Journal of Computer Science 7 (6): 884-891, 2011
ISSN 1549-3636

Dynamic Allocation of CPUs in Multicore Processor for Performance Improvement in Network Security Applications

Sudhakar Gummadi and Radhakrishnan Shanmugasundaram
Department of Computer Science and Engineering,
Arulmigu Kalasalingam College of Engineering,
Anand Nagar, Krishnankoil-626190,
Tamil Nadu, India

Abstract: Problem statement: Multicore and multithreaded CPUs have become the new approach for increase in the performance of the processor based systems. Numerous applications benefit from use of multiple cores. Increasing performance of the system by increasing the number of CPUs of the multicore processor for a given application warrants detailed experimentation. In this study, the results of the experimentation done by dynamic allocation/deallocation of the CPU based on the workload conditions for the packet processing for security application are analyzed and presented. **Approach:** This evaluation was conducted on SunfireT1000 server having Sun UltraSPARC T1 multicore processor. OpenMP tasking feature is used for scheduling the logical CPUs for the parallelized application. Dynamic allocation of a CPU to a process is done depending on the workload characterization. **Results:** Execution time for packet processing was analyzed to arrive at an effective dynamic allocation methodology that is dependant on the hardware and the workload. **Conclusion/Recommendations:** Based on the analysis, the methodology and the allocation of the number of CPUs for the parallelized application are suggested.

Keywords: Logical CPUs, OpenMP, packet processing, performance analysis, dynamic allocation, parallelized application, parallel programming

INTRODUCTION

With the rapid development of chip multiprocessing techniques, multicore architecture has become more and more widely used in intensive computing applications as well as in computer networking systems. The amount of improvement in performance by the use of a multicore processor is dependent on the software algorithms and their implementation. The possible gains are limited by the part of the software that can be parallelized to run on the multiple cores simultaneously; as proposed by Amdahl's law. Scheduling of parallel activities on the multicore processor is very vital to improve the performance of the system. The underlying hardware of the multicore processor has to be effectively used to obtain the optimum performance of the system.

Multithreaded processor supports concurrent thread execution at the more fine-grained instruction level, aiming at better utilizing the resources of processor by issuing instructions from multiple threads. Multicore processors achieve thread concurrency at a higher level, focusing less on utilization per core and aiming at scalability via replicating cores (Sodan *et al*., 2010). A multicore processor (or Chip-level Multiprocessor, CMP) combines two or more independent cores into a single package of an Integrated Circuit (IC). A multicore processor implements multiprocessing in a single physical package (Lee and Shakaff, 2008).

The per chip core counts are increasing significantly. For example, Oracle's SPARC T3 processor features up to 16 cores and 128 threads on a single chip with integrated logic for 1GbE networking and cryptographic coprocessor engines (http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/sparc-t3-chip-ds-173097.pdf). Octeon® II CN6880 of Cavium Networks is a 32 core processor with over 85 application acceleration engines that provides high-performance, high throughput solution for intelligent networking applications (http://www.caviumnetworks.com/OCTEON_MIPS64.html). Programming of the multithreaded multicore

Corresponding Author: Sudhakar Gummadi, Department of Computer Science and Engineering, Arulmigu Kalasalingam College of Engineering, Anand Nagar, Krishnankoil 626126, Virudhunagar District, Tamil Nadu, India Tel: +91-4563-289129/+91-9842115148 Fax: +91-4563-289322

processor needs a thorough understanding of the hardware and the effective use of the Application Program Interface (API) for parallel programming. The task partitioning in multicore processors is done based on the application requirement and the time taken for execution of these tasks. OpenMP API is one of the parallel programming models used to exploit the available parallelism of multicore processors.

In a multicore environment, CPUs or a set of CPUs can be assigned to a particular process. Dynamic assignment of the CPU to the process based on the workload characteristics improves the overall performance of the system. Proper performance indicators need to be used for simulation, testing and realization of multicore implementations. Parallelization of a network security application is considered for generating the load and analyzing the performance of the system.

Multithreaded multicore processor architecture: The UltraSPARC T1 is a chip multicores/multi-threads processor that contains 8 cores and each of the SPARC cores has 4 hardware threads. A single pipeline processes instructions from four threads and completes one instruction in each cycle. All together, the chip handles 32 hardware threads and is addressed as 32 logical CPUs (Sun Microsystems, 2006; Leon *et al*., 2006).

Each SPARC core has a 16 KB, 4-way associative, 32B line size of Level 1 instruction cache (I Cache), 8 KB, 4-way associative, 16B line size of Data Cache (D Cache), 64-entry fully associative instruction TLB (Translation Lookaside Buffer) and 64-entry fully associative data TLB that are shared by the four hardware threads. The eight SPARC cores are connected through a crossbar to an on-chip unified 3 MB, 4-way associative L2 cache (64B lines). The L2 cache connects to 4 on-chip DRAM controllers, which directly interface to DRAM interface.

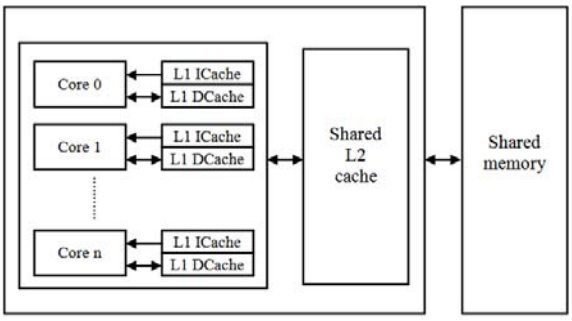

Fig. 1: Simplified block diagram of multicore processor with external shared memory

Figure 1 shows a simplified block diagram of the multicore processor wherein each core has separate L1 instruction cache and L1 data cache. All the cores share the common L2 cache with external shared memory.

Each hardware thread of UltraSPARC T1processor has a unique set of resources in support of its execution. The per-thread resources include registers, a portion of I-fetch datapath, store buffer and miss buffer. Multiple threads within the same SPARC core share a set of common resources in support of their execution. The shared resources include the pipeline registers and data-path, caches, Translation Lookaside Buffers (TLB) and execution unit of the SPARC core pipeline.

Thread switching takes place during every SPARC core clock cycle. At the time of a thread selection, the priority is given to the least recently executed yet available thread. Load instructions will be speculated as cache hits and the thread executing a load instruction will be deemed as available and allowed to be switched-in with a low priority. Because of shared structures like caches, the performance of a thread is also affected by other threads running on the same core.

UltraSPARC T1 processor has one Modular Arithmetic Unit (MAU) per core that supports modular multiplication and exponentiation. The hardware thread that initiated the MAU stalls for the duration of the operation, but the other three threads on the core can progress normally. The eight MAUs (one per core) result in very high throughput on UltraSPARC T1 processor-based systems for encryption operations.

OpenMP: The OpenMP™ Application Program Interface is a portable, parallel programming model for shared memory multithreaded architectures (Sun Microsystems, 2009; Chapman *et al*., 2009). OpenMP specification version 3.0 introduces a new feature called tasking. By using the tasking feature, applications can be parallelized where units of work are generated dynamically, as recursive structures or while loops. The task directive defines the code associated with the task and its data environment. The task construct can be placed anywhere in the program and whenever a thread encounters a task construct, a new task is generated.

When a thread encounters a task construct, it may choose to execute the task immediately or defer its execution until a later time. If task execution is deferred, then the task is placed in a conceptual pool of tasks that is associated with the current parallel region. The threads in the current team will take tasks out of the pool and execute them until the pool is empty.

Ayguade *et al*. (2009) have evaluated the performance of the runtime prototype with several applications using OpenMP tasking feature and have measured the performance in terms of the speedup for different number of CPUs and have proved that

OpenMP task implementation can achieve very promising speedups when compared to other established models like OpenMP nested, task queues and CLIK.

Packet processing and parallelization: Packet processing functions have to be done in real time. If packet processing times exceed inter-arrival times, system instability will result (e.g., due to input buffer overflows). Consequently, packet processing must process packets at network line rates. When packet processing function is implemented in multicore processor based system, the packet processing rate is dependant on the number of threads and cores used for processing and the effective utilization of the hardware resources by the application programs. Packet processing workload is characterized by a large number of simple tasks and massive amounts of input/output operations. Typical applications include forwarding of packets, packet classification, packet scheduling, packet statistics and monitoring and security application (Weng and Wolf, 2009). Though none of this processing is particularly complex, the Gigabit data rates that need to be supported by network systems generate significant performance demands. The overall packet processing tasks are split into three different tasks, namely, receiving, processing and transmitting. Time critical functions take place in the processing task and the nature and extent of parallelism for the processing task and the processor architecture determines the system performance (Ettikan and Abdullah, 2003).

The necessary performance of network system is achieved through exploiting the inherent parallelism in network processing. Packets that belong to different network connections can be processed independently as a TCP/IP network makes no guarantees on packet order. Packets belonging to the same connection usually should be processed in-order for performance reasons, but this is not mandatory. As a result, almost arbitrary levels of parallelism can be achieved by replicating packet processing functionality on multiple cores and handling packets in parallel. Limits on this parallelism are imposed by the number of hardware threads and the number of cores on a single chip.

The processing demands on the packet processing system are affected by two factors: first, by computational characteristics of all tasks in the system; and second, by network traffic that exercises the processing system. In order to derive an optimal allocation of tasks to processing resources at runtime, both factors need to be quantified and considered in the mapping process (Wu and Wolf, 2008a).

Weng and Wolf (2009) presented the analytic performance model that could be applied for understanding tradeoffs in the Network Processor design space to determine suitable network processor topologies and multithreading configurations.

Wu and Wolf (2008b) have proposed the task duplication process depending on the number of tasks in the workload and the number of available processing resources. Each task is mapped to the packet processing resource based on the task locality and interconnect usage.

Lee (2010) used a simulator and showed the performance gains of a scientific algorithm that was designed to take the benefits of multithreaded multicore architecture for real scientific application problems.

In this study, we present the performance analysis of packet processing by dynamically assigning the CPUs of Sun Microsystems UltraSPARC T1 processor. OpenMP tasking feature is used for parallelizing the code for execution on the hardware threads referred as CPUs. We also propose a dynamic CPU allocation model for improving the execution time for the parallelized process.

MATERIALS AND METHODS

The performance evaluation was done on SunFire T1000 server having Sun Microsystems UltraSPARC T1 processor. Sun Studio12 Update 1 Integrated Development Environment (IDE) on Solaris 10 Operating System was used to develop the programs in C language and to test the programs. OpenMP tasking feature was used for implementing parallelism within the process. Libpcap Application Program Interface (API) was used for reading the packets from the physical interface or writing the packets to the physical interface. Furthermore, POSIX.1b Realtime Extension Library was used for message passing, process scheduling and timer options. System V message queues were used for queuing the packets between various stages. Encryption of payload was done using PKCS#11 Cryptographic Framework Library.

Fluke Networks OptiView Series III Integrated Network Analyzer was used as an external traffic generator for generating the packets of required size and at the required rate. We captured the packets from one physical interface, performed the encryption of the payload and transmitted the packets using another physical interface. Figure 2 shows the conceptual diagram of allocation of processor sets for packet processing functions. Five processor sets were created and initially CPUs were assigned to each of these processor sets. One processor set each was bound to each of the processes.

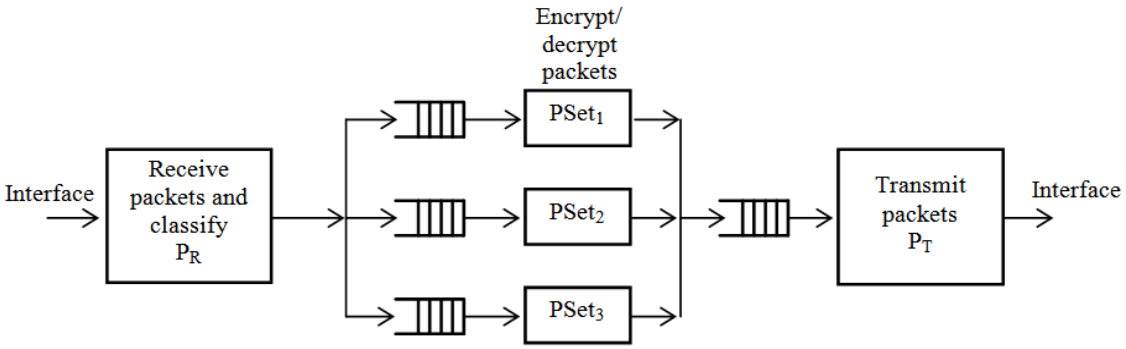

Fig. 2: Allocation of processor sets for packet processing functions

As shown in Fig. 2, one CPU was assigned to processor set $PSet_R$ for the receive_packets process that receives the packets from the physical interface and queues the packets in the appropriate queues based on the classification. Another CPU was assigned to processor set $PSet_T$ for transmit_packets process that dequeues the encrypted packets from the queue and transmits the packets to the physical interface. For dequeuing, encryption and enqueuing, three processor sets $PSet_1$ to $PSet_3$ are bound to three different independent processes so that any encryption mechanism and key value could be used for each of these processes. The encryption/decryption process is on the packets queued in each of the queues based on the classification.

For parallelization and execution by multiple CPUs assigned to a processor set, an explicit task is specified using the OpenMP task directive. Allocation/deallocation of the CPU belonging to different cores with reference to the processes running on them were done to study the performance of the parallelized process in terms of the execution time. Packet sizes for each of the combinations were also varied for further analysis. The code for dynamic assignment of CPUs is executed in each of the parallel processes. CPU_allocation_status word is a semaphore accessed and updated by all the parallel processes during dynamic reallocation. PSet_assign function is used for the assignment/de-assignment of the CPU to/from a particular processor set. Fork function is used to create the required independent processes and the processor sets are bound to each of these processes.

RESULTS AND DISCUSSION

Analysis of performance based on the static allocation of CPUs to processor sets: The total execution time taken for a particular processor set for the parallelized process of dequeuing, DES encryption and enqueuing for 576 packets with varying number of CPUs and for three different packet lengths is shown in Fig. 3. For each of the processor sets $PSet_1$ to $PSet_3$, static assignment of CPUs was done uniformly among two cores each for the parallelized process. Packet sizes of 128 bytes, 512 bytes and 1280 bytes are considered for varying the load for the processors that are executing the program in the parallel region. As shown in the Fig. 3, the execution time for processing of larger size packets takes more time as encryption is done on the payload that is larger. As the number of CPUs for a processor set increase, for 576 packets of a given size, the execution time taken for a processor set decreases for the parallelized operation. These values are taken as reference for comparison of performance for dynamic allocation.

Analysis of performance based on the dynamic allocation of CPUs to processor sets: Initially, for experimentation, static assignment of 4 CPUs to two cores was done uniformly. Based on the queue size, one CPU was deallocated from the processor set $PSet_1$. Tests were performed based on the dynamic deallocation of the CPU based on the queue length. Execution time taken for allocation/deallocation was computed to be 1.9 milliseconds that is the overhead for the allocation/deallocation process. Figure 4 shows the execution time for the total parallelized process with one CPU removed from the processor set at different values of the queue length. The variation in the execution time is linear with respect to the length of the packets at which the deallocation of the CPU is done.

When the queue size increases beyond the capacity of the queue, the subsequent packets would be dropped. Alternatively, as the queue is getting full, the source has to be informed to reduce the rate of packet transmission. This would delay the transmission rate.

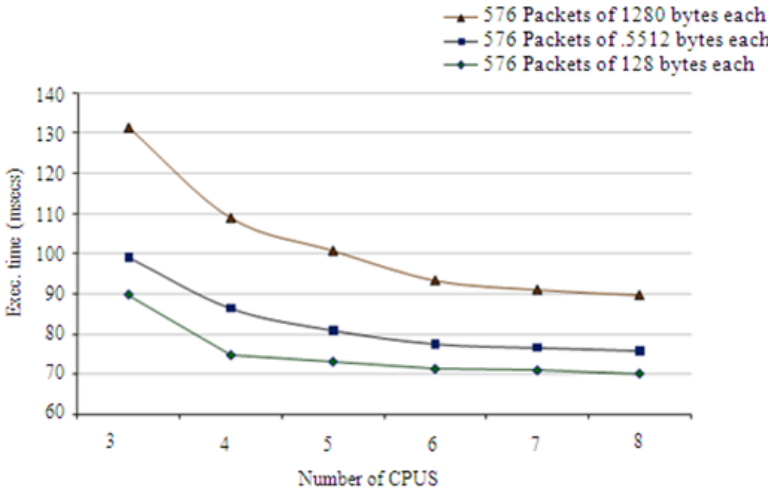

Fig. 3: Execution time based on the static allocation of CPUs on cores

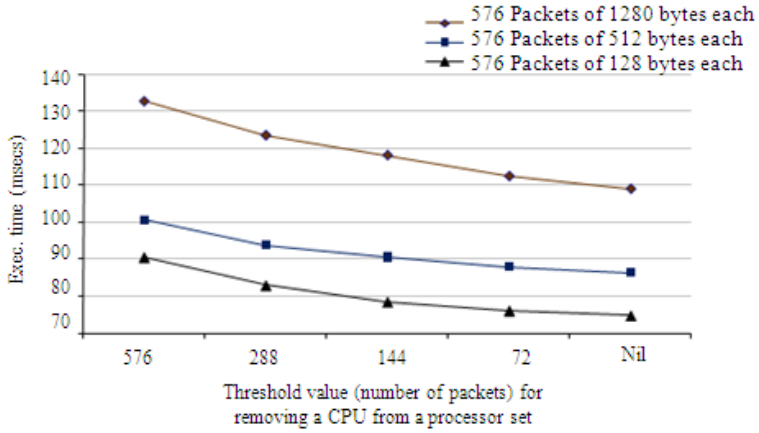

Fig. 4: Total execution time based on the deallocation of CPU at different queue lengths

To increase the rate of packet processing, a CPU could be added to the processor set that is bound to the process. In our experimentation, CPU is added dynamically to processor set $PSet_1$ that already has four CPUs statically assigned. These four CPUs belong to 2 different cores. Five possible options were experimented in terms of which a CPU is identified for the dynamic assignment. The execution time for the parallelized process for all the five options is shown in Fig. 5.

It is observed that when the CPU of the free core is allocated, the execution time is less as compared to the other options. Table 1 shows the relative improvement of the execution time of the various options of CPU identified for dynamic allocation to the processor set.

CPU allocation model: Based on the study and experimental results, it is analyzed that the overall execution time for the parallelized process varies depending on the CPU allocation to the free cores or on the cores having CPUs already assigned to the same or the different processes. Deallocation of the CPU from the processor set is done for a particular process when the number of packets in the corresponding queue is less than the prescribed value. This releases the CPUs that could be utilized by the processor sets for the processes that need more CPUs for faster execution of the packets. Accordingly, deallocation and allocation model is proposed for dynamically adding or removing the CPU form a particular process as mentioned in Fig. 6. This allocation model is focused on improving the throughput for packet processing.

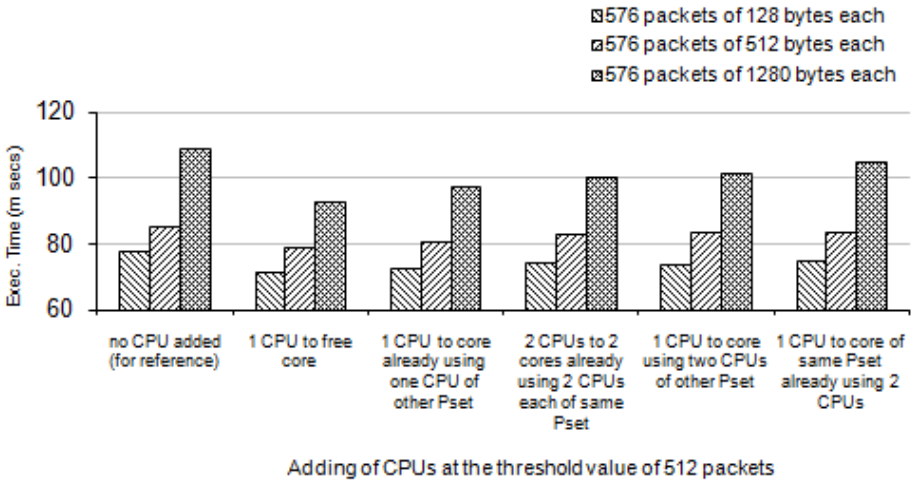

Fig. 5: Total execution time based on the dynamic allocation of CPU

```
Deallocation
Read CPU_allocation_status word and the current PSet information
Ensure that there are atleast two CPUs assigned to the PSet
Identify core that is having less CPUs assigned to the present PSet
Deallocate CPU from the core identified.
Update CPU_allocation_status word
Allocation
Read CPU_allocation_status word and the current PSet information
If CPUs assigned to the PSet are less than the maximum limit, proceed. Else, exit.
If any core is free, assign CPU of a free core to the PSet
else
If any CPU in the cores assigned to the present PSet is free,
      assign CPU to the core having less number of assigned CPUs
else if
If any CPU in the cores associated with the PSet of the lower priority process is free,
      assign the CPU to the core having less number of assigned CPUs
Update CPU_allocation_status word
```

Fig. 6: Pseudo code for dynamic allocation/deallocation of the CPU to/from the processor set

When we deallocate, as there would be less number of CPUs for the parallelized process within the processor set, the execution time for processing the remaining packets will increase but as the number of packets in the queue would be less, this marginal delay would not be significant.

This allocation/deallocation model could be generalized to any multicore processor. The total number of CPUs available per core is also a parameter for the assignment. Based on the implementation of the model proposed, programming is done for dynamic allocation of an additional CPU when the queue size is 500 packets. Allocation was done dynamically wherein the CPU of the unutilized core was assigned to the process that already has 4 CPUs assigned equally on 2 cores. Figure 7 shows the total execution time for processing 600 packets of different packet lengths, both with and without dynamic allocation using a particular processor set bound to a process that dequeues, encrypts and enqueues packets using a parallelized code for each of the CPUs of the processor set. Also shown are the packet processing rates with and without dynamic allocation.

Results were also obtained by increasing the size of the queue and accordingly setting the threshold values for dynamic allocation as a value that is 100 packets less than the maximum size of the queue. Figure 8 shows the overall execution time for the packet processing in the parallelized process for a particular processor set. The execution time is less by 13.5% for 600 packets and by 16.3% for 1500 packets with dynamic allocation. The number of packets processed per second for each of the conditions is also shown. It is observed that for all the four queue sizes, the improvement factor of packets processed per second is uniform.

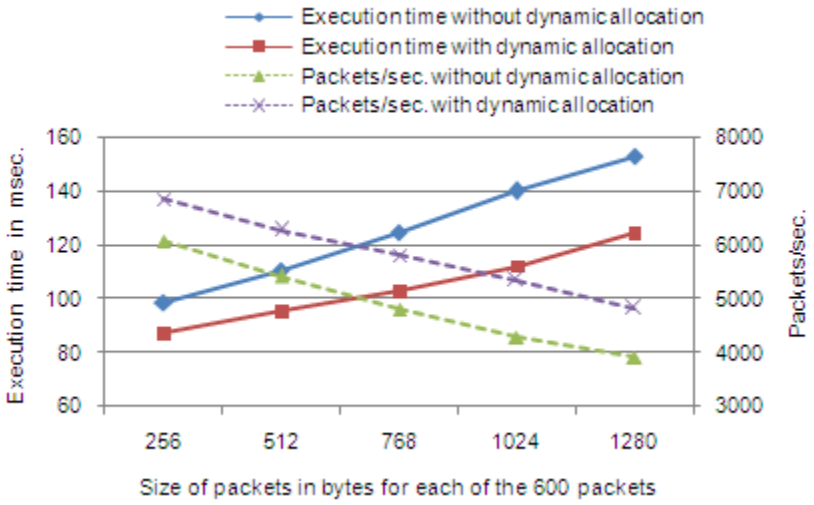

Fig. 7: Performance of the parallelized process with dynamic allocation for different packet sizes

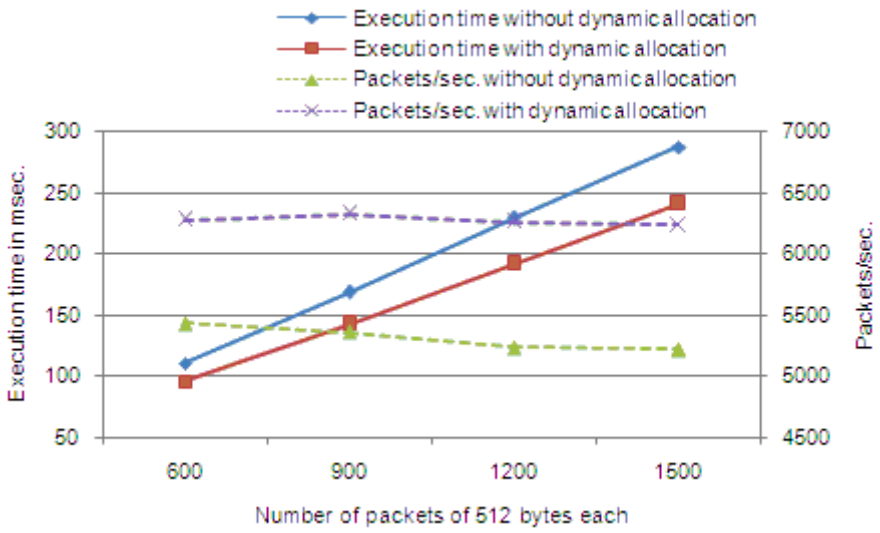

Fig. 8: Performance of the parallelized process with dynamic allocation of CPU done when the number of packets in the queue is 100 packets less than the maximum number of packets

Table 1: Relative improvement of execution time with CPUs dynamically added when the queue length is 576 packets of 512 bytes each

Details of CPU added to Processor Set	Relative improvement in execution time
1 CPU to a free core	1.080
1 CPU to a core that already has 1 CPU assigned to the other PSet	1.061
1 CPU to core that already has 2 CPUs assigned to the other PSet	1.021
1 CPU to core that already has 2 CPUs assigned to the same PSet	1.024
2 CPUs to 2 cores that already have 2 CPUs of each core assigned to the same PSet	1.028

CONCLUSION

The dynamic allocation of the CPUs must be done keeping in view the number of CPUs per core and the common resources shared by these CPUs for a particular application. The dynamic allocation of number of CPUs for a given parallelized code must be done based on the priority of the queue and the rate of packet arrival. It was observed that performance improvement factor for the execution time varies depending on the status of the core of the corresponding CPU that is added to the process dynamically. Similarly, deallocation is done to ensure that a core is

free that could be used for allocation of CPUs for any other priority process. Future work would be done to address the issues related to dynamically computing the threshold value for the dynamic allocation or deallocation.

ACKNOWLEDGMENT

The researchers acknowledge TIFAC-CORE in Network Engineering (established under the Mission REACH program of Department of Science and Technology, Govt. of India) for providing necessary facilities for working on this project.

REFERENCES

Ayguade, E., N. Copty, A. Duran, J Hoeflinger and Y. Lin *et al*., 2009. The design of OpenMP tasks. IEEE Trans. Parallel Distribut. Syst., 20: 404-418. DOI: 10.1109/TPDS.2008.105

Chapman, B., L. Huang, E. Biscondi, E. Stotzer and A. Shrivastava *et al*., 2009. Implementing OpenMP on a high performance embedded multicore MPSoC. Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, May 23-29, IEEE Computer Society, Washington, DC, USA., pp: 1-8. DOI: 10.1109/IPDPS.2009.5161107

Ettikan, K. and R. Abdullah, 2003. Survey of Network Processors (NP). Malaysian J. Comput. Sci., 16: 21-37. http://mjcs.fsktm.um.edu.my/document .aspx? FileName=266.pdf

Lee, I., 2010. Analyzing performance and power of multicore architecture using multithreaded iterative solver. J. Comput. Sci., 6: 406-412. DOI: 10.3844/jcssp.2010.406.412

Lee, W.F. and A.Y.M. Shakaff, 2008. Implementing a large data bus VLIW microprocessor. Am. J. Applied Sci., 5: 1528-1534. DOI: 10.3844/ajassp.2008.1528.1534

Leon, A.S., B. Langley and J.L. Shin, 2006. The UltraSPARC T1 processor: CMT reliability. Proceedings of the IEEE Custom Integrated Circuits Conference, Sept. 10-13, IEEE Press, USA., pp: 555-562. DOI: 10.1109/CICC.2006.320989

Sun Microsystems, 2006. Open SPARC T1 Microarchitecture Specification. Sun Microsystems, Inc, USA. http://users.ece.utexas.edu/~mcdermot/vlsi-2/OpenSPARCT1_Micro_Arch.pdf

Sun Microsystems, 2009. Sun Studio 12 Update 1: OpenMP API User's Guide. Sun Microsystems, Inc, USA. http://download.oracle.com/docs/cd/E19205-01/820-7883/820-7883.pdf

Sodan, A.C., J.C. Machina, A. Deshmeh, K. Macnaughton and B. Esbaugh, 2010. Parallelism via multithreaded and multicore CPUs. Computer, 43: 24-32. DOI: 10.1109/MC.2010.75

Weng, N. and T. Wolf, 2009. Analytic modeling of network processors for parallel workload mapping. ACM Trans. Embedded Comput. Syst., 8: 29. DOI: 10.1145/1509288.1509290

Wu, Q. and T. Wolf, 2008a. On runtime management in multi-core packet processing systems. Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Nov. 06-07, ACM, New York, USA., pp: 69-78. DOI: 10.1145/1477942.1477953

Wu, Q. and T. Wolf, 2008b. Dynamic workload profiling and task allocation in packet processing systems. Proceedings of the International Conference on High Performance Switching and Routing, May 15-17, IEEE Xplore Press, Shanghai, pp: 123-130. DOI: 10.1109/HSPR.2008.4734432