Adaptive Performance Monitoring for Embedded Multicore Systems

Chun-Yi Shih[*], Ming-Chih Li[*], Chao-Sheng Lin[*], Pao-Ann Hsiung[*+], Chih-Hung Chang[@],
William C. Chu[†], Nien-Lin Hsueh[‡], Chihhsiong Shih[†], Chao-Tung Yang[†], and Chorng-Shiuh Koong[$]
[*]National Chung Cheng University, [@]Hsiuping Institute of Technology, [†]Tunghai University,
[‡]Fengchia University, [$]National Taichung University, Taiwan. E-mail: [+]pahsiung@cs.ccu.edu.tw

Abstract — **With the advent of multicore processors, the performance of software has been elevated to new unforeseen heights via parallelization. However, this has not been achieved without new problems cropping up due to parallelization. One serious issue is the performance bottleneck due to cache misses or resource starvation, which is hard to detect in application software especially when the software has dynamically changing behavior. Performance monitors are usually employed for such purposes. Nevertheless, monitors have introduced their own computation and communication overheads, especially in embedded multicore systems. In this work, we try to estimate the effects of monitor overheads on different types of applications, such as CPU-bound and IO-bound tasks. Further, we give suggestions on the number and type of monitors to use for such embedded multicore applications. Besides trying to reduce monitor overheads, we also aim for the accuracy and the immediacy of the monitored information. Through a real-world example, namely digital video recording system, we demonstrate how different monitoring periods affect the tradeoff between accuracy and immediacy of the monitored information.**

Keywords: embedded multicore system, monitor overhead, IO/CPU-bound task, monitor accuracy, monitor immediacy

I. INTRODUCTION

Multicore processors have invaded not only the desktop computing infrastructure, but have also crept into the embedded computing paradigm. New mobile phones are now equipped with a mobile chip called Tegra 2 from Nvidia, consisting of a dual-core ARM Cortex 9 CPU and an 8-core GeForce GPU. With this progress, embedded systems can now run applications with high performance due to inherent parallelism in the computing infrastructure. However, true parallelization has also made multi-threaded embedded software more difficult to design and verify. Some well-known issues include resource sharing problems such cache conflicts and I/O device usage, inter-core communication, non-uniform memory access, unbalanced workload, data sharing, high power consumption, and performance bottlenecks. A variety of solutions exists for solving these problems. For detecting performance bottleneck at runtime, monitors are designed into the embedded software. This work focuses on how to design performance monitors for embedded multicore systems.

Monitors can be designed either by instrumenting an application software code with specific monitoring code or as an independent thread. Instrumentation is widely employed for testing and monitoring at design time, without embedding the monitoring code into final production applications. For embedded runtime monitoring, separate monitors are more preferable because they are more amenable to future management due to a clear distinction between application code and monitoring code. However, runtime monitors do not come for free. They create issues of their own. For example, monitors directly impact the computational performance of their monitoring target applications and introduce communication overheads across cores that affect the overall system performance. The latter issue on communication has been addressed in some related work. However, the former computational overhead issue has not been addressed in-depth. This work will look into this issue from several perspectives as described in the following.

We propose an *adaptive performance monitoring* (APM) scheme that tries to reduce the impact of the monitor on the application, without sacrificing accuracy or immediacy of the monitored information. We will define these terms later in Section III. More specifically, in APM, we will be discussing monitor design in terms of three different viewpoints. First, from the application viewpoint, we will be distinguishing between CPU-bound and IO-bound tasks and how monitors affect different types of tasks. Second, from the system viewpoint, we will be comparing the effects of the monitors for systems with different computational power. Lastly, from the monitor design viewpoint, we will be discussing the type of monitors to be used, the number of monitors to be implemented, and the effects of the monitoring time period on the computing overhead, the information accuracy, and the information immediacy.

As a motivational example, consider two different types of monitor deployments as depicted in Figure 1, including one-to-all and one-to-one. In the *one-to-all* deployment, there is only one monitor thread in the whole system, which monitors all application threads. Due to its low requirement on core utilization, its impact on application threads can be minimized. However, with increasing monitor workload, the accuracy and immediacy of the monitored information will decrease. In other words, a busy monitor will be unable to collect and provide accurate information in real-time. If this information is feedback to the application for runtime adaptation, then the one-to-all monitor will degrade the performance of the adaptive applications. Another issue is the possibly large communication overhead incurred by the single monitor due to the continuous exchange of monitoring information between the target threads and the single monitor thread. At the other extreme is the *one-to-one* deployment, where each application thread has a devoted monitor thread. In this scheme, monitor threads can collect and provide information in real-time, thus effectively ensuring accuracy and immediacy of the monitored information. However, a large number of monitor threads may incur an unacceptable amount of computational overhead. Since there are pros and cons with both monitor deployments, we will try to come up with an adaptive solution that suggests when to use less number of monitors and when to use a larger number of monitors.

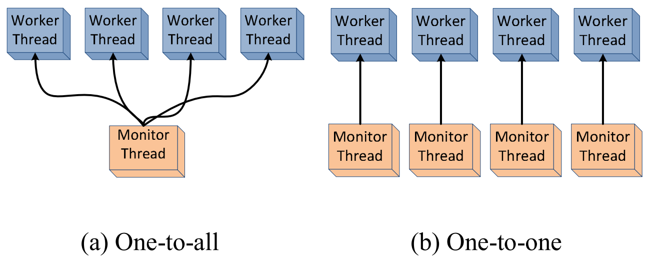


Fig. 1. Different monitor deployments

In the rest of this article, related work will be described in Section II, the main concepts and design terminologies for APM will be defined in Section III. Experiment results will be given in Section IV. Conclusions and future work will be given in Section V.

II. RELATED WORK

System monitors are being used in various application domains such as networking for application security and reliability and multimedia for application quality-of-service. According to the different objectives, the design of system monitors can be categorized into two classes, namely software-only designs and software-hardware designs.

A well-known software-only monitor design is code instrumentation through compiler techniques such as in Purify [1] and TaintCheck [2] tools. However, the instrumentation of code for program monitoring incurs high overhead and interference on the target program. In [3], the authors proposed a software monitor which controls and polices end-to-end admission control schemes, but this technique is only suitable for network applications routers or firewalls.

Hardware support can reduce the problems which are induced by software-only designs. In [4], the authors proposed a run-time monitor for secure program execution and a dedicated hardware monitor is implemented to help the system monitoring. For software debugging and memory access monitoring, the works in [5, 6, 7] require hardware support to help or optimize the monitoring tasks. A host-target combination is proposed for dynamic performance monitoring in embedded systems [8]; however, their target is not the multicore programs. In contrast, we design our performance monitor based on the multi-core processor architecture.

New monitor designs for multi-core architectures have been proposed recently. A dispatch-based approach called Log-Based Architecture [9, 10] and a distill-based approach [11, 12, 13] have been proposed. The main idea is to segregate the monitor and the application into different processes that run on different cores. The main issue addressed in these work is the communication overhead between the monitor and the target threads that are on different cores. The authors address the communication issue because their prime objective is security and reliability of the target applications. However, here in this work, our prime objective is performance, thus instead of addressing the communication problems, our study focuses on addressing the computation overhead of the proposed monitor. We try to minimize the computation overhead of the monitoring system, without sacrificing the accuracy and immediacy of the monitors.

III. ADAPTIVE PERFORMANCE MONITOR DESIGN

A monitor is a defined as an independent program that collects and provides information on other application programs running in the system. A monitor that focuses on functional data is called a *functional* monitor. For example, the memory monitors such as Purify [1] and TaintCheck [2] and the protocol monitors [3] are all functional monitors because they need to monitor the functional data from the application targets. A monitor that focuses on non-functional data is called a *non-functional* monitor. For example, monitoring the quality-of-service (QoS) of an application such as its performance, throughput, or latency is a type of non-functional monitor. Another example is monitoring the power consumption or the core utilization of an application. The basic differences between functional and non-functional monitors are as follows.

- A functional monitor is more like an assertion verifier which *constantly* reads and checks functional data for correctness or security. However, a non-functional monitor is like an estimator of the system performance based on resource usages by the application.
- A functional monitor is very strict in its correctness checking, for example, no memory leaks, no security violations, etc. A non-functional monitor needs to be very accurate in is QoS estimation, for example, the real-time performance of a video encoder/decoder in frames per second (fps), the throughput of a network streaming in bytes per second (bps), etc.

Most state-of-the-art monitors [1-13] are functional ones. In this work, we focus on the design of non-functional monitors, which have received little attention but which are very important for embedded multicore systems. The goal in the design of non-functional monitors is to reduce the computational overhead of the monitors, while maintaining the accuracy and immediacy of the monitored information.

A monitor is defined formally as $M = \langle id, A, p, q, r\rangle$, where id is a unique integer index of the monitor, A is a set of application threads that are monitored by M, p is the time period of the monitor in microseconds, q is the desired accuracy of the monitor, and r is the desired immediacy of the monitor. The accuracy of a monitor is defined as the percentage of monitoring error allowed before the monitored information is considered invalid. For example, a monitor for a video encoder with a 30 fps capture rate should have a maximum of 30 fps encoding rate; however, monitoring inaccuracy could either increase or decrease the encoding rate at runtime. If a 3.33% monitoring error is allowed, then the maximum encoding rate fluctuation between 29 fps and 31 fps is considered valid. Otherwise, the encoding rate is to be

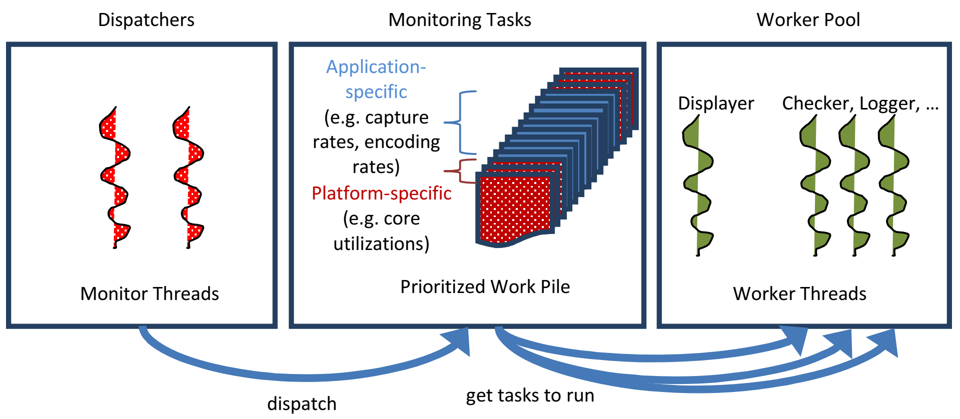


Fig. 2. Adaptive Performance Monitoring Architecture

considered invalid and ignored. The immediacy value for a monitor is defined as the latency allowed between the change of a data and the detection of the data change. For example, a 5 µs immediacy value for video encoding rate means any change in video encoding rate should be detected in that time interval.

The target problem in this work is defined as follows. Given an application consisting of a set of tasks and executing in an embedded multicore system, we need to design a set of non-functional performance monitors $\{M_i \mid M_i = \langle i, A_i, p_i, q_i, r_i\rangle\}$ such that the total overhead of monitoring is minimized while all accuracy requirements q_i and immediacy requirements r_i are satisfied. The main parameter to be determined for each monitor is its time period p_i, which will affect the satisfaction of both q_i and r_i. The total overhead is affected by all the periods p_i and also by the allocation of the monitors and the application tasks to the different cores in the processor. Here, for simplicity, the *overhead* of a monitor is defined as the percentage of increase in the execution time of a task due to monitoring. In the future, we will consider other formulations of monitor overhead such as the effect on throughput, latency, and power consumption due to monitoring.

To solve the above problem, we devised a novel architecture for performance monitoring as illustrated in Fig. 2, which consists of three parts, namely the dispatchers, the monitoring tasks, and the worker pool. The motivation behind adopting such an architecture is as follows. The binding or association between a monitor and its set of monitored tasks could be either synchronous or asynchronous. A synchronous association means the computations and the communications required for monitoring are all performed synchronously by the monitor thread itself. An asynchronous association means the computations are delegated to other worker threads, while the communications are performed by the monitor thread. The proposed performance monitoring architecture implements an asynchronous association. The dispatchers are monitor threads that are responsible for communication with the application tasks. The monitoring tasks are implemented as a prioritized work pile, which consists of the tasks that are delegated by the dispatchers. There are basically two types of monitoring tasks including application-specific tasks such as the video encoding rate and the platform-specific tasks such as the core utilization. The worker pool consists of a pool of threads that play different roles such as the displaying, logging, and checking of the monitored information. There are several parameters to be determined for this architecture, including the number of monitor threads, the size of the prioritized work pile, the number of priority levels, and the number of worker threads.

Using such architecture, in order to solve the target problem, we will answer the following questions in the next section.

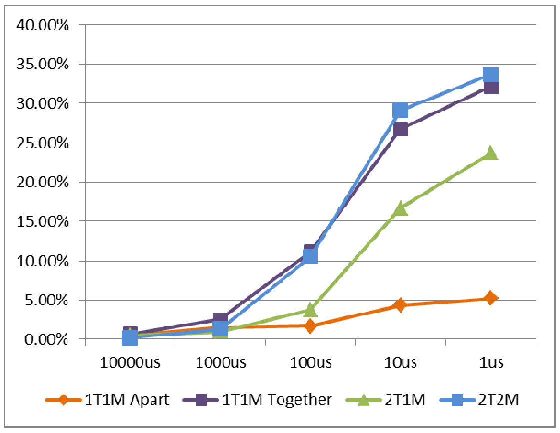
- How must the monitor threads be allocated to the processor cores so that we have low monitoring overheads?
- How many monitor threads must be designed so that monitoring overhead is reduced?
- How to determine the monitor periods such that overhead is reduced while accuracy and immediacy requirements are all satisfied?

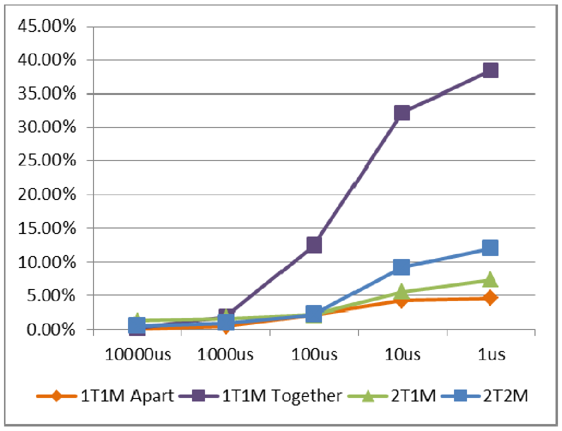
IV. Experimental Results

The proposed adaptive performance monitor architecture was implemented on two different platforms: (a) Intel Core2 Duo SU7300, with 4GB DDR3 SDRAM, Linux 2.6.35-25, and (b) ARM 11 MPCore, embedded Linux 2.6.35, 512MB SDRAM, and 128MB NOR Flash. We used the GNU gcc version 4.4.5 to compile our experiment programs without using any compiler optimizations.

A. Monitor Thread Allocation to Cores

To answer the first two questions posed in the previous section on the number of monitor threads and their core allocations, we created two types of application tasks, namely IO-bound task and CPU-bound task. The IO-bound tasks performed a large amount of file operations such as reading and writing data. Each task starts by reading a large number of integers from a file. The integers are incremented and then the results are written to another file. The CPU-bound task mainly performed the computation-intensive job of multiplication of two floating point matrices. Each task maintains its own counter on the number of operations performed. Here, an operation is defined as the reading and writing of one number in the IO-bound task, and as the completion of one matrix multiplication in the CPU-bound task. The monitors must check the counters of the IO-bound and CPU-bound tasks periodically. Semaphores are used to protect the counters.





(a) Monitored IO-bound task(s) (b) Monitored CPU-bound task(s)

Fig. 3. Comparison of overheads for four different monitor configurations

Initially, to profile the execution time of the tasks, that is, the original workload without any monitor, we ran the IO-bound tasks and the CPU-bound tasks individually. Then, different monitors were added to the tasks and the monitored tasks were executed once more. For the one-to-all monitor configuration, a round-robin scheme was employed by the single monitor when checking the task counters. Compared to the original execution time of the tasks, the resulting execution times of the monitored tasks were always larger. This difference in execution times was taken as the monitor overhead, which varied with different monitor configurations.

To answer the first two questions on number of monitors and their core allocations, we experimented with four different scenarios as follows, where a worker thread executes either an IO-bound task or a CPU-bound task:

- 1T1M Apart: One worker thread and one monitor thread on two different cores
- 1T1M Together: One worker thread and one monitor thread on the same core
- 2T1M: Two worker threads and one monitor thread, the two worker threads are on different cores, while the monitor thread is running with one of the worker threads on the same core.
- 2T2M: Two pairs of one worker thread and one monitor thread, each pair is assigned to one core.

We experimented with five different monitoring periods, including 10000us, 1000us, 100us, 10us, and 1us.

Figure 2a shows the computational overheads of four different monitor configurations for the IO-bound tasks. Figure 2b shows the computational overheads for the CPU-bound tasks. From the experimental results, we can make the following useful observations.

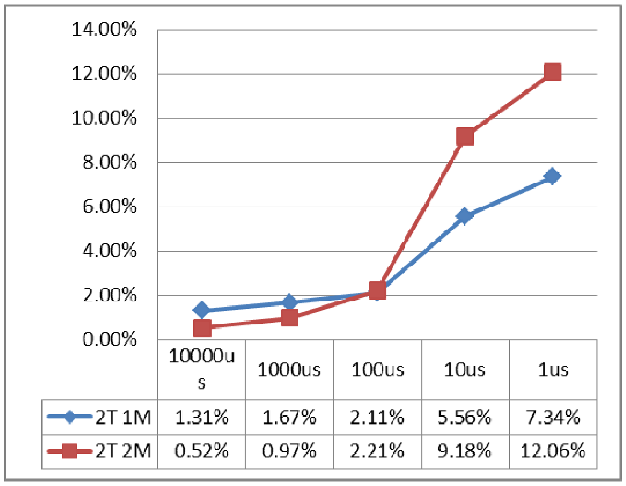
First, the monitoring overhead increases with decreasing periods. This is because smaller periods mean a higher frequency of monitoring and thus the amount of interruptions to the worker threads increases.

Second, we observe that irrespective of IO-bound or CPU-bound tasks, the monitoring overheads are almost negligible (less than 5%), if the monitor thread and the worker thread are running on two different cores. Therefore, if there is a free core, it is suggested to separate the monitor and the worker threads onto different cores.

Third, if the monitor thread is running with a worker thread on the same core, it affects the worker thread significantly. The monitoring overhead increases drastically with a decrease in monitoring period. For instance, with a monitoring period of 1000us, the monitoring overhead is less than 5%. However, in the worst situation of monitoring period of 1us, the monitoring overhead is as high as 35% for IO-bound tasks and 40% for CPU-bound tasks. Note that although higher monitoring frequency leads to greater information accuracy, the monitoring overhead may affect the performance of worker threads seriously.

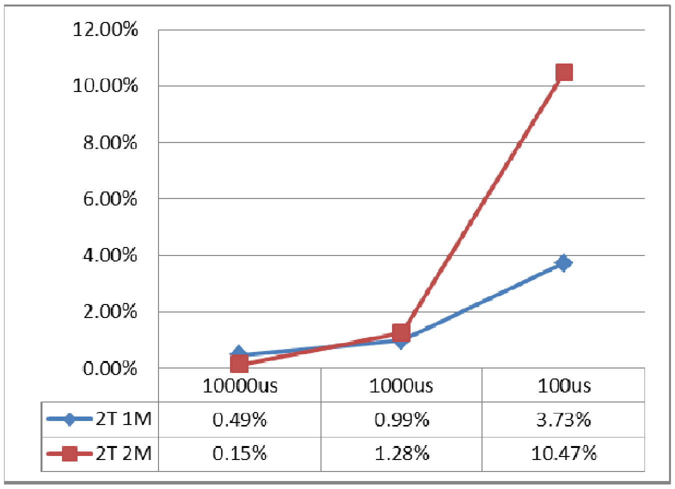
Finally, let us compare the effects of monitoring on different types of tasks, i.e., IO-bound and CPU-bound tasks. We can observe that the different numbers of monitors have a more pronounced effect on the IO-bound tasks. For example, in both 2T1M and 2T2M cases, the monitoring overheads for IO-bound tasks reaches as high as 25% in the 2T1M case and 35% in the 2T2M case. However, for the CPU-bound tasks, the overheads are only 7% in the 2T1M case and 13% in the 2T2M case. The reason for such a significant difference is that IO-bound tasks are inherently not computation-intensive and thus even a small number of interruptions from monitors will incur a larger percentage of overhead, compared to the computation-intensive CPU-bound tasks.

The above analysis has provided answers to the first question on how to allocate cores to monitor threads. Let us further use the 2T1M and 2T2M cases to find answers to the second question on how many monitors to allocate. Intuitively, one would think that a smaller number of monitors would reduce the monitoring overhead. However, for large monitoring periods, a larger number of devoted monitors instead exhibit lesser overhead due to reduced communication traffic between the monitor and the worker threads.



	10000us	1000us	100us	10us	1us
2T 1M	1.31%	1.67%	2.11%	5.56%	7.34%
2T 2M	0.52%	0.97%	2.21%	9.18%	12.06%

(a) Monitored CPU-bound tasks



	10000us	1000us	100us
2T 1M	0.49%	0.99%	3.73%
2T 2M	0.15%	1.28%	10.47%

(b) Monitored IO-bound tasks

Fig. 4. Decision on the number of monitors for two tasks

B. Number of Monitor Threads

Let us delve specifically into the cases of two worker threads, including 2T1M and 2T2M. From Figures 4a and 4b, we can observe that the two devoted monitors (2T2M) incur a lower overhead than the single monitor (2T1M) when the monitoring period is large. In the case of CPU-bound tasks, as long as the monitoring period is larger than 100us, the two devoted monitors incur smaller overheads. For example, for 1000us period, 2T1M incurs 1.67% overhead, while 2T2M incurs 0.97% overhead. In the case of IO-bound tasks, as long as the monitoring period is larger than 4000us, the two devoted monitors incur smaller overheads. For example, for 10000us period, 2T1M incurs 0.49% overhead, while 2T2M incurs 0.15% overhead.

The above analysis can be used to answer the second question: "How many monitor threads must be designed so that monitoring overhead is reduced?" The answer depends on the monitoring period. As concluded from Figure 4a, for the CPU-bound tasks, if there is a need to monitor them at most once per 100µs, then it is suggested to use two monitors, one per task. Otherwise, a single monitor is preferred due to smaller monitoring overhead. From Figure 4b, we can also answer the same question for IO-bound tasks, if there is a need to monitor the IO-bound tasks at most once per 4000µs, then it is suggested to use two monitors, one per task. Otherwise, a single monitor is preferred. For a different number of tasks, different number of monitors might be required. A more thorough analysis is desired, which will be our future focus. One more issue to be discussed here is the reason for a large difference between when to switch between single and multiple monitors for the CPU-bound (100µs) and IO-bound tasks (4000µs). The intuitive reason is that devoted monitors incur computation overheads. Since IO-bound tasks perform little computation and thus even small amounts of interruptions from devoted monitors, for example once per 1000µs, results in overheads comparably larger than a single monitor. A single monitor is thus preferred. In contrast, this effect is not so pronounced at the same monitoring period of 1000µs for CPU-bound tasks, thus devoted monitors are preferred.

C. Accuracy and Immediacy of Monitored Information

In the following, we go one step further to investigate the tradeoff between the accuracy and the immediacy of monitored information in embedded multicore systems. We applied our adaptive performance monitoring to a *Digital Video Recording* (DVR) system [14], which is a real-time multimedia system for online and on-demand video streaming, with multiple cameras and multiple client connections. As illustrated in Figure 5, DVR consists of parallel tasks for supporting multiple cameras, parallel data for block-based discrete cosine transformation (DCT) of each video frame, and parallel pipeline for the video encoding sequence consisting of quantization, DCT, and Huffman encoding. DVR was used to illustrate the benefits of the VERTAF/Multi-Core (VMC) framework [14] for the development of multi-core embedded software. VMC automatically generated the parallel code for DVR based on the *Quantum Platform* (QP) [15] and the Intel's *Threading Building Block* (TBB) [16]. However, the code generated for DVR did not consist of an optimized monitor and thus the performance bottlenecks were difficult to detect. Further, we also found that it was difficult to check the accuracy of the monitored information. Thus, through the results of this work, we designed a monitor suitable for DVR. In the following, we focus on how information accuracy is affected by the monitor.

In DVR, we designed a monitor for the buffers and monitored the video frame capture rate and encoding rate. To measure the variance in overheads incurred by monitors, we experimented with different monitoring periods, including 1, 0.5, 0.1, and 0.01 seconds. The average encoding rates for different monitoring periods as provided by our monitor are compared in Figure 6. Generally, the performance of the monitored task (the parallel video encoder, here) is degraded with a decrease in the monitoring period due to higher monitoring frequency and overhead. This trend in performance degradation is observed for the monitoring periods of 1, 0.5,

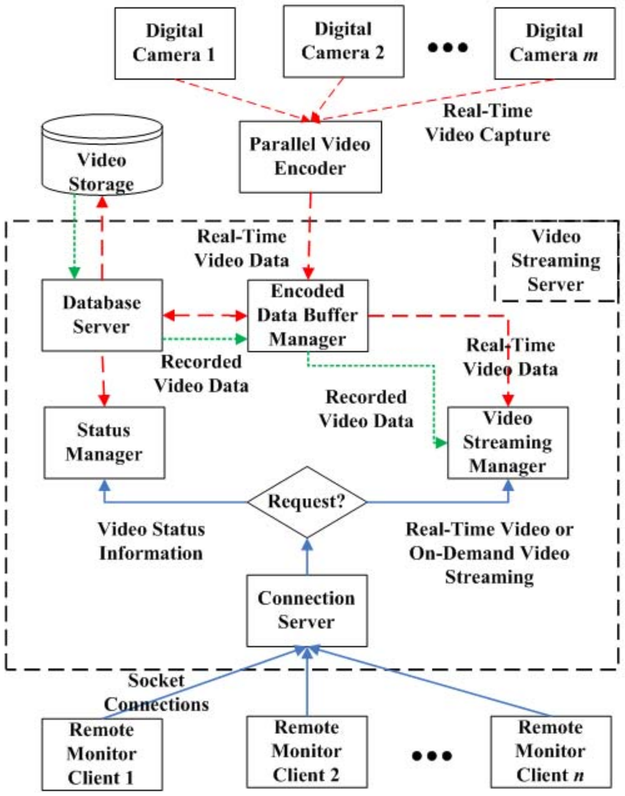
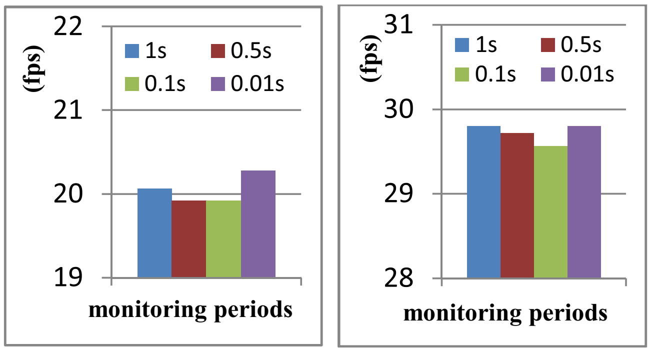


Fig. 5. Digital Video Recording System

and 0.1 second. However, both the encoding rate and the capture rate unexpectedly increased in the case of 0.01 second as the monitoring period, whereas in fact, it should be lower than in the case of 0.1 second. On investigation of these unexpected results, we found an accuracy problem in the monitor design, which is described in details in the following.

To explain the accuracy problem, we delved into the encoding rates given by the monitor over a time period. When the monitoring period is 0.01 second, the recorded encoding rates consisted of three to five consecutive 0 values for the encoding rates, and then a very large encoding rate of 125 fps. This cycle of several 0's and one large rate repeatedly occurs. As a result, the instantaneous encoding rate as calculated by the monitor is misleading and has a wide-range of errors. When the monitoring period is 0.1 second, the recorded encoding rates consisted of one or two consecutive encoding rates of 25 fps and then a comparatively larger value of 37 fps. This cycle occurs repeatedly in the monitored information. The large variance in observed encoding rates indicates that the rates are invalid. Furthermore, they also cause inaccuracy in the calculation of the average encoding rates. With larger monitoring periods, such as 1 and 0.5 second, the observed encoding rates are within a reasonable range of 28 fps to 32 fps. This indicates that larger monitoring periods are desirable for accuracy; however, in that case the immediacy of the monitored information would be affected and become poor. Consequently, a proper monitoring period is required to achieve valid observations and evaluations of target systems.



(a) Encoding Rates (b) Capturing Rates

Fig. 6. Monitor inaccuracy results in unexpected performance changes

V. Conclusions

Parallel application development for embedded multicore systems is often faced with the need to detect performance bottlenecks. Performance monitor is a viable solution; however, there are several design issues related to the monitor design for embedded multicore systems such the allocation of monitor threads to cores, the number of monitor threads to be designed, and the tradeoff between accuracy and immediacy of monitored information. In this work, we have proposed a novel adaptive performance monitoring framework that can be used to solve the above issues. Since monitor deployment onto cores may affect the performance of the monitored application tasks, we performed extensive experiments by considering not only different types of tasks (CPU-bound, IO-bound), but also different monitoring periods. We estimated the overheads caused by the different monitoring periods for the different types of tasks. We found that for larger periods, devoted monitors incurred lower overheads compared to single monitor; however, for smaller periods, single monitor was preferable due to lower overheads. This was true for both CPU-bound and IO-bound tasks; however, the overhead was more pronounced for IO-bound tasks. Further, we also clarified that there is a tradeoff between information accuracy and immediacy due to different monitoring periods having contrasting effects on accuracy and immediacy. On one hand, large monitoring periods reduce overhead, increase accuracy, but degrade immediacy. On the other hand, small monitoring periods allow high immediacy, but increases overhead and decreases accuracy. In the future, we plan to analyze the monitoring framework such that deeper issues such as how to tradeoff between adaptivity and sensitivity can be addressed.

VI. References

[1] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *the Proceedings of the Winter 1992 USENIX Conference*, 1991, pp. 125–138.

[2] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Procs. of the 12th Network and Distributed System Security Symposium* (NDSS), February 2005.

[3] I. Mas, J. Brage, and G. Karlsson, “Lightweight monitoring of edge-based admission control,” in *Procs. of the International Zurich Seminar on Communications*, July 2006, pp. 50-53.

[4] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, “Hardware-assisted run-time monitoring for secure program execution on embedded processors,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, January 2007.

[5] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, “Memtracker: Efficient and programmable support for memory access monitoring and debugging,” in *Procs. of the 13th International Symposium on High-Performance Computer Architecture*, February 2007.

[6] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, “iwatcher: Simple, general architectural support for software debugging, ” in *Procs. of the 31st Annual International Symposium on Computer Architecture*, 2004.

[7] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic, “Heapmon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection,” IBM J. Res. Dev., vol. 50, no. 2/3, pp. 261–275, 2006.

[8] Y. Guo, Z. Chen, and X. Chen, “A lightweight dynamic performance monitoring framework for embedded systems,” in *Procs. of the International Conference on Embedded Software and Systems*, May 2009, pp. 256-262.

[9] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. Mowry, and et al., “Log-based architectures for general-purpose monitoring of deployed code,” in Procs. of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ACM, 2006, pp. 63–65.

[10] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. Mowry, and et al., “Flexible hardware acceleration for instruction-grain program monitoring,” in *Procs. of the 34th Annual International Symposium on Computer Architecture*, June 2008.

[11] G. He and A. Zhai, “Improving the performance of program monitors with compiler support in multi-core environment,” in *Procs. of the IEEE International Symposium on Parallel & Distributed Processing*, May 2010.

[12] G. He, A. Zhai, and P.-C. Yew, “Ex-mons: An architectural framework for dynamic program monitoring on multicore processors,” in *Procs. of the 12th Workshop on Interaction between Compilers and Computer Architectures*, 2008.

[13] G. He and A. Zhai, “Efficient Dynamic Program Monitoring on Multi-Core Systems,” *Journal of Systems Architecture*, Vol. 57, pp. 121-133, 2011.

[14] C.-S. Lin, C.-H. Lu, Y.-R. Chen, S.-W. Lin, and P.-A. Hsiung, “VMC: A Model-Driven Framework for Multi-Core Embedded Software Development,” *Journal of Computer Science and Technology*, 2011.

[15] Quantum Leaps, What is QP™? Quantum Leaps®, LLC. Retrieved May 10, 2010, from http://www.state-machine.com/products/.

[16] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly Media, Inc., 2007.