

# 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 [7]. 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. In conventional multi-threaded systems, application developers and deployers make these decisions manually using 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 of 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 modeling language that extends Petri nets with a “color” for each token. 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 system components. Our work uses *CPN tools* [2], which help construct and analyze CPN models via an engine that conducts simulation-based performance analysis using the functional language Standard ML [4].

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.

## 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 involves a target-tracking simulation application composed of active objects [6], such as target, tracker, and satellites shown in Figure 1. There can be mul-

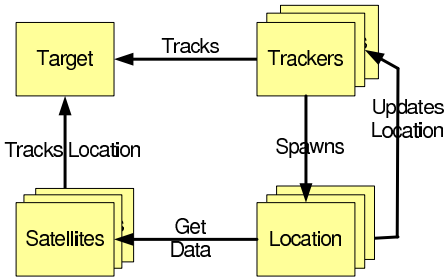


Figure 1. Active Objects in Target Tracker

multiple instances of trackers and satellites; each tracker collects the target’s latest location from a satellite. To increase the probability of finding the target, the application must be configured with the right number of trackers and satellites.

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 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 active object has its own control flow and can block contending for shared data with other objects. We define the following active objects in our application case study (shown in Figure 1):

- **Target**, which simulates a target that moves through an 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 sleeping, it moves in a particular direction with designated velocity. There is one instance of the target in the application.

- **Satellites**, which gather information of the latest position of the target. Within the application, the latest coordinates of the target is placed in a global variable that each satellite reads periodically.

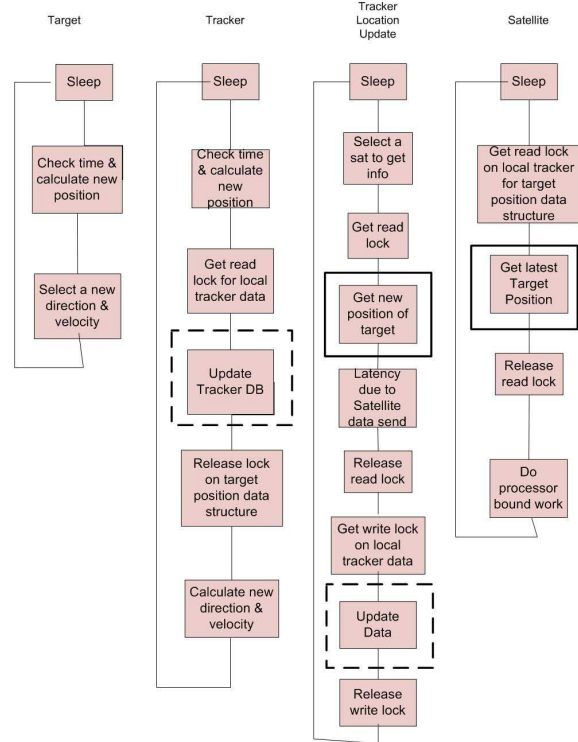


Figure 2. Application Logical Flows in the Target Tracking Simulator

- **Trackers**, which pursue the target by obtaining its latest position via the location objects described below. Each tracker recalculates its new direction and velocity every period depending on the target’s latest position. It also checks if it “hits” the target, *i.e.*, if its current position is within some small distance of the target.

- **Tracker location updates**, which are created by trackers 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. Each pair of satellite and tracker objects are associated with a location active 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 in-

creased 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  $2 \times 3 = 6$  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. The target completes one iteration of sleep and computation per period, as shown in Figure 2. The application runs two scenarios: (1) with all locks and (2) with no 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 key 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 tell if the average throughput of the tracker decreases or increases when a particular configuration is changed. The magnitude of the difference is less 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  $\mu_{tr}$  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  $\mu_{loc}$  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  $\mu_{sat}$  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:

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

The configuration that maximizes the value of this function should provide the preferred application setting, which can be computed by predicting tracker, location, and the satellite throughput and using them in the above equation.

## 3 Experiments

This section discusses how we created a model of the application case study described in Section 2 and validated the model against profiled data.

### 3.1 Application Profiling

**Experiment design.** Our application case study is profiled under various thread configurations to collect performance data we use to calibrate and validate the simulation model. The experiments run on a single CPU, Intel Pentium, 1.70 GHz machine with 1 GB RAM. The OS is Windows XP Professional Version 2002 with service pack 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  $N_{loc}/T_{tg}$ , where  $N_{loc}$  is the number of location iterations.

To capture the throughput and response time of different threads, we profiled the activities of their associated active objects. Application methods of the target object were instrumented to include timestamp recording. We also 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.

**Analysis of results.** The results in Table 1 show variability which is non-intuitive. 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 ob-

Config	With Mutex				Without Mutex			
	Target run time (secs)	Satellite Throughput (per-ods/sec)	Tracker throughput (per-ods/sec)	Location Throughput (per-ods/sec)	Target run time (secs)	Satellite Throughput (per-ods/sec)	Tracker Throughput (per-ods/sec)	Location Throughput (per-ods/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

jects 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.

Such 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 variability arises either from cache effects or operating system jitter. Caching could cause this difference since 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.

We did not use any real time scheduling in our experiments so fluctuations in performance could also arise from OS jitter. Petrini et al [5] and Kramer et al [3] show how OS jitter can cause variability in performance. For simplicity, we will use the term “cache effects” or “OS jitter” to refer to such variability in the paper.

### 3.2 Colored Petri Net Model Construction

We now explain the simulation model of the application case study using Colored Petri Nets (CPNs). Figure 3 shows a screenshot of the CPN tool and our application modeled using CPN. The four aspects of the application that are part of the system modeling process include (1) *modeling application flow*, which models the logic of each object similar to the workflows shown in Figure 1, (2) *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,” (3) *modeling resource access*, which models the concurrent access of the physical resources by each thread, and (4) *modeling cache effects/OS jitters*, which models the variability in computation time due to simultaneous threads performing similar work on the CPU. Below, we elaborate on the modeling of these four aspects.

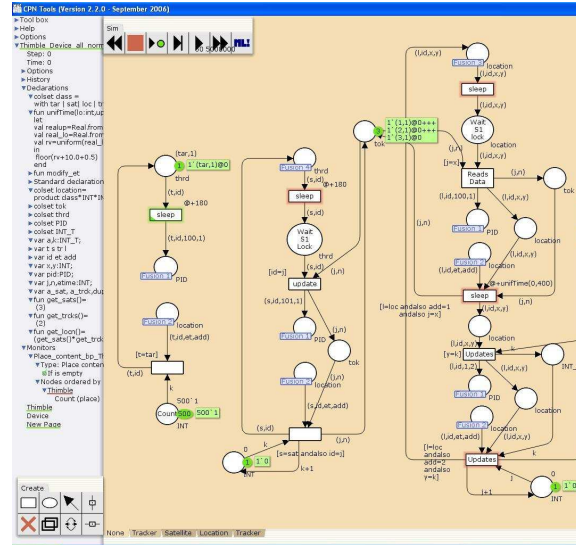


Figure 3. CPN Model of Application Case Study

#### 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

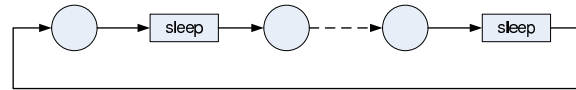


Figure 4. A CPN Model of Target’s Active Object Thread

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 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 software lock, which is available if a token is present in that

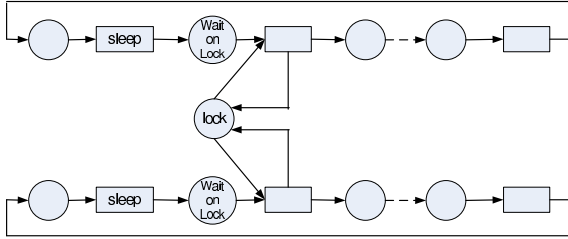


Figure 5. Contention Model for Software Lock

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.

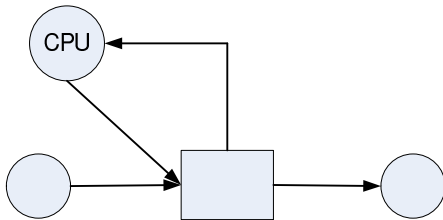


Figure 6. The CPN Model of CPU

### 3.2.4 Modeling Cache Effects/OS Jitters

Cache effects/OS jitters 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 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 ‘tint’ is less than 180 the execution time is modified to 94% of the original. In the extreme, if it is

```

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 for Cache Effects

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. The various percentage values were tweaked multiple times until the response time of the target thread in the model converged to the empirical data.

### 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,

Config	Target run time (secs)	With Mutex			Without Mutex			
		Satellite Throughput (periods/sec)	Tracker Throughput (periods/sec)	Location Throughput (periods/sec)	Target run time (secs)	Satellite Throughput (periods/sec)	Tracker Throughput	Location Throughput (periods/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 from profiling the application (Section 3.1) with the model prediction results (Section 3.2). These results are explained from the perspective of two conflicting factors: (1) the CPU 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

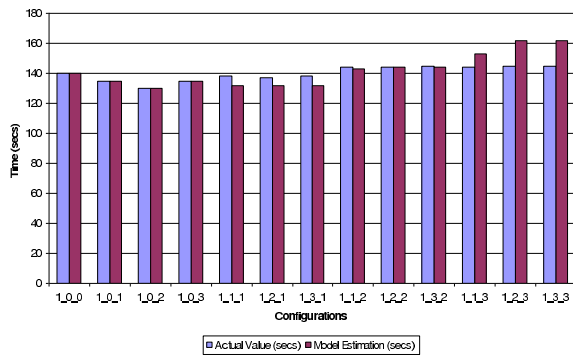


Figure 8. Response Time of Target Thread with Locks

occurs for two reasons (1) the target does not contend with other objects, so it does not face any extra blocking as the number of other objects increases and (2) 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 satellite thread behavior with locks in the system. Each set of three configurations in this graph (*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

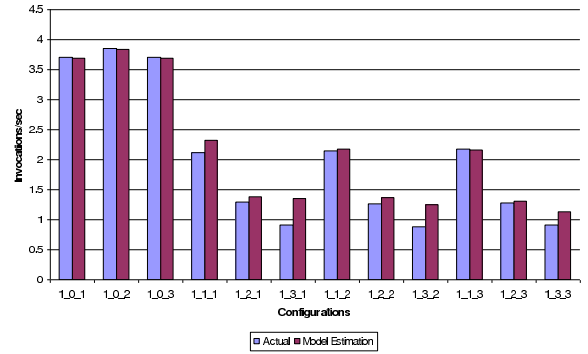


Figure 9. Throughput of Satellite Thread with Locks

in Figure 10.

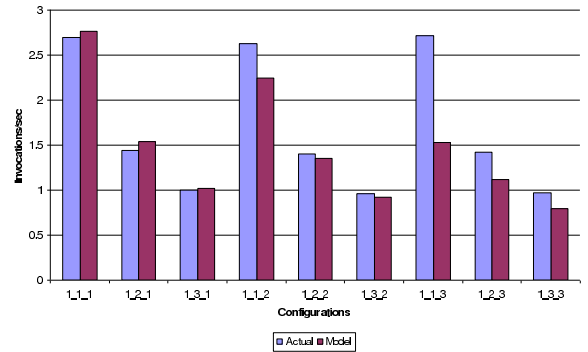


Figure 10. Throughput of Location Thread with Locks

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 application behavior trend helps identify the optimal thread configuration. The accuracy of the prediction is less important since we are only interested in determining if 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

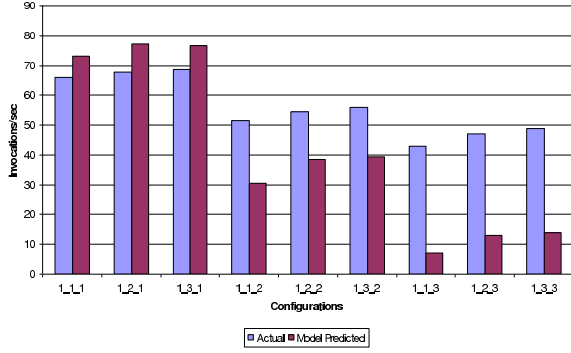


Figure 11. Throughput of Tracker Thread with Locks

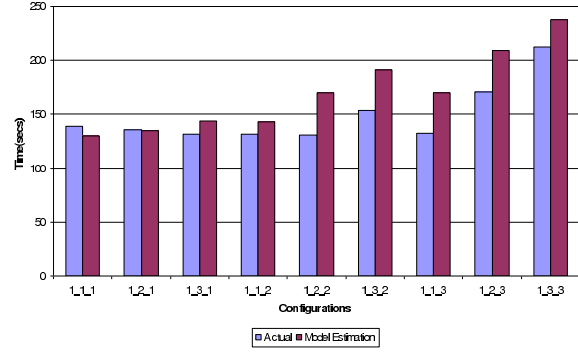


Figure 13. Response Time of Target Thread without Locks

data in the Figures 13, 14 and 12.

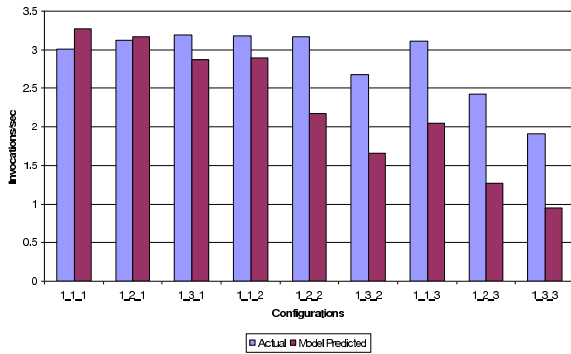


Figure 12. Throughput of Location Thread without Locks

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 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 ac-

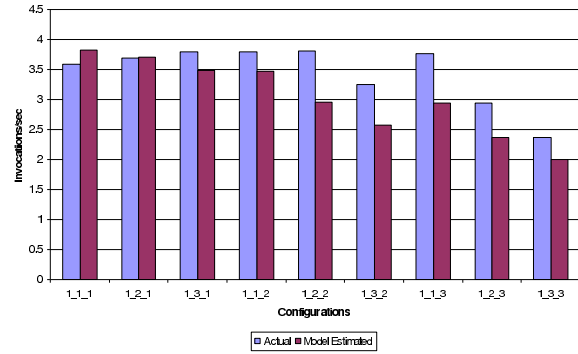


Figure 14. Throughput of Satellite Thread without Locks

tual 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 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) from Section 2 to compute the hit chance value for each configuration, as shown in Table 3. The average throughput values of tracker and satellite are used from the model predicted data in Table 2. Table 3 shows configuration 1\_3,1 maximizes the trackers hitting the target, as explained in Section 2, so this configuration should thus be optimal. To verify whether this configuration is optimal, the running application was then profiled

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 Various Configurations**

to record the number of times the trackers hit the target, as shown in Table 4.

We also needed to verify the validity of Equation(1) as a right quantifier of the application performance. We therefore used the measured value of tracker, location, and satellite throughput to compute the value of the equation for each configuration (omitted due to lack of space). Using these values we ranked each configuration and it matches exactly with the ranking given by the data from actual hit counts(Table 4) except one configuration, 1\_2\_3. This result proves that Equation(1) is a reasonable estimator of the application performance.

Config	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. Runtime Target Hit Occurrences**

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. Comparing the data shown in Table 3 and Table 4, we see quite a few discrepancies in the ranking of the configurations. This is due to the error in the prediction of the throughput of the various active objects. If

the error is reduced, the prediction will be more accurate.

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

## 5 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.

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](http://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 at Lockheed Martin Advanced Technology Laboratory.

## 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] W. Kramer and C. Ryan. Performance variability of highly parallel architectures. *Proceedings of the International Conference on Computational Science (ICCS 2003)*, 2659:560–569.
- [4] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.
- [5] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asc q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.
- [6] 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.
- [7] H. Sutter. The free lunch is over: A fundamental turn towards concurrency in software. *Dr. Dobbs Journal*, 30(3), 2005.