Using the Parallelism Viewpoint to Optimize the Use of Threads in Parallelism-Intensive Software Systems

Naeem Muhammad, Nelis Boucke, and Yolande Berbers

***Abstract*—The use of multithreading can enhance the performance of a software system. However, its excessive use can degrade the performance. For example, a thread-per-job approach might lead to a large amount of threads with increased associated overheads.**

In this paper we explore the use of the Parallelism Viewpoint to support one possible strategy to reduce the number of threads, namely finding candidate threads that can be replaced by thread pooling. Thread pooling reduces the large number of threads by reusing threads from an existing pool. As an example we analyze the threads of a precision critical parallelism-intensive electron microscope software system.

Results show that the viewpoint provides a profound insight into the threading structure of the system, which helps in reducing the number of threads in a cost-effective way. And, the total time gain along with such reduction is encouraging.

***Index Terms*—Multithreading, architecture viewpoint, parallelism viewpoint, software performance, thread pooling.**

I. Introduction

Multithreaded applications are considered to be more efficient because of their better software and hardware resource utilization provided by the parallel execution of tasks. Despite potential benefits, system designers should be careful while designing a thread model of the system. The excessive use of threads can degrade the system performance by enlarging the associated overheads [1]. Among these overheads are thread creation and deletion, context switching and increased thread management complexity. In this paper, we focus on the thread creation and deletion overheads.

These overheads can be diminished by using thread pooling, which is an efficient multithreading technique. In thread pooling, a set of worker threads is created at system startup and is reused for various tasks. An optimal use of this technique is possible by using worker threads for shorter tasks.

We use the Parallelism Viewpoint to identify threads in parallelism-intensive legacy systems that can be replace by a pool of threads. The Parallelism Viewpoint is an architecture viewpoint supporting visualizing and analyzing the parallelism of a system.

The description includes identification of parallelism specific concerns, corresponding stakeholders and a set of model kinds to model those concerns. The general information on this viewpoint is described in a technical report [2]. This paper adds a detailed example of using the viewpoint to identify potential threads for pooling.

Manuscript received June 13, 2012; revised August 29, 2012.

Naeem Muhammad, Nelis Boucke and Yolande Berbers are with the Department of Computer Science, Katholieke Universiteit Leuven, Belgium (e-mail: naeemmuhammad@gmail.com).

The motivation behind using the Parallelism Viewpoint for thread analysis is two-fold. First, architecture level analysis is a proven cost-effective approach that provides an opportunity to find and fix issues up-front in the system development [3]. Second, the Parallelism Viewpoint provides an in-depth understanding, not only about the thread behaviour but also about associated concerns and stakeholders. Such understanding is essential while making any change in the thread model.

In this paper, we describe with the help of an industrial case how the Parallelism Viewpoint can be used to identify threads that are suitable to be replaced with a thread pool. The case is a large and complex parallelism-intensive software system used for electron microscopes. It is a client-server distributed system whose design follows a component-based architecture. It runs on the Microsoft Windows XP operating system. Because of the heterogeneous nature of the machine, its devices come from multiple domains such as electronics, mechanics and physics. The software is responsible for data acquisition and control of these devices. It has a large code base with multi-million lines of code and employs several hundred threads to perform various microscopy functions. We believe that the performance of the software can be enhanced by reducing the number of threads. This can be achieved by replacing them with a small sized thread pool, provided that accurate threads are selected for such replacement.

The remainder of this paper is organized as follows. In the next section we briefly describe the building blocks of Parallelism Viewpoint. We outline the analysis approach in section III and apply it on an industrial case in section IV. Section V contains related work, and finally in section VI we draw conclusions and state our future work.

II. Parallelism Viewpoint

The Parallelism Viewpoint is a domain-specific form of the concurrency viewpoint. The concurrency viewpoint provides support mainly for describing concerns related to the communication and synchronization mechanisms of the concurrent systems [4]. The Parallelism Viewpoint extends this support for concurrent systems by providing support to describe parallelism behaviour.

Essentially, a viewpoint must explicitly describe the concerns of a particular domain, identify the stakeholders of these concerns and specify a set of model kinds [5]. In compliance to these requirements the description of the Parallelism Viewpoint consists of parallelism specific concerns, corresponding stakeholders and a set of five model kinds to model these concerns.

Following are the concerns for thread pool analysis, which we identified through extensive interaction with various stakeholders of the electron microscope system, domain experts and researchers from the parallelism domain.

Number of tasks: Total number of tasks (operations) performed by a thread in some selected scenarios.

Total time: Total time consumed by a thread during its life in some selected scenarios.

Number of threads: Total number of threads employed by a system to perform some functionality in some selected scenarios.

Thread active/idle behaviour: Active and idle pattern of a thread during its life cycle for some selected scenarios.

System architects, developers and testers are among the stakeholders who hold these concerns. System architects, while designing or altering the thread model should have a clear understanding about these concerns. It is vital for a developer to recognize these concerns as he is responsible for the actual realization of the thread model. Testers, on the other hand, look into the system for these concerns to identify possible performance bottlenecks caused by threads.

The model kinds include: Time distribution, Task distribution, Thread behaviour, Task types and Thread management. We make use of the first three model kinds in this research work because they address the above concerns. In the following subsections we briefly describe these models whereas a comprehensive description covering all aspects of the viewpoint is given in [2].

A. *Time Distribution Model Kind*

Threads which are the basic units of execution use their quota of CPU time to perform their tasks [6]. Time distribution is a model kind that illustrates the total time used by every thread in a system over a period of time. Since the devised approach is scenario based, a single instance of this model kind shows the time distribution across threads for a particular scenario. Stakeholders can use this model to analyze a system for the *total time* and *number of threads* concerns. Fig. 1. shows an instance of this model kind. Along the horizontal axis it shows the threads running in the system, whereas a vertical bar represents the total amount of time consumed by a thread.

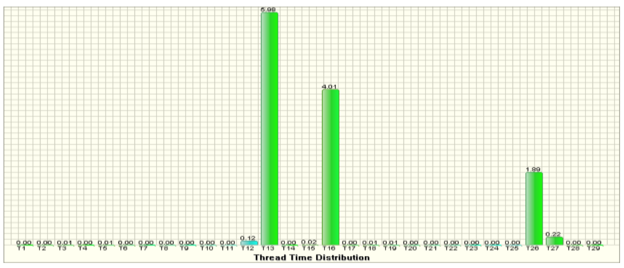

Fig. 1 Time distribution model kind

B. *Task Distribution Model Kind*

A system makes use of multiple threads to distribute its workload. The task distribution model kind portrays this distribution. It shows the total number of tasks performed by every thread of the system. Similar to the time distribution, this model kind also depicts distribution for a particular scenario. Primarily, it addresses the *number of tasks* concern in the viewpoint. Task distribution can be analyzed to identify threads performing too many tasks and those with a very small number of tasks. Fig. 2. contains an instance of the task distribution model kind. The horizontal axis shows threads in the system, whereas vertical bars represent the total number of tasks performed by each thread.

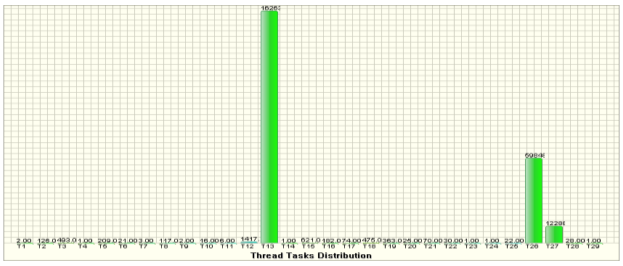

Fig. 2. Task distribution model kind

C. *Thread Behaviour Model Kind*

Along with the overall distribution of time across threads, it is important to understand the active/idle behaviour of the system threads. This determines how important a thread is, at least from the timing perspective. Thread behaviour is a model kind that portrays this behaviour by showing activities of a single thread performed during its life cycle. It is mainly used for analyzing a system for the *thread active/idle behaviour* concern of stakeholders.

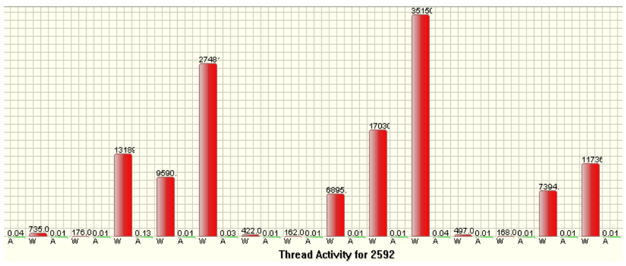

Fig. 3. Thread behaviour model kind

Similar to the previous model kinds it also illustrates thread behaviour based on a particular scenario. An instance of this model kind is shown in Fig. 3. Horizontally, the model kind shows the sequence of Active (A) and Waiting (W) times of a thread whereas vertically it represents the total active and waiting times.

III. Using the Parallelism Viewpoint for Thread Pool Analysis

Fig. 4. illustrates an overview of our analysis approach. The approach primarily consists of two processes, thread pool analyzer and validation. These processes utilize the Parallelism Viewpoint models to prepare a prioritized list of threads suitable to be replaced with a thread pool. Hereunder we discuss these processes, their inputs and outputs.

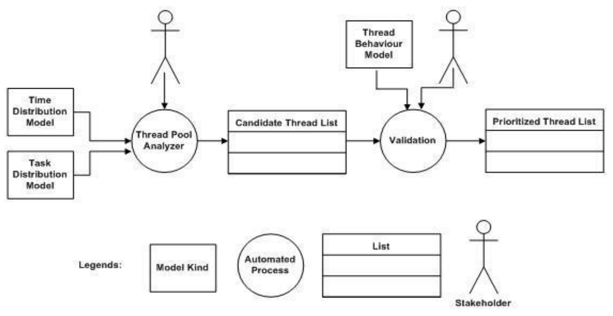

Fig. 4. Thread pool analysis approach

The analysis starts with developing the time distribution, task distribution and thread behaviour models; the process of developing them is described in [2]. Instances of these models for our example research case are shown in figure 1, 2 and 3.

We propose developing these models for multiple and related scenarios, in order to better understand the thread behaviour [7]. Multiple scenarios are important to understand the behaviour for various user actions. Whereas related scenarios help in distinguishing threads performing similar tasks from others. Such distinction is important and is used while designing thread elimination strategy.

Thread pool analyzer: An automated process that uses the above models to prepare a candidate list of suitable threads. The list is prepared based on the total CPU time consumed and the number of tasks performed by a thread.

An optimal use of thread pooling is possible with small sized tasks [13]. Furthermore, worker threads of a pool are efficient when used for a short period of time. Therefore, in this process the analyzer filters threads consuming a small amount of total time and performing very few tasks. Assigning values to these parameters is very specific to the application under investigation. The input from stakeholders discussed in the previous section is needed for this purpose. In this paper, for our example case we set a 100 milliseconds (ms) limit for the total time consumed and 100 for the total number of tasks performed per thread. These values are very small as compared to the average total time consumed and the total number of tasks performed by the threads in the system.

The candidate list consists of all the threads meeting these timing and number of task limits.

Validation: The validation process uses the thread behaviour model, developed for every thread in the list in the first step, to assign it a priority. For every thread, its behaviour models from all scenarios are analyzed together to find any change in its behaviour.

In principle all threads in the list can be replaced with a pool consisting of a small number of threads because they meet the limits set in the previous step. However, these threads may vary in the degree of change in their behaviour for different scenarios. A thread is perfectly suited for elimination if its behaviour remains constant for every scenario, otherwise not. We introduce three levels of priorities for threads, which are assigned based on the level of change in their behaviour for all selected scenarios. Table I contains the description of these priorities.

TABLE I: PRIORITY CRITERIA

Priority	Description
3	No change in thread behaviour (task & time) for all scenarios
2	Small increment in the total number of tasks, regardless of any change in the total time
1	Major increment in the total number of tasks, regardless of the change in the total time.

We assign priority 2 to those threads that have less increment in their tasks. Because, any increase in the number of tasks introduces extra thread scheduling, that represents an increase in thread activity. A thread with a major increment in the number of tasks will be assigned priority 1.

We do not consider the total time as a varying factor because an increase in it will not produce any overhead as compared to any change in the total number of tasks. It is likely that a thread may consume additional time for one or more of its tasks, without requiring extra CPU allocation. The change in the total time and total number of tasks does not however exceed the defined limits, for our example case 100 ms and 100 tasks.

The outcome of this process is a prioritized list of threads, which can be used to eliminate threads in the list and replace them with a pool having a smaller number of threads. A thread with a higher priority of 3 is most suitable for elimination whereas a priority 1 thread is least fit. Priority 2 threads, depending upon the agreement among stakeholders identified in section II, can either be eliminated along with priority 3 threads or spared.

IV. THREAD ANALYSIS APPLICATION

In this section we describe the use of the proposed analysis for our example case and discuss its results.

We used a set of three related scenarios, (scenario A) system startup, (scenario B) moving the specimen to a certain position in the microscope and (scenario C) bringing it back to its home position. The startup scenario is significant because it helps in identifying unnecessary initialization of threads at inception stage. The other two representative scenarios represent a very important function of an electron microscope. That is, to move the specimen to various positions.

As stated earlier the analysis starts with developing the time and task distribution models. Figure 1 and 2 show two of these models we developed for scenario B, for the electron microscope software. The models provide clear understanding about the time and task distribution of the system for scenario B. We can observer from these models that most of the threads are consuming no CPU time. Furthermore, majority of the threads are performing very few tasks.

The thread pool analyzer uses these models to prepare a list of candidate threads. The list contains threads which consumed a total time less than or equal to 100 ms and performed a total number of task less than or equal to 100. The outcome of this process is a list containing all the threads meeting these limits, shown in Table II.

TABLE II: NUMBER OF THREADS MEETING THE TIME AND TASK LIMITS

	Scenari o A	Scenari o B	Scenario C	Tota l
Total Threads	17	29	30	76
Threads (time<=100ms & task<=100)	13	17	17	47

We can observe from the candidate list that the total number of threads in the system increases as the utilization of the system advances. Interesting to note is the increase in the number of candidate threads, pointing out amplification of the thread creation and deletion overheads. In every scenario we find more than 50% threads meeting the set time and task limits. This behaviour remains consistent even when tightening the limits.

The validation process assigns a priority to every thread in the candidate list by monitoring any change in its behaviour in all three scenarios. Such changes are monitored by using the thread behaviour model; figure 3 shows an instance of the model of our example case for scenario B.

Table III contains a prioritized list for the electron microscope software. Thread ID represents the actual id of the thread given by the operating system whereas the Priority column represents the assigned priority.

TABLE III: PRIORITIZED THREAD LIST

Thread ID	Priority	Thread ID	Priority
708	3	3552	3
1056	3	3776	1
1660	3	3796	2
2528	3	3808	2
2536	3	3952	3
2560	3	3964	3
2592	3	3996	2
3652	3	4080	3
3676	3		

The majority of the threads in the list have priority 3, representing that their behaviour remains constant. For thread 3776 we found a major increase in the number of tasks it performed. It performed 5, 25 and 35 tasks in scenario A, B and C respectively. We assigned it priority 1, therefore is not eligible for replacement. Threads 3796, 3808 and 3996 have a small increment in the number of tasks they performed as compared to the priority 1 thread. Therefore, they are assigned priority 2. Threads 3808 and 3996, although have priority 2, we observed that the change in their number of tasks is very small. Therefore, we find it suitable to replace these threads along with priority 3 threads. Thus, in total we find 15 threads suitable to be replaced with thread pooling. We show a limited prioritized thread list in table 3. In the complete list, more than 60% of the threads hold priority 3. Note that, any change in thread behaviour in the case of priority 1 and 2 threads, always remains within the defined time and task limits.

Now that we have identified threads that can be replaced by a pool of threads, the next step is to find a suitable size of the pool. Finding an optimal size to maximize the expected gain is still a challenging task.

We estimate the pool size based on the total number of tasks performed by all the eligible threads and the average number of tasks performed by a thread in the system. For our example case, the total number of tasks performed by all 15 threads eligible for replacement is 209. And, the average number of tasks performed by a thread in the system is 268.

The number of total tasks performed by all the threads that are to be replaced is less than the average number of tasks performed by a thread in the system. In principle a single thread can be employed to perform 209 tasks. However, in order to consider the concurrent execution of tasks we must have at least two threads. Therefore we can replace in total 15 threads (with priority 3 and 2) with a pool of 2 threads.

We can estimate the total gain by using a formula given in [8]. Total Gain= c1.r-c2.n. Where c1 represents the thread creation and deletion time, c2 is the time taken for a single context switch, r is the current total number of threads in a system and n represents the number of threads in a pool. In our case r =15, the total number of threads we want to replace, and n =2 as we use a pool of two threads. Thus, Total Gain = c1.15-c2.2.

The actual value of r for the electron microscope is very high as the total number of threads in it reaches several hundreds and we found that the number of threads eligible to be replaced is proportional to the number of threads in the system.

Considering the fact that the context switch overhead is less than the thread creation and deletion overhead (c1> c2) we can observe a clear gain from the equation. The gain boosts as the number of threads in the system increases, such as in the case of our example system.

The results of the analysis indicate that the Parallelism Viewpoint provides a profound insight into the thread structure of the system. Such insight can be used to analyze a system for many threading related concerns.

V. RELATED WORK

Performance optimization of multithreaded software systems is a well-established research area. A variety of optimization methods and techniques looking at different aspects of performance are available and/or being developed. Flanagan et al. [9] proposed a modular approach called Calvin for analyzing the thread behaviour of multithreaded software systems. They analyze the system behaviour by performing modular checking of each procedure call made by threads present in the system. Also, Li and Malony [10] diagnose the performance bottlenecks of parallel applications with the help of a model-based diagnosis framework called Hercule. In this paper, we also analyze the runtime behaviour of threads to optimize the software performance. In contrast to the above two approaches we perform the architectural level analysis by using the Parallelism Viewpoint. We believe that analyzing non-functional requirements at architecture level is a cost-effective approach.

To achieve similar goals, Dean and Shen [11] presented an approach for integrating existing threads in order to reduce the total number of threads. In their work, they improve the performance of the system by overlapping the execution of multiple threads. To improve the performance by reducing the number of threads, our and Dean and Shen's research work require a change in the thread model of the system. However, our work differs in that we replace existing threads with a pool of new threads instead of integrating them.

Raissi's research work given in [12] reinforces our proposal of utilizing thread pooling to enhance software performance. He analyzes the impact of using thread pooling for a cryptography framework called DSOCARE. The results of the study show that the performance improved by 78%. Such a high improvement however depends upon the right size of the pool. Zabatta and Ying [8] characterized the factors associated with the thread pool size. They provide an analytical method to determine an optimal pool size and a fairly simple way of calculating the total performance gain. We utilized their formula to calculate the performance gain for the electron microscope software system.

VI. Conclusions and Future Work

In this paper we explored the use of the Parallelism Viewpoint to optimize the use of threads in legacy software systems. We put forward the use of a thread pool instead of thread-per-job. Based on the time utilization and number of tasks performed characteristics, we identified threads appropriate to be replaced with a small sized thread pool. We successfully applied our approach on an industrial case, a parallelism-intensive electron microscope software system. The generic nature of the viewpoint models that we used in this paper makes our approach applicable to other systems as well. The results encourage us to explore the use of the Parallelism Viewpoint to analyze parallelism-intensive software system for various aspects of threading.

The two level filtering used in this paper makes sure that the right threads are picked for possible replacement with a thread pool. We found that the number of threads suitable for such replacement increases as the use of the system advances.

We conclude that the number of threads in a system can be reduced to a large extent by employing the thread pooling technique. And, the total gain in performance is encouraging. The real benefit however lies in identifying the optimal size of the pool, which remains a challenging job.

We identified the pool size based on the total number of tasks performed by the threads in the list and the average number of tasks performed by a thread in the system. In our future work we will extend this technique to identify a precise pool size. Furthermore, as a part of our research work, we are also building a flow-latency viewpoint to describe latencies of flow-intensive software systems. Our future work involves studying the impact of implementing thread pooling on latencies of the system flows.

Acknowledgment

This work has been carried out as a part of the Condor project (http://www.esi.nl/Projects->Condor) at FEI company under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

We would also like to thank our colleagues Sjir van Loo, Dries Langsweirdt and Auke van Balen for their valuable inputs and feedback for this research work.

References

[1] D. Xu and B. Bode, “Performance Study and Dynamic Optimization Design for Thread Pool Systems,” presented at the International Conference on Computing, Communications and Control Technologies, 2004.

[2] N. Muhammad, N. Boucke, and Y. Berbers. (2010). Parallelism Viewpoint: An Architecture Viewpoint to Model Parallelism Behaviour of Parallelism-Intensive Software Systems. [Online]. Available: http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW589.abs.html

[3] P. Clements, Kazman, and R. M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Boston, MA: Addison-Wesley, 2002.

[4] N. Rozanski, and E. Woods, Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Addison-Wesley, 2005.

[5] ISO/IEC 42010, “Systems and Software Engineering –Recommended Practice for Architectural Description Of Software-Intensive Systems,” 2007.

[6] William Stallings; Operating System: Internals and Design Principles. Pearson Higher Education, 2009.

[7] S. S. Somé, and X. Cheng, “An Approach For Supporting System-Level Test Scenarios Generation From Textual Use Cases,” in *Proc. of the ACM Symposium on Applied Computing, ACM, 2008*, pp. 724-729, New York.

[8] F. Zabatta, and K. Ying, “A Thread Performance Comparison: Windows NT and Solaris on A Symmetric Multiprocessor,” presented at the 2nd USENIX Windows NT Symposium, 1998.

[9] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia, “Modular Verification of Multithreaded Programs,” *The oretical Computer Science*, vol. 338, no. 1-3, June 2005, pp. 153-183.

[10] N. L. Li and A. D. Malony, “Automatic Performance Diagnosis of Parallel Computations with Compositional Models,” presented at IEEE International Parallel and Distributed Processing Symposium, 2007, pp. 211.

[11] A. G. Dean and J. P. Shen, “Techniques for Software Thread Integration in Real-Time Embedded Systems,” in *Proc. of the IEEE Real-Time Systems Symposium, IEEE Computer Society,* 1998, pp. 322.

[12] J. Raissi, “Performance Impact of Thread Pooling in DSOCARE,” in *Proc. of the IEEE,* SoutheastCon, April 2005, pp.108-113.

[13] L. Yibei, M. Tracy, and L. Xiaola, “Analysis of Optimal Thread Pool Size,” *ACM SIGOPS Operating Systems Review*, vol. 34 no. 2, April, 2000, pp. 42-55.