Modeling Software Contention using Colored Petri Nets

Nilabja Roy, Akshay Dabholkar, Nathan Hamm,
Larry Dowdy and Douglas Schmidt
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37203 , USA

Abstract

Commercial servers, such as database or application servers, often attempt to improve performance via multi-threading. Improper multi-threading architectures can incur contention, limiting performance improvements. Contention occurs primarily at two levels: (1) blocking on locks shared between threads at the software level and (2) contending for physical resources (such as the cpu or disk) at the hardware level. Given a set of hardware resources and an application design, there is an optimal number of threads that maximizes performance. This paper describes a novel technique we developed to select the optimal number of threads of a target-tracking application using a simulation-based Colored Petri Nets (CPNs) model.

This paper makes two contributions to the performance analysis of multi-threaded applications. First, the paper presents an approach for calibrating a simulation model using training set data to reflect actual performance parameters accurately. Second, the model predictions are validated empirically against the actual application performance and the predicted data is used to compute the optimal configuration of threads in an application to achieve the desired performance. Our results show that predicting performance of application thread characteristics is possible and can be used to optimize performance.

1 Introduction

Emerging trends and challenges. Servers, such as database servers or web servers, typically receive incoming requests, process them, and then returns responses to the requesting clients.

One way to improve the response time of a server is to create multiple threads to service requests. Each incoming request can be assigned to a thread that processes it and prepares the response.

With the growing adoption of multi-core and multi-processor machines, software applications require multi-threading to leverage hardware resources effectively [10]. In theory, multi-threading can significantly improve system performance. In practice, however, multi-threading can incur excessive overhead due to *software contention* (*e.g.*, mutually exclusive operations needed to mediate thread access to shared data) and *physical contention* (*e.g.*, access to hardware resources, such as CPUs and memory). There is a trade-off between (1) increasing the number of threads to decrease client response time vs. (2) a larger number of threads causing bottlenecks that can increase response time.

What is needed, therefore, is a technique for selecting the optimal number of threads, which depends upon various factors including the underlying hardware, multi-threading architecture, and application logic. The following are two phases in the software lifecycle that can benefit from such a technique:

• **Application development.** Developers of multi-threaded applications must carefully evaluate various performance tradeoffs. For example, although a large number of critical sections can increase response time, critical sections are also required for correct functionality of applications by ensuring safe access to resources shared by multiple threads. While developing multi-threaded applications, therefore, developers must consider various factors, such as (1) the maximum number of critical sections that a thread can access before the performance degrades, (2) what type of multi-threaded architecture to use (*e.g.*, thread pool, thread-per-request, thread-per-connection, or the Half Sync/Half Async or Leader/Follower patterns [9]) provides the best results for a particular application and hardware environment, (3) identifying which application components are bottlenecks so they can be redesigned to reduce contention.

• **Application deployment.** To prepare an application for production use, deployers must first estimate its hardware needs based on its expected workload and application performance requirements. Once an application is installed, its configurable parameters must be set to the appropriate values. An important parameter is the maximum number of concurrent threads, which is a form of admission control used to maintain the liveness of the server. The multi-threaded architecture used in the application also has a big impact on the number of threads that could be used effectively. Deployers must therefore make decisions related to the hardware needs of the application, the multi-threaded architecture of the application and the optimal number of concurrent threads.

In conventional multi-threaded systems, application developers and deployers make these decisions manually us-

ing their experience and intuition, which can be tedious and error-prone. Moreover, when workloads change, it is hard to estimate the effect on application performance since there is no explicit and analyzable model application component behavior. As a result, performance problems typically emerge late in the software life-cycle during the integration phase, where they are more costly to fix.

Solution approach → Optimize an application configuration using simulation models. This paper presents and evaluates a method for modeling the software and physical contention of multi-threaded applications to estimate the number of threads needed to produce optimal performance using a particular set of hardware resources. This method constructs a simulation model of a complex multi-threaded application using *Colored Petri Nets* (CPNs) [1], which are a discrete-event modelling language that combines Petri nets with the functional language Standard ML [8]. A CPN model of a system is an executable model consisting of different states and events, along with a notation that represents the time taken to trigger events. CPNs are suited for modeling concurrency, communication, and synchronization among different components in a system. Our work uses *CPN tools* [2], which help construct and analyze CPN models via an engine that conduct simulation-based performance analysis.

We use CPNs in this paper to model simultaneous resource possession for a target tracking application containing many threads sharing multiple locks. We first profile the application and collect runtime performance data, which is used to parameterize the CPN model. The CPN model is then run to predict application performance under various configurations. We compare the predictions with measured data to validate the CPN model. This paper describes the challenges we addressed building the CPN model and using it to predict the behavior of our target-tracking application.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes a target-tracking application case study designed using multiple threads and locks; Section 3 presents and analyzes the profiled data of the application, describes the CPN model building process, and validates the it using the profiled data; Section 4 selects the optimal configuration of various threads of the application using CPN model predicted data; Section 5 discusses related work; Section 6 presents concluding remarks and lessons learned.

2 Application Case Study: Target Tracking Simulator

This section describes the application we created and used as a case study to evaluate our work on performance prediction of multi-threaded applications.

2.1 Overview of the Target Tracker

Our case study centers on a target-tracking simulation application composed of active objects [9], such as target, tracker, and satellites shown in Figure 1. There can be multiple instances of trackers and satellites; each tracker collects the target's latest location from a satellite. To increase the probability that the target will be found, the application must be configured with the right number of trackers and satellites.

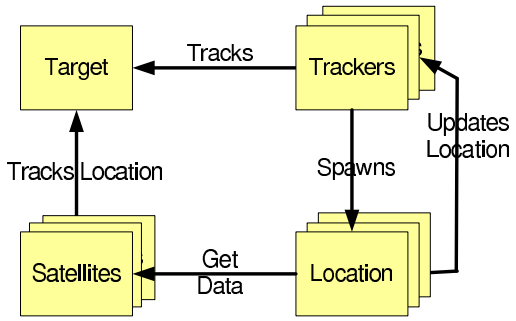

Figure 1. Active Objects in the Target Tracking Simulator

Each active object has its own thread and executes methods of its own object, *i.e.*, there is a one-to-one correspondence between an active object and a thread. Every active object, such as target, tracker, and satellite, executes its application logic as shown in Figure 2. Sometimes an active object interacts with the other active objects to exchange data, *e.g.*, each tracker collects data from the satellite during every period. An active object therefore performs a periodic task that sleeps for a specified length of time, wakes up and performs some work, and goes back to sleep, as shown in Figure 2.

As evident from the Figure 2, each type of active object has its own logical flow and contends for shared data with other objects thus blocking each other. An overview of the various active objects in our application case study is shown in Figure 1 and described below.

1. ***Target.*** This simulates a target that moves across a given area and tries to evade its trackers. Every time it wakes up, it randomly calculates a new direction and velocity and goes to sleep again. While it sleeps it moves in a particular direction with designated velocity. There is only one instance of the target in the application.
2. ***Trackers.*** The role of the trackers is to chase the target. They obtain the latest position of the target through the use of the location objects described below. A tracker recalculates its new direction and velocity each period depending upon the latest position of the target. It also checks if it hits the target. A target is considered hit

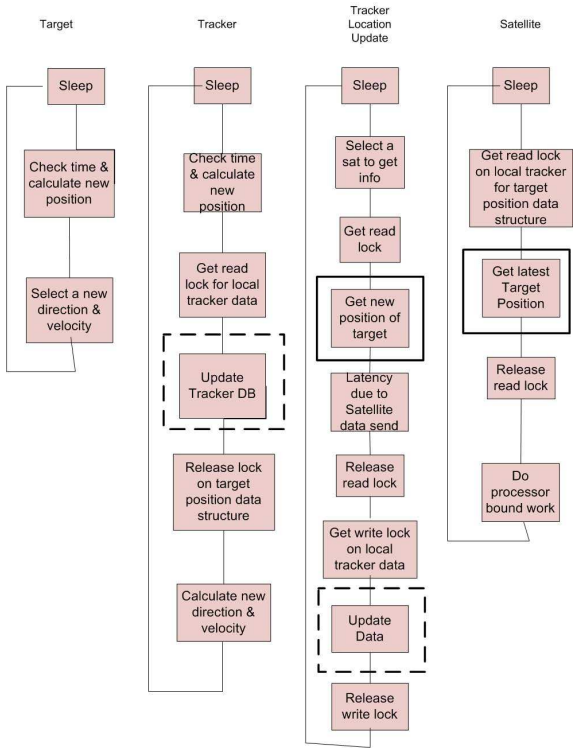

Figure 2. Application Logical Flows in the Target Tracking Simulator

if its current position is within some small distance of the target. There are multiple instances of the tracker.

3. ***Satellites.*** The satellites gather information of the latest position of the target. Within the application the latest coordinates of the target is placed in a global variable which each satellite reads periodically.
4. ***Tracker location updates.*** These are update objects and are created by the tracker entities. A location object is spawned for each satellite present in the application. The location objects periodically call on the satellite, obtain the latest position of the target, and update the local database within the tracker. There is a location active object for each pair of satellite and tracker object.

Although the target object does not exhibit any contention with any other object, the other objects contend with each other. As shown in Figure 2, the "Update tracker DB" activity in the tracker flow contends with the "Update Data" activity in the Location flow. Likewise, the "Get new position of target" activity contends with the "Get latest target position" activity on the satellite flow. The blocking time on these locks increases when the number of objects increases which also increases the number of threads.

2.2 Case Study Application Goals

Our case study application is designed to track down the target a maximum number of times. In theory it may appear that the chances of hitting the target grows with an increased number of satellites and trackers, though in practice this approach may increase contention, which can decrease tracker and satellite throughput, as well as decrease their effectiveness and increase the time to hit the target. In particular, increasing the number of active objects or threads might improve application performance but it could also degrade performance by increasing bottleneck contention. Application deployers will therefore benefit from a technique that can determine the optimal number of trackers and satellites needed to hit the target in the least amount of time.

2.2.1 Predict Application Performance

The first goal of our case study is to predict the performance of the target tracker application under configurations that differ in terms of the number of tracker and the satellite objects. The notation we use to depict each configuration is: # of target objects_# of tracker objects_# of satellite objects. Thus, a configuration of 1_2_3 means that there is 1 target, 2 trackers, and 3 satellites. As mentioned in section 2.1 there is a location object for each pair of tracker/satellite. As a result, the configuration 1_2_3 would have 2x3=6 location objects, resulting in a total of 1+2+3+6=12 objects. Since there is a single thread per active object, this means there are 12 threads in the application for this configuration.

We observe the application until the target performs 500 periods. In each period, the target completes one iteration of sleep and computation, as shown in Figure 2. The application runs under two separate scenarios: (1) with all the locks and (2) with none of the locks. The latter method is obviously incorrect from a functionality point of view but it quantifies the impact of contention and blocking on performance.

The accuracy of the prediction is not important. The important point is that the relative performance characteristics should be captured by the model, *i.e.*, the performance patterns/trends should be predicted. For example, the model should be able to tell if the average throughput of the tracker decreases or increases when a particular configuration is changed. The magnitude of the difference, however, is not important.

2.2.2 Extract Optimal Configuration

We use the performance data predicted by a simulation model of the application to choose the best configuration for the application, where "best" is defined as the greatest likelihood of the trackers hitting the target. To use the model predicted data, we use a utility function that quantifies the chances to hit the target the most number of times by maximizing the following factors:

- **Tracker activity** should maximize $N_{tr} * \mu_{tr}$, where N_{tr} is the number of trackers configured in the application and μ_{tr} is the average throughput of each tracker. This expression represents the number of

times a tracker activity takes place in unit time, *e.g.*, per second.

- **Location updates** should maximize $N_{tr} * \mu_{loc}$, where μ_{loc} is the average throughput of the location object for each tracker. This expression represents how frequently the latest position is updated to the tracker.
- **Satellite throughput** should maximize $N_{sat} * \mu_{sat}$, where N_{sat} is the number of satellites configured and μ_{sat} is the average throughput of each satellite. This expression represents the number of times the satellite updates the latest location of the target.

The chance of hitting the target with N_{tr} trackers is expressed by the function $H(N_{tr})$ and is computed as:

$$H(N_{tr}) = N_{tr} * (\mu_{tr} + \mu_{loc}) + N_{sat} * \mu_{sat} \qquad (1)$$

where N_{tr} is the number of trackers, μ_{tr} is the average throughput of the tracker, μ_{loc} is the average throughput of the location, N_{sat} is the number of satellites and μ_{sat} is the average throughput of the satellite. The configuration that maximizes the value of this function should provide the preferred setting for the application, which can be computed by predicting the throughput of the tracker, location, and the satellite and using them set the desired QoS value.

3 Experiments

This section discusses the following steps we performed to create a model of the application case study described in Section 2 and validate the model against profiled data:

1. Profile the application to record application activity within critical sections of code
2. Create a Petri net model of the application logic that captures the contention among hardware/software resources
3. Validate the model by using the profile data and the performance predicted by the model.

The remainder of this section discusses each of these activities.

3.1 Application Profiling

Experiment design. Our application case study is profiled under different thread configurations to collect performance data that is then used to calibrate and validate the simulation model. The platform used for the experiments is a single CPU, Intel Pentium, 1.70 GHz machine with 1 GB of RAM. The operating system used is Windows XP Professional Version 2002 with service pack 2. As mentioned in Section 2.2, this application runs until the target completed 500 iterations. The time taken by the target is recorded (T_{tg}), along with the number of iterations of other objects or threads. After this data is recorded the throughput of satellite and location are measured. The throughput of the satellite is defined as N_{sat}/T_{tg}, where N_{sat} is the number of iterations of a satellite. Likewise, the throughput of the location is N_{sat}/T_{loc}, where N_{loc} is the number of location iterations.

To capture the throughput and response time of the different threads, we needed to profile the activities of their associated active objects. Application methods of the target object were therefore instrumented to include timestamp recording. Similarly, we inserted instrumentation code into the satellite and tracker objects to count the number of iterations.

Experiment results. After inserting the instrumentation code, we ran our application case study for 13 different thread configurations and collected the profiled data. The results are shown in Table 1. Each row of the Table 1 contains the data recorded for a single configuration.

	With Mutex				Without Mutex			
Config	Target run time (secs)	Satellite Throughput (periods/sec)	Tracker throughput (periods/sec)	Location Throughput (periods/sec)	Target run time (secs)	Satellite Throughput (periods/sec)	Tracker Throughput (periods/sec)	Location Throughput (periods/sec)
1_0_0	140	-	-	-	140	-	-	-
1_0_1	135	3.706	-	-	135	3.71	-	-
1_0_2	130	3.85	-	-	130	3.84	-	-
1_0_3	135	3.7	-	-	135	3.7	-	-
1_1_1	138	2.12	65.89	2.69	139.2	3.58	61.89	3.01
1_2_1	137	1.29	67.85	1.44	135.3	3.68	29.5	3.12
1_3_1	138	0.91	68.79	0.99	131.77	3.79	18.54	3.19
1_1_2	144	2.15	51.43	2.62	131.74	3.78	54.31	3.18
1_2_2	144	1.26	54.49	1.39	130.7	3.81	23.32	3.17
1_3_2	145	0.89	55.92	0.95	153.2	3.24	9.61	2.68
1_1_3	144	2.18	42.96	2.7	132.6	3.76	44.64	3.10
1_2_3	145	1.28	47.20	1.42	170.39	2.94	10.73	2.43
1_3_3	145	0.91	48.95	0.97	212.5	2.37	4.24	1.91

Table 1. Profiled Data from the application

Analysis of results. The results in Table 1 show a significant cache effect. For example, the data for the configuration 1_0_1 (with 1 target and 1 satellite) in the table shows a throughput of 3.70 iterations/sec for the satellite active object, whereas the throughput of the satellite active object in configuration 1_0_2 (*i.e.*, with 1 target and 2 satellites) is 3.85 iterations/sec. The throughput for satellite objects therefore increases as the number of satellites increase. When the number of satellites increases to 3, however, the throughput decreases since CPU utilization increases due to higher contention.

Cache effects can also be seen from the response time of the target in Table 1. For example, when the target active object runs on its own (1_0_0) the time taken to complete 500 iterations is 140 secs, where when a satellite active object runs concurrently with it (1_0_1) the time reduces to 135 secs. This difference stems from the fact that the target and the satellite active objects perform similar arithmetic computations, so as the number of satellite objects increase the cache effects become apparent until the CPU utilization reaches a certain threshold, after which the response time starts to increase.

3.2 Colored Petri Net Model Construction

We now explain the simulation model of the application case study created using Colored Petri Nets (CPNs). Figure 3 shows a screenshot of the CPN tool and the model of our application represented using CPN. The following four

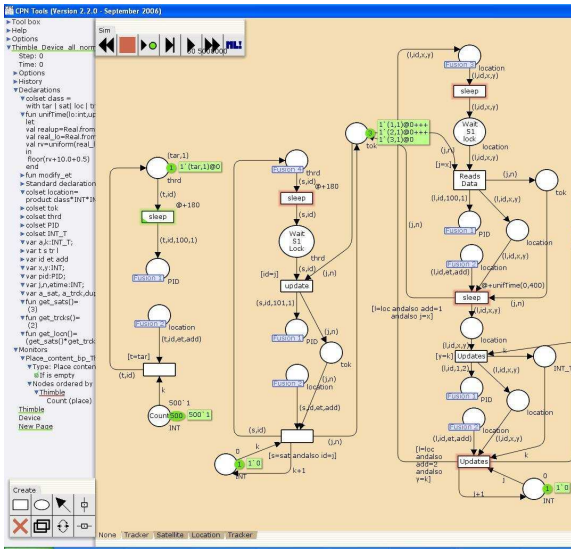

Figure 3. CPN Model of the Application Case Study

aspects of the application are part of the system modeling process:

- **Modeling application flow**, which models the logic of each object similar to the workflows shown in Figure 1.
- **Modeling lock contention**, which models the waiting and acquiring on the software locks, *i.e.*, process scoped mutexes, also known on Windows as "critical sections."
- **Modeling resource access**, which models the concurrent access of the physical resources by each thread.
- **Modeling cache effects**, which models the changes on computation time due to simultaneous threads performing similar work on the CPU.

Below we elaborate on the modeling of these four aspects.

3.2.1 Modeling Application Flows

Colored Petri nets model application flows via *places*, *transitions*, and *tokens*. Each transition moves tokens from the input places to the output places. The placement of a token in a place indicates the location of control within the application thread.

Figure 4 shows the application flow of the thread in the active object. In this figure places are connected through transitions. Whenever the input places has a token, the connected transition can fire and move the token. Control therefore moves from each place to the next corresponding to the workflow shown in Figure 1.

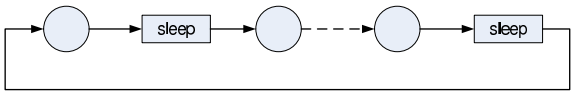

Figure 4. A CPN Model of the Thread in the Target active object

Figure 4 shows how sleep is used to implement a delay that simulates the interval where the task fires. Transition firing times of the second and third transitions model physical device access, which is the CPU in this case. As seen in the figure, when the device access is completed control flows back to the starting position.

3.2.2 Modeling Lock Contention

Colored Petri nets can also model contentions. For example, Figure 5 shows a portion of a CPN model where the threads in the satellite and location active objects contend for a shared lock. The place named "lock" represents the

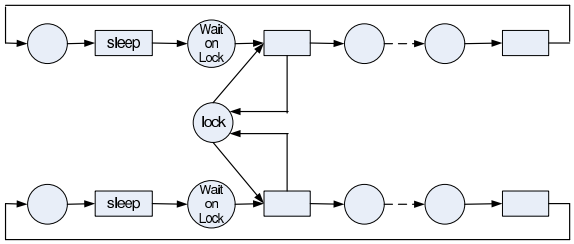

Figure 5. The Model of Contention for a Software Lock

software lock, which is available if a token is present in that place. The places in the thread flow named "Wait on lock" model the thread waiting on the lock. If the token is available, the transition on a single thread is executed and the token moves out of the place "lock," which causes the other thread to block until the token again becomes available.

3.2.3 Modeling Resource Access

CPNs can model resources (such as the CPU) similarly to locks. Multiple objects contend for the CPU, but only one thread at a time can access it. A place is therefore created in the model to represent the CPU and every object has a connection to it.

Since the CPU is accessed by all threads, the model becomes visually cluttered. A feature of hierarchical nets of the CPN tool can be used, however, to move the place representing the CPU to a different page of the CPN model. It is then referred from every flow. The broken arrows connecting the two places shown in the Figure 4 represent the underlying contention for the CPU. Figure 6 shows the model of the CPU.

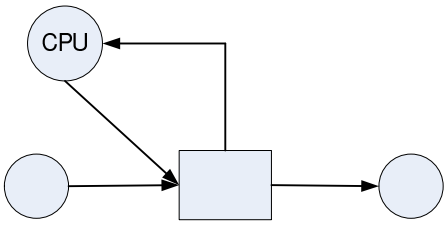

Figure 6. The CPN Model of the CPU

3.2.4 Modeling Cache Effects

Cache effects were observed during profiling, as discussed in Section 3.1. These effects should be incorporated within the CPN model so the model predicted performance data is as close to the actual values as possible. Figure 7 gives an empirical formula that is implemented within the place representing the CPU. This formula calibrates the execution

```
if (tint < 35)
 then
  val mod_et = modify_et(et,40)
 else
   if (tint < 45)
       val mod_et = modify_et(et,80)
  else
      if (tint < 180)
        val mod_et = modify_et(et,94)
       else
        val mod_et = modify_et(et,100)
```

Figure 7. The Formula to Implement Cache Effects

time of a thread running on the CPU. The formula decreases the execution time of a thread as the inter-arrival time between threads decreases.

The 'tint' variable in the formula represents the current inter-arrival time. If the value of 'tint' is less than 180 the execution time is modified to 94% of the original. In the extreme, if it is less than 35, the execution time is modified to 40% of the original. The percentage numbers above were computed by calibrating the CPN model via repeatedly running it with the data from configurations 1_0_0, 1_0_1, 1_0_2, 1_0_3 in Table 1.

3.3 Calibrating the Model

The techniques described in Section 3.2 helped implement the CPN model of the application. We now describe how the CPN model is calibrated using the profile data gathered as described in Section 3.1. Some of the profile data are used as a training set to tune the model parameter; the rest of the data are used to validate the model. The data for the configurations 1_0_0, 1_0_1, 1_0_2, 1_0_3 in Table 1 are used to train the model. These timing data were used to tune the formula to model the caching shown in Figure 7. The model is repeatedly run with the different configurations and the various percentage values in the formula is tweaked multiple times to converge to the above values shown in Figure 7.

Once the model is properly calibrated, it is run for the remaining configurations. For each configuration, the response time of the target thread and the throughput of the satellites and the location threads are calculated. Table 2 gives the resulting model prediction data,

	With Mutex				Without Mutex			
Config	Target run time (secs)	Satellite Through-put (peri-ods/sec)	Tracker Through-put (peri-ods/sec)	Location Through-put (peri-ods/sec)	Target run time (secs)	Satellite Through-put (peri-ods/sec)	Tracker Through-put	Location Through-put (peri-ods/sec)
1_0_0	140	-	-	-	140	-	-	-
1_0_1	135	3.69	-	-	135	3.70	-	-
1_0_2	130	3.83	-	-	130	3.85	-	-
1_0_3	135	3.69	-	-	135	3.69	-	-
1_1_1	132	3.59	73.12	2.77	130	3.83	51.22	3.26
1_2_1	132	3.69	77.41	1.54	135	3.70	25.67	3.16
1_3_1	132	3.79	76.74	1.02	144	3.49	11.06	2.87
1_1_2	143	3.78	30.58	2.25	143	3.48	8.30	2.89
1_2_2	144	3.81	38.49	1.35	170	2.96	4.94	2.18
1_3_2	144	3.24	39.49	0.92	191	2.57	4.05	1.66
1_1_3	153	3.76	7.19	1.54	170	2.94	4.62	2.05
1_2_3	162	2.94	13.10	1.12	209	2.37	3.39	1.26
1_3_3	162	2.37	13.79	0.79	237.77	1.99	2.79	0.95

Table 2. Model Predicted Data

3.4 Model Validation

We now compare the results obtained from profiling the actual application (Section 3.1) with the results from the model prediction (Section 3.2). These results are explained from the perspective of two conflicting factors: (1) the CPU that is the hardware resource bottleneck and (2) the software lock contention due to shared data accessed by various threads. Results are presented with different thread configurations on the x-axis and the runtime performance metric on the y-axis.

3.4.1 Target Thread Response Time

Figure 8 shows that the response time of the thread in the target active object remains nearly constant as the number of objects are varied in the application case study. This result

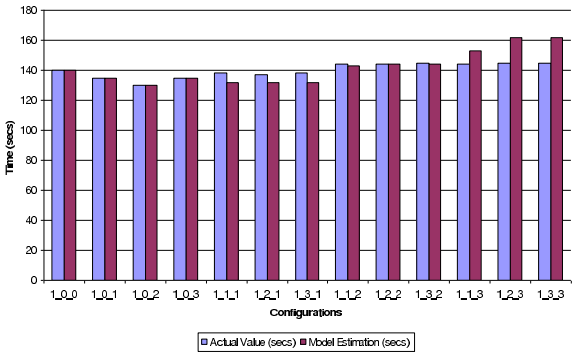

Figure 8. Response Time of Target Thread with Locks

occurs for the following two reasons:

- The target does not contend with other objects, so it does not face any extra blocking as the number of other objects increases.
- As the number of objects increases, the threads in these objects block each other due to software locks, which keeps the CPU relatively free so the target thread can use the CPU when needed.

This result seems non-intuitive since the underlying hardware is a single CPU machine. It seems reasonable that increasing the number of threads in an application running on a single CPU should increase the overhead and reduce the performance of each thread. The results in Figure 8, however, show how the performance of a thread that uses no software locks will increase when more threads that *do* use locks are added to the application.

3.4.2 Throughput of Satellite and Tracker

Figure 9 shows the behavior of the satellite thread when there are locks in the system. Each set of three configura-

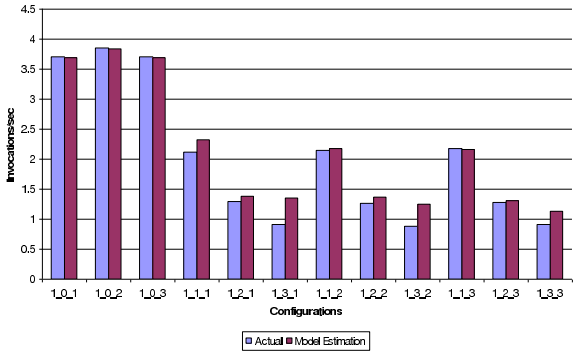

Figure 9. Throughput of Satellite Thread with Locks

tions in this graph should be considered together, *e.g.*, data for configuration 1_1_2, 1_2_2 and 1_3_2 should be considered together. Between the former configurations the number of location threads are increased, which increases contention and decreases throughput since the threads now spend more time blocked on the locks. The location thread also exhibits a similar trend as the satellite data, as shown in Figure 10.

The tracker throughput is shown in Figure 11. The error percent in model data is larger compared to other data, but the general trend of the application behavior is captured. For example, in each set of three successive readings with one satellite (1_1_1, 1_2_1 and 1_3_1), two satellites (1_1_2, 1_2_2 and 1_3_2), and three satellites (1_1_3, 1_2_3 and 1_3_3) the throughput increases as the number of trackers increase. This trend of the application behavior corresponding to each thread configuration helps identify the optimal configuration. The accuracy of the prediction is less important since we are only interested in determining if

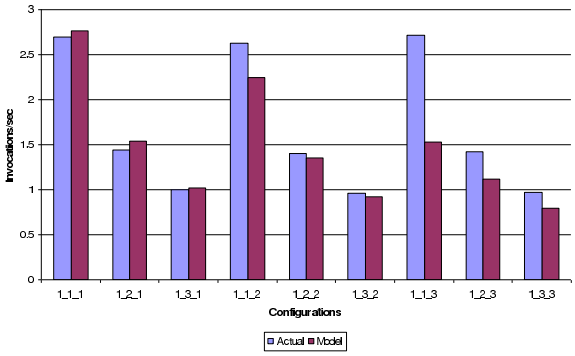

Figure 10. Throughput of Location Thread with Locks

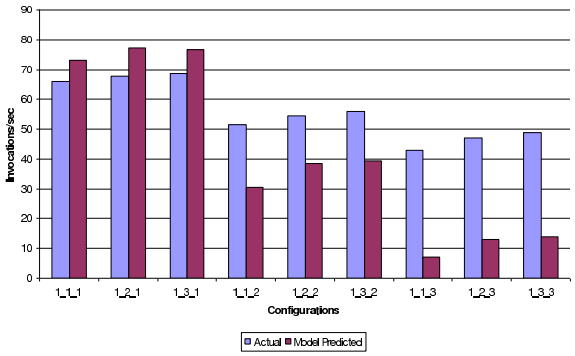

Figure 11. Throughput of Tracker Thread with Locks

a configuration is better than another, not how much better they are.

3.4.3 Performance Metrics with the Locks Removed

For this experiment we removed all the locks in the application, which clearly compromised its behavior since shared data could be corrupted due to simultaneous modifications by multiple threads. We removed the locks, however, to compare the performance of each thread and show the impact of using locks in the system. We also modified the CPN model and used it to predict the performance of the system. The model predicted data is shown along with the measured data in the Figures 13, 14 and 12.

Figure 13 shows the target thread response time, which increased as the number of objects increased. In this case, when the number of other objects increased they do not block each other and directly contend for the CPU, which increases the waiting time of the target at the CPU and its response time. Figure 14 shows the behavior of the satellite thread when there are no locks in the system. When the data in Figure 14 is compared with Figure 9, it is clear that throughput degrades less as the number of threads or objects increase due to the fact that there are no bottleneck due to locks. Nevertheless, the throughput still goes down

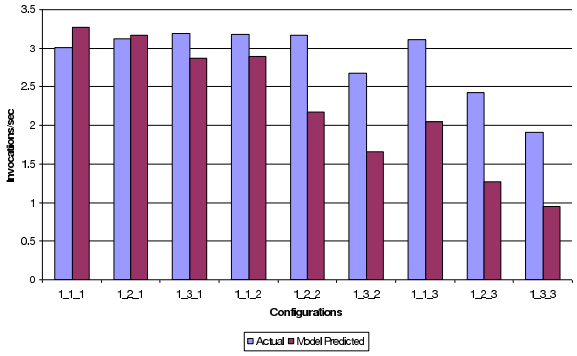

Figure 12. Throughput of Location Thread without Locks

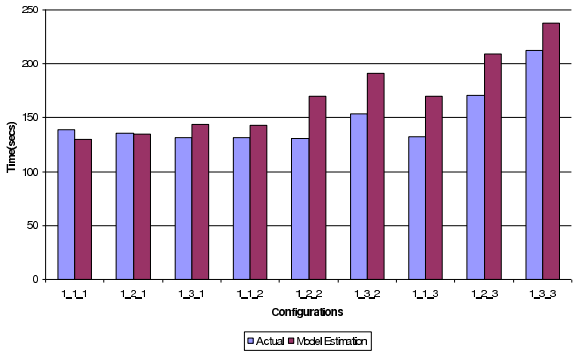

Figure 13. Response Time of Target Thread without Locks

due to the increased CPU contention.

3.4.4 Model Prediction

Although the CPN model accurately predicted the underlying trend in application behavior in the experiments described above there were errors in the model prediction. Some specific points have inconsistencies, *e.g.*, configuration 1_1_3 seems to indicate problems since the throughput of tracker and location predicted by the CPN model is much less than the actual value. Figures 11, 10 and 12 show that the model prediction differs significantly from the actual data. Potential reasons for these differences include (1) there is increased OS activity due to context switching or other activities that increase the throughput of the thread and/or (2) some form of cache effects cause this behavior. Overall, however, the CPN model mimics the application behavior, so developers and deployer can use these models to estimate application behavior accurately.

4 Application Configuration

This section demonstrates how the performance data predicted by the model can be leveraged to optimize application thread configurations. In particular, our case study used the results presented in Table 2 to find the optimal thread

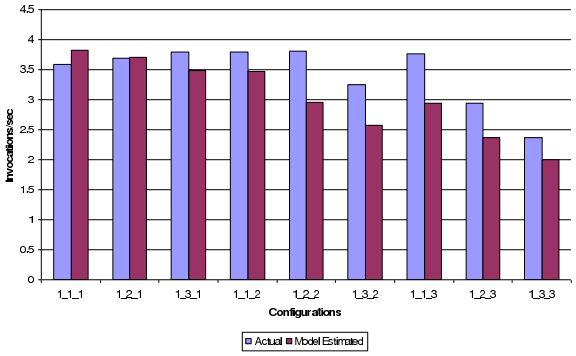

Figure 14. Throughput of Satellite Thread without Locks

Config	Tracker Num.	Tracker Throughput (periods/sec)	Location Throughput (periods/sec)	Satellite Number	Satellite Throughput (periods/sec)	Hit chance
1_0_0	0	_	_	0	_	0
1_0_1	0	_	_	1	3.69	3.69
1_0_2	0	_	_	2	3.83	7.67
1_0_3	0	_	_	3	3.69	11.09
1_1_1	1	73.11	2.77	1	2.32	78.20
1_2_1	2	77.41	1.54	1	1.38	159.28
1_3_1	3	76.74	1.01	1	1.36	234.63
1_1_2	1	30.58	2.25	2	2.18	37.19
1_2_2	2	38.49	1.35	2	1.37	82.43
1_3_2	3	39.49	0.91	2	1.26	123.75
1_1_3	1	7.19	1.53	3	2.16	15.22
1_2_3	2	13.10	1.12	3	1.30	32.37
1_3_3	3	13.79	0.79	3	1.13	47.18

Table 3. Target hit chances for different configurations

configuration. To verify the decision made using the model, we profiled the application and calculated the number of hits made by the trackers for each configuration.

We first used Equation(1) described in section 2 to compute the hit chance value for each configuration, as shown in Table 3. The average throughput values of the tracker and satellite is used from the model predicted data given in Table 2. Table 3 shows that configuration 1_3_1 has best chance of the trackers hitting the target, as explained in Section 2. This configuration should therefore be optimal for the application.

To verify whether the above configuration is optimal, the running application was then profiled to record the number of times the trackers hit the target. The results of this profiling is shown in Table 4. This table shows that configuration 1_3_3 has the highest number of hits, which validates that the configuration chosen using the modeled data and the utility function given by equation(1) is optimal.

The results above show how a simulation model can be used to determine the optimal configuration of threads for our case study application. Combining simulations with profiling helps application deployers optimize the perfor-

Confi g	Tracker 1	Tracker 2	Tracker 3	Total Hits
1_0_0	-	-	-	0
1_0_1	-	-	-	0
1_0_2	-	-	-	0
1_0_3	-	-	-	0
1_1_1	212	-	-	212
1_2_1	127	142	-	269
1_3_1	220	222	230	672
1_1_2	163	-	-	163
1_2_2	111	121	-	232
1_3_2	183	190	179	552
1_1_3	130	-	-	130
1_2_3	159	144	-	303
1_3_3	148	153	161	462

Table 4. Actual Runtime Target Hit Occurrences

mance of application thread configurations without the need for tedious and error-prone manual effort.

5 Related Work

Prior work has explored techniques for modeling software contention using analytical techniques and modeling thread contention using Petri nets. This section compares and contrasts our work with this related work.

In [5] and [6] two queueing network models are created: (1) a *hardware queueing network* model of the physical contention and (2) a *software queueing network* model of the software contention. Each model is solved iteratively until the results from the two converge to within a predefined value.

Our CPN-based approach uses a simulation model rather than an analytical model to improve model accuracy. Although simulation models require more time to predict performance [6] they are appropriate for our purposes since we analyze the models before application deployment. Our solution is also based upon modeling of the application flow and does not require detailed knowledge of queueing-theoretic techniques or simultaneous resource possession. Domain experts with good knowledge of the application can therefore readily create a simulation model using CPN.

Queuing Petri Nets are used in [4] to model the performance of distributed component-based systems. That paper conducts a case study of the performance evaluation of a J2EE application server and then presents a performance evaluating method for modeling thread contention in a load balancer used with the application server. The focus in [4] is on modeling the number of threads in a thread pool for the load balancer. In contrast, our work models the thread contention caused from software locks and hardware resources, which is complementary to the work in [4].

Analytic performance models of software servers are developed in [7], which also studies the thread contention due to usage of thread pools. This paper develops a queueing-theoretic analytical model to obtain the optimal number of threads in a software server that uses a fixed number of threads in a pool. The underlying assumption in the use case is that each service provided by a thread does not contend with any other thread for software locks. Unlike our work with CPNs, that paper does not evaluate the problem of software contention due to software locks.

A Petri net model of an application is presented in [3], which captures software contentions and models software locks in a manner similar to ours. The main difference is that [3] does not consider the case of multi-level resource contention, *i.e.*, a thread performs its entire computation once it acquires a software lock. In contrast, in our approach a thread waits for a software lock and then contends for the hardware resource, which is more representive of common multi-threading scenarios.

Simulation-based performance of web servers [11] has created a simulation based model of a web server. That paper models physical resources, such as CPU, disk, and network, but does not consider the complex interation between software resources and hardware resources. In contrast, our approach also models both these resources.

6 Concluding Remarks

The work presented in this paper describes a technique we developed to model and simulate software contention. We used Colored Petri Nets (CPN) to validate the model data with the results captured by profiling the application. CPN models the non-determinism inherent in the case of multiple threads contending on a single lock. Profiling is performed to measure application runtime performance and the resulting data is validated against data predicted by the CPN model. The results show that the CPN model accurately predicts the pattern of behavior in the application within certain error limits.

We learned the following lessons based on our research conducted thus far:

• **The effect of using locks** in multi-threaded applications is not obvious. The use of a lock in one component can affect performance in apparently non-related components. It is seen from the profile data that locks can increase performance of some application components at the expense of others. For example, Section 3.4 showed that as the number of satellites grow in our case study the performance of the target thread improves, whereas the performance of the satellite and location degrades.

• **The optimal number of threads** in an application depends upon the application logic and the underlying hardware involved, which requires extensive dynamic and static analysis to determine. Section 4 showed how we could determine the optimal configuration of threads based on the

interaction of multiple threads, such as target and satellite, in a single processor system.

• **Petri net simulations** are helpful in pinpointing performance bottlenecks. Simulation techniques combined with profiled data can help predict and understand application behavior, which helps developers and deployers tune the proper number of threads to optimize performance. We developed a CPN model of our case study application that was calibrated using profiled data and which helped us to extract the optimal configuration of threads, as shown in Section 4.

• **The performance of multi-threaded applications is hard to predict when the underlying hardware changes**. For example, the throughput of the satellite threads degrades as the number of satellites grow, even when there are no locks in the system, as shown in Section 3.4. This behavior depends on various factors, such as the number of processors in the system, the priority of the threads, and the scheduling strategy used by the OS. Multi-processor and multi-core hardware infrastructure along with various scheduling mechanisms will be used for further experiments.

• **Capturing the effects of both hardware/software cache and OS effects (such as overhead of context switch or memory (de)allocation) is complex** using conventional profiling techniques. These effects are generally captured by monitoring system activity (such as performance monitoring on Windows, which is external to any application), so it is hard to connect them to application-specific events, such as the response time of the target thread or the throughput of the satellite thread, as discussed in Section 3.1. OS-specific settings, such as virtual memory allocation policies, also require further study since they determine the amount of cache effects present in the system. To ensure that model predicted performance is closer to actual performance, our future work will quantify these effects and included them in the CPN models.

• **Automated generation** of Colored Petri-net models is useful since manually creating Colored Petri net models is time consuming and hard to debug, calibrate, and validate. For example, it required a substantial amount of time to create the CPN model for the case study application described in Section 3.2. In future work we will therefore create automated tools to parse the code for the application and generate the CPN model. Likewise, CPN models of various hardware platforms, such as multi-core processors and disk drives, will be pre-constructed and combined with the application logic model to generate the complete CPN model.

• **Application bottlenecks due to software and hardware resource contention can be identified from CPN models of an application**. For example, Section 3.4 showed how satellite throughput degrades as the number of threads increased due to longer waiting time on software locks and the processor. Further analysis of model prediction data is required, however, to identify all critical sections and pinpoint where each thread spends most of its time. Our future work will therefore focus on identifying the bottlenecks in the system at the software and hardware levels.

The CPN model of the application and the application code used in this paper are available as open-source software from `www.dre.vanderbilt.edu/~nilabjar/SoftwareContention`.

Acknowledgements

We would like to thank Daniel Waddington for valuable comments and technical advice. This work was done as a part of the Software Technology Initiative (STI) project launched by Lockheed Martin Advanced Technology Laboratory, Cherry Hill, New Jersey.

References

[1] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use: volume 1*. Springer-Verlag London, UK, 1996.

[2] K. Jensen, L. M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254, 2007.

[3] K. M. Kavi, A. Moshtaghi, and D.-J. Chen. Modeling multithreaded applications using petri nets. *Int. J. Parallel Program.*, 30(5):353–371, 2002.

[4] S. Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, 2006.

[5] D. A. Menascé. Two-level iterative queuing modeling of software contention. In *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, page 267, Washington, DC, USA, 2002. IEEE Computer Society.

[6] D. A. Menascé, V. A. F. Almedia, and L. W. Dowdy. *Performance by design: Computer Capacity Planning by Example*. Prentice Hall, Upper Saddle River, NJ, 2004.

[7] D. A. Menascé and M. N. Bennani. Analytic performance models for single class and multiple class multithreaded software servers. In *Int. CMG Conference*, pages 475–482. Computer Measurement Group, 2006.

[8] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[9] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[10] H. Sutter. The free lunch is over: A fundamental turn towards concurrency in software. *Dr. Dobbs Journal*, 30(3), 2005.

[11] L. Wells, S. Christensen, L. M. Kristensen, and K. H. Mortensen. Simulation based performance analysis of web servers. In *PNPM '01: Proceedings of the 9th international Workshop on Petri Nets and Performance Models*

(PNPM'01), page 59, Washington, DC, USA, 2001. IEEE Computer Society.