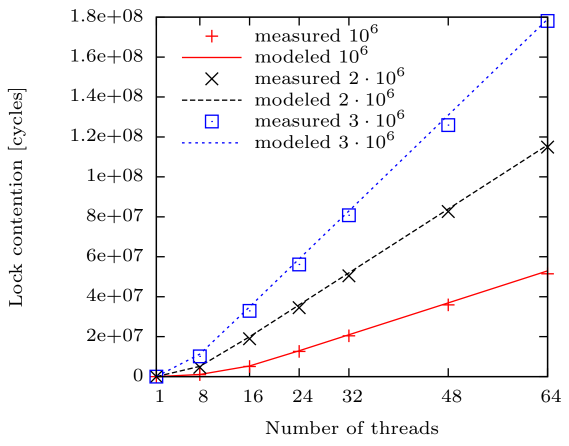

Figure 6: Measured vs. modeled lock contention of simple microbenchmark 1: the three lines are instances of the experiment with different lock holding times, shown in the legend. The computation and lock holding times are exponentially distributed.

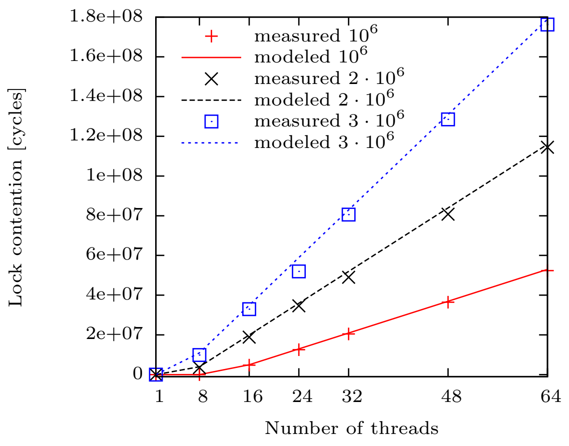

Figure 7: Measured vs. modeled lock contention of simple microbenchmark 1: the three lines are instances of the experiment with different lock holding times, shown in the legend. The computation and lock holding times are deterministic.

main thread, one class for the I/O thread and one class for the threadpool threads. Each class has different lock access patterns and different local computation times. Figure 10 shows that our model is accurate for *lock* 1 and *lock* 2. However, the discrepancy between the measured contention and our model increases for *lock* 3 as the number of threads increases. This may be due to the short lock holding times (on average 7000 cycles as measured in the real *bodytrack*) for lock 3, which makes the effect of the lock overhead more significant.

5.4 Dedup

The *dedup* benchmark is a data compression application, implemented with pipeline parallelism. There are five pipeline stages (shown in Figure 11, taken from [9]).

The first and last stages (*DataProcess* and *SendBlock*) are sequential. Respectively, they read from the input file and write to the output file. The intermediate stages are parallel: *FindAllAnchors*, *ChunkProcess* and *Compress*.

- *FindAllAnchors*: uses the Rabin-Karp fingerprint to partition the input data into smaller data chunks.
- *ChunkProcess*: maintains a hashtable for storing data chunks. It calculates the SHA1 key of each chunk and uses the key as the hashtable index. After calculating the SHA1 key, it searches the hashtable for the chunk. If a duplicate chunk is found, that chunk then bypasses the *Compress* stage. Otherwise, the chunk will be inserted into the hashtable.
- *Compress*: fetches data chunks from the hashtable, compresses the data and adds the compressed data back to the hashtable.

Adjacent stages share two buffer queues (shown in the lower part of Figure 11). There is one lock protecting each queue. The previous stage puts data into the queue and the next stage takes data from the queue.

There are two types of locks: hashtable locks and buffer queue locks. There is one hashtable lock for each bucket in the hashtable. The hashtable locks are accessed by the *ChunkProcess* stage and *Compress* stage. We observed that the buffer queues lock have very little lock contention. In this evaluation, we thus focus on the hashtable locks.

5.4.1 *Experimental settings*

We set up our experiment on the 64-core machine (see Section 5.1). The setup was as follows:

- 53 hashtable locks and 4 buffer queue locks are used.
- n threads in each parallel pipeline stage (n varies between 2 and 8).
- Random input data. We used a set of random input data (670MB), which is similar to the size of the original *native* input. With the native input, the hash function in *dedup* generates very uneven hash keys. This results in an unfair effect that some locks are accessed more times than others. To reduce the number of hash collisions, we used random input.

5.4.2 *Modeling the lock contention*

In our experiment of running *dedup*, threads from different stages finish their work at different points in time, when oversubscription is not used. We divide the whole execution into three phases according to the number of active threads. Such a division is necessary because the system characterization changes in the different phases. In phase 1, all threads are active. In phase 2, the *ChunkProcess* threads and *Compress* threads are active. In phase 3: only the *Compress* threads are active. Dividing the execution into three phases, the number of active threads in each phase is constant.

In our analysis, each parallel pipeline stage is considered as a job class. There are three classes in our queueing network: *FindAllAnchors*, *ChunkProcess* and *Compress*. Each of them has a separate routing matrix and service times. After profiling a run, we construct a queueing network. By solving the network, we calculate the lock contention time for all the threads.

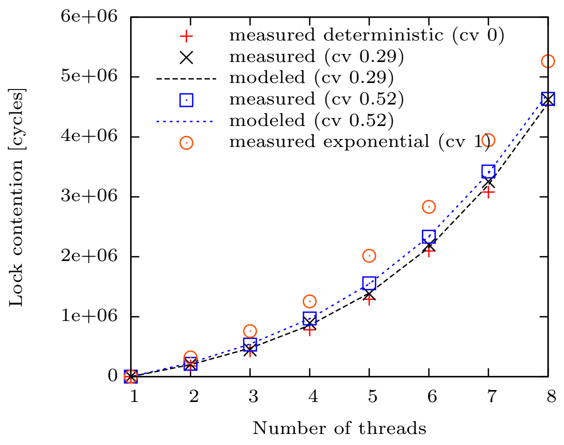

Figure 8: Measured vs. modeled lock contention with lock holding time of different coefficient of variation (cv) with $1-8$ threads

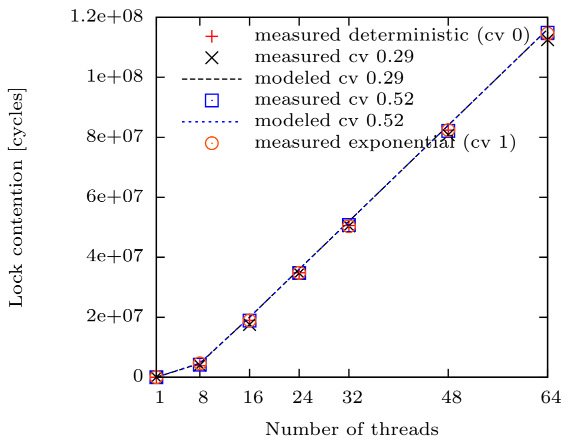

Figure 9: Measured vs. modeled lock contention with lock holding time of different coefficient of variation (cv) with $1-64$ threads.

The measured and modeled contention are obtained as follows:

Measured contention: We sum up the lock contention for all the locks during the whole execution of each class of threads (*ChunkProcess* and *Compress*) from our profiled run.

Modeled contention: After profiling *dedup*, we get the parameters for the queueing network for the lock accesses (routing matrix R and service times). We add the lock overhead obtained in section 5.1.1 to each lock. To obtain the lock contention time, we solve the queueing network with MVA and calculate the weighted sum of all the lock contentions where the weights are the number of accesses to each lock. Note that we do not predict the lock holding times for each lock with different number of threads. Instead, we calculate the lock holding times by subtracting the Release lock timestamp and Acquire lock timestamp. The lock holding time might increase with more threads due to contention for other shared resources. In our analysis, we factor out this effect and focus on the lock contention.

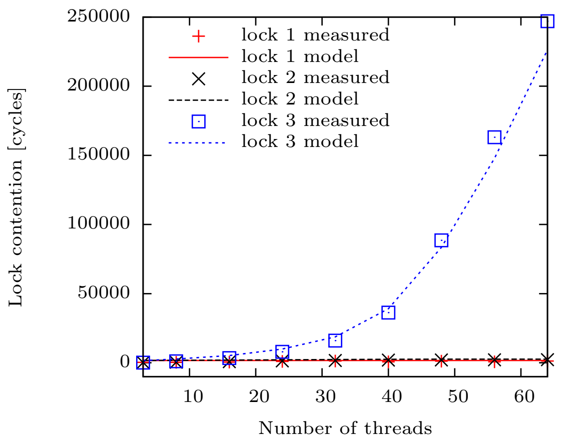

Figure 10: Bodytrack microbenchmark contention with increasing parallelism

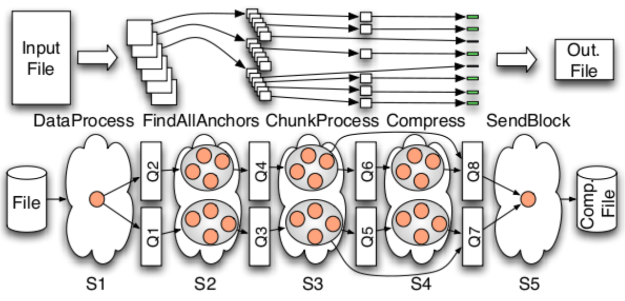

Figure 11: Structure of the dedup benchmark: figure taken from [9]

5.4.3 Evaluation results

Figure 12 shows our evaluation results for the measured and modeled lock contention for the *ChunkProcess* and *Compress* threads. Our model describes the lock contention well when there are at least 5 threads in each stage (15 threads in total). The average error in these cases ($n \geq 5$) is 8.46% and 15.17% for *ChunkProcess* and *Compress* stages respectively. When there are less than 5 threads we observe an overestimation for threads in both classes (57.77% for the *ChunkProcess* stage and 11.49% for the *Compress* stage). This may be due to the fact that we add a constant lock overhead regardless of whether the lock is contended.

6. CONCLUSIONS

In this paper, we proposed a method of using queueing networks to model lock access behavior in parallel programs and studied its accuracy in predicting the lock contention. We evaluated the model with three classes of benchmarks: simple microbenchmarks with simple lock access patterns, microbenchmarks mimicking the lock access behavior of real benchmarks and a real benchmark.

The model predicts accurate lock contention for programs with simple lock access patterns. With our simple benchmarks, we studied the effect of the distribution of lock hold-

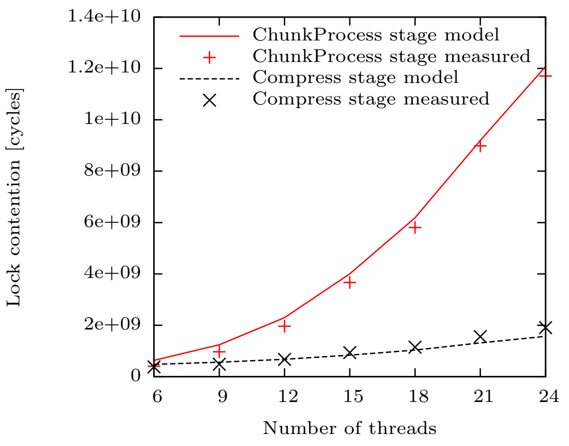

Figure 12: dedup measured and modelled contention for the *ChunkProcess* and *Compress* stages

ing times on the lock contention. If the coefficient of variation of the lock holding time is between 0 and 1, our model gives an overestimation.

When the lock holding time is on the same magnitude as the lock overhead (a few hundred to a few thousand cycles), we observe a discrepancy between the model and measured lock contention.

Ongoing research includes predicting the lock holding times as a function of the number of threads. We plan to extend our model with such an extension to predict the lock contention for any number of threads.

7. ACKNOWLEDGMENTS

This work is supported by the Swedish Foundation for Strategic Research through CoDeR-MP, and by the Swedish Research Council through UPMARC. We are very grately to David Eklöv for the valuable discussions and comments on the writing of this paper, to Andreas Sandberg for his implementation of the CLH lock and to Nikos Nikoleris for his modified version of Mutrace. We would like to thank the anonymous reviewers for their helpful suggestions.

8. REFERENCES

[1] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.*, 22(1):94–136, Feb. 2004.

[2] A. O. Allen. *Probability, statistics, and queueing theory with computer science applications*. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] J. W. Boyse and D. R. Warn. A straightforward model for computer performance prediction. *ACM Comput. Surv.*, 7(2):73–93, June 1975.

[5] S. Eyerman and L. Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 362–370, New York, NY, USA, 2010. ACM.

[6] S. Lam. A simple derivation of the mva and lbanc algorithms from the convolution algorithm. *Computers, IEEE Transactions on*, C-32(11):1062 –1064, nov. 1983.

[7] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *In Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171. IEEE Computer Society, 1994.

[8] M. A. Marsan. Advances in petri nets 1989. chapter Stochastic Petri nets: an elementary introduction, pages 1–29. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[9] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical modeling of pipeline parallelism. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 281–290, Washington, DC, USA, 2009. IEEE Computer Society.

[10] L. Poettering. Measuring lock contention, 2009.

[11] M. Reiser and H. Kobayashi. On the convolution algorithm for separable queuing networks. In *Proceedings of the 1976 ACM SIGMETRICS conference on Computer performance modeling measurement and evaluation*, SIGMETRICS '76, pages 109–117, New York, NY, USA, 1976. ACM.

[12] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2):313–322, Apr. 1980.

[13] D. J. Sorin, J. L. Lemon, D. L. Eager, and M. K. Vernon. Analytic evaluation of shared-memory architectures. *IEEE Trans. Parallel Distrib. Syst.*, 14(2):166–180, Feb. 2003.

[14] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 269–280, New York, NY, USA, 2010. ACM.