A Unified, Low-overhead Framework to Support Continuous Profiling and Optimization

Ming Zhang† Xubin He‡ Qing Yang†

†Electrical and Computer Engineering
University of Rhode Island
Kingston, RI 02881
{mingz, qyang}@ele.uri.edu

‡Electrical and Computer Engineering
Tennessee Technological University
Cookeville, TN 38505
hexb@tntech.edu

Abstract

We propose a unified, low-overhead framework (ULF) to support continuous system profiling and optimization based on a specifically designed embedded board. Instead of building a new profiling tool from scratch, ULF provides a unified interface to integrate various existing profiling tools and optimizers, and helps to easily build future tools. ULF uses an embedded processor to offload tasks of post-processing profiling data, which reduces system overhead caused by profiling tools and makes ULF especially suitable for continuous profiling on production systems. By processing the profiling data in parallel and providing feedback promptly, ULF supports on-line optimization. Our case study on I/O profiling demonstrated that ULF-enhanced profiling tool dramatically reduces overhead making continuous profiling on production systems feasible.

Key words: Embedded system, continuous profiling, on-line optimization, performance evaluation

1 Introduction

Program profiling [1, 2, 3, 4, 5, 6] is an important mechanism to observe system activities and profiling-based optimization has become a key technique in program optimization [7]. Extensive research has been reported in the literature in program profiling including software techniques [3, 8, 9, 10, 11, 12, 13] and hardware techniques [4, 14, 15] to list a few. One of the important issues in profiling is profiling overheard. Some of existing studies [7, 15] attempt to minimize profiling overheads to allow runtime profiling and optimization that are very important because variations of program behaviors at end users can be substantial. Profiling and optimization overhead is mainly caused by:

Gathering raw data: Profiling tools do sampling to gather raw data using instrumentation code [3, 8, 9] or interrupts [13, 16].

Recording raw data: Profiling tools save the generated raw data to local disks or system buffer. Vtune [10] transfers profiling data to a remote system via network. Saving data to a local storage device causes contention with systems' original I/O activities while transferring via network causes skew for network activity profiling.

Processing raw data: Profiling tools usually delay processing data until enough profiling data are gathered. On-line optimizers such as Morph [13] use system idle time to analyze data.

Feedback: Applying optimized feedback solutions to host systems.

To minimize profiling overhead and support continuous profiling and optimization at runtime, we propose a unified framework for low overhead profiling (ULF). Our approach is to use a specifically designed embedded processor board to offload most of profiling and optimization functions from the host CPU. As a result, profiling and optimization operations are done in parallel to applications to be optimized making it possible to carry out runtime profiling and optimization on production systems with minimum overhead. Our ULF is a general framework with a set of easy-to-use APIs allowing any existing or newly proposed profiling and optimization techniques to make use of ULF for low overhead profiling and optimization on production systems. Functions running on the ULF board are in forms of plug-ins to be loaded by a user at run time. They do not generate overhead on host system and thus do not degrade host system performance. We have carried out a case study of applying an existing profiling tool on this ULF board with very little change on the tool to make it work on the board. Our experiment results show that ULF is highly effective and reduces overhead dramatically from 40% to 0.41%.

The remainder of this paper describes ULF in more detail. Section 2 presents the architecture and design of the framework. Section 3 gives a case study of I/O profiling using our ULF, and presents the measured results. Section 4 discusses cost and integration of ULF. Section 5 examines related work and Section 6 concludes the paper.

2 Design of ULF

ULF is a hybrid of hardware and software containing the following three main components:

ULF Board. We designed an embedded board that carries out main functionality of ULF. A ULF board contains an embedded processor that provides computing power to whole system and offloads the processing task of raw data from a host processor. In this way, we turn the post-processing to parallel processing from which on-line optimization can benefit.

Software running on a host system. This software provides APIs for other profiling tools to utilize the functionality of ULF. It runs on host systems as a library or a kernel module that exports routines for profiling tools running in kernel space.

Software running on ULF Board. The software includes an embedded operating system to drive ULF Board, a library to provide helper routines to ease the post-processing on raw data, and plug-ins to help profiling tools to implement user-defined functionalities.

2.1 Hardware design

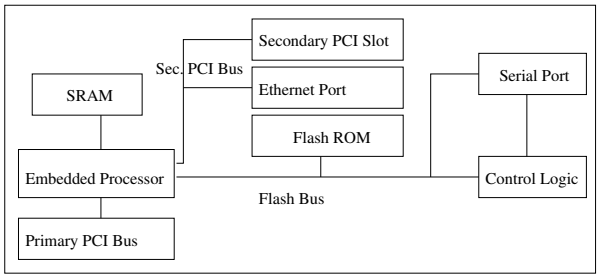

Figure 1. ULF Board block diagram

The hardware design is essential to the whole ULF. It is an embedded system board that plugs into host system's PCI slot as a normal PCI device. The detailed functional blocks are shown in Figure 1. ULF provides data transfer rate up to 528 MB per second with 64 bits 66MHz PCI bus. An embedded processor is used to process raw profiling data. The processor also supports Message Unit that provides a mechanism for transferring data between a host system and the embedded processor on ULF board. The Message Unit notifies the respective system of the arrival of new data through an interrupt. Both host systems and ULF can process the interrupts via registered handlers. Like many other embedded systems, the Message Unit in our design supports common functionalities such as Message Registers, Doorbell Registers, Circular Queues and Index Registers. A sufficient amount of RAM is included on the ULF board. The RAM is divided into two parts. One part of the memory is used privately to store code and data used by the embedded processor while another piece of the RAM is shared between the local embedded processor and the host processor. A Flash ROM on board contains the embedded operating system code and data processing routines. A ULF Board also contains an Ethernet port and a serial port that can provide connections with external systems. A secondary PCI slot is used to provide flexible expandability to this board. For example, a disk connected to ULF board through the secondary PCI can be used to save profiling data for post-processing. The system timer and other control functions are implemented by the control logic as shown in Figure 1.

2.2 The Interface to Host System

When a ULF Board is plugged into a host PCI slot, it acts as a normal PCI device and exports several registers and a region of I/O memory. Although it can be accessed via low-level PCI-specific APIs directly, we provide a set of upper-level APIs to encapsulate the low-level details of PCI devices in order to ease the use of ULF Board. Profiling tools can use these upper-level APIs to finish all tasks without knowing the low-level hardware details. These APIs fall into following categories.

Resource Management APIs. Before using ULF board, profiling tools need to initialize the board and request resources from it. These resources include I/O memory, registers, Message Units, DMA channels, and etc. After finishing using the board, profiling tools also need release these resources. There is a request and release routine corresponding to each kind of resource.

Data Transfer APIs. These APIs are used to transfer data between a host processor and ULF Board. Different read/write routines are provided to transfer data in different size units such as Byte, Word, and DWORD. We also provide "memcpy" for larger size data transfer operations.

Message APIs. The message APIs are encapsulation of the Message Unit. Through these APIs, we provide a mechanism to exchange information between a host processor and an embedded processor. Since each Message Unit is also a hardware resource, to request and free the use of Message Unit is accomplished via corresponding resource management APIs. Profiling tools can use message APIs to send user-defined messages to the embedded processor.

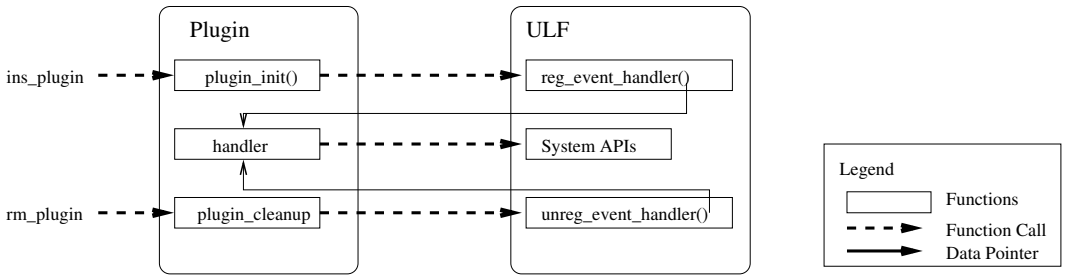

Figure 2. Interaction between Plug-ins and Boards

They may also register callback routines via message APIs, and these routines are invoked when corresponding process running on the embedded processor send messages back to them.

Other helper APIs. Besides above APIs, there are also other APIs providing helper functions, such as error handling routines and status reporting routines.

2.3 ULF Plug-ins

After raw profiling data are transferred from a host to ULF Board, we need a unified mechanism to support rapid data processing and provide effective feedback. This is achieved by plug-ins as shown in Figure 2, which is very similar to how a Linux system manages its kernel modules.

Each profiling tool either uses ULF-predefined plug-ins to finish common profiling or provides a plug-in to ULF in order to finish its specific functionality. For example, a profiling tool may save the raw profiling data to a disk for later use. Or an on-line optimizer may analyze raw profiling data, deduct instructions that guide how to provide optimization and feedback to the host system on the fly. The optimizer may even use the instructions to guide cross-compile compiler running on ULF Board to compile optimized code for host system and apply that optimized code to host directly. All these specific functionalities are determined by profiling tools and implemented as specific plug-ins. Currently, ULF provides a common set of plug-ins to save raw data on attached storage devices.

ULF provides a unified interface to plug-ins by several APIs. Each plug-in uses API "ins_plugin" to link with the system on ULF Board and register at least one event handler using API "reg_event_handler". This handler will be called when the system receives message from a host. A plug-in can transfer some data to a host and notify it by using the API "send_data" with the data address and the data length. Then the corresponding registered call back routine on the host fetches the data and carries out its specific task. After finishing all tasks, the plug-in uses "unreg_event_handler" to unregister all previously registered handlers and unloads itself by "rm_plugin".

2.4 Interactions between the profiling tools and ULF

A ULF provides a flexible unified interface to be utilized by profiling tools and it is important to clarify how ULF interacts with different profiling tools that have specific requirements and functionalities. Using a continuous on-line optimizer as an example, let us see how an optimizer interacts with ULF.

Initial stage. This stage initializes ULF on both the host and ULF Board. The optimizer locates ULF Board and allocates I/O memory resource using resource management APIs. The optimizer also registers a call back routine in the host in order to get feedback from ULF. To process raw profiling data on-line, a plug-in for the optimizer is registered on ULF Board.

Running stage. The optimizer runs on the host and keeps gathering raw profiling data. It may transfer these data to the board continuously or in a larger unit using data transfer API. After each data transfer, the optimizer uses the message API to notify ULF Board that the data is ready, which is a specific interrupt. The system on ULF Board receives this message and forwards it to the corresponding plug-in. Then the plug-in is invoked with this message and the data pointer, and processes the raw data according to the user-defined criteria. After the plug-in gathers enough raw data, finishes processing these data, and gets optimization solutions, it notifies the host system. The call back routine in the host receives this notification and applies optimization solutions to system. This finishes one optimization loop. This step keeps running until the completion of profiling and optimization.

Cleanup stage. When profiling and optimization are finished, the optimizer uses a message API to send an end signal to ULF Board. The plug-in on the board will finish its processing and send an acknowledge message to the host. Then the optimizer releases resources and exits. The plug-in also unloads from ULF.

2.5 Example Applications

With its unified interface, low overhead data collection, and sufficient computing power, our ULF can be utilized in many system level profiling and optimization environments. In addition to the case study that we carry out in the following section, there are many potential applications that can be benefited from ULF.

Low Overhead Profiling. Profiling tools gather raw profiling data on a host as usual and transfer the data to ULF Board. Then the plug-ins process and analyze the data in parallel. They can also store raw data or processed data to an optional disk or send them to remote systems via a network if network is not part of system under test. This on-line processing is especially useful when we need a real-time feedback to dynamically measure a system.

On-line Program Optimizer.Morph[13] is an example to utilize ULF. Morph can provide on-line optimization to programs, while it uses idle time of host to process profiling data and to recompile optimized code offline. By offloading all processing to ULF Board, we have an enhanced Morph that allows host to keep running while processing profiling data and recompiling optimized code on the fly. Then heavy loaded system can also benefits from this even without substantial periods of idle time available.

On-line File System Cache Optimizer. By monitoring dynamic file system access patterns and transferring profiling data to ULF Board, an optimizer can use high accurate although complex algorithms to predict future access patterns and direct the host file system to use better cache replacement and prefetching policies. By offloading the computing of detecting and deduction algorithms, such an optimizer can reduce the host's performance loss caused by these algorithms and can use complex algorithms to obtain larger improvement while the extra overhead caused by algorithms is moved to ULF.

3 A Case Study: I/O Profiling

Since the gap between I/O systems and processors keeps increasing, I/O systems frequently become bottlenecks. In order to optimize I/O systems, it is essential to carry out I/O profiling to characterize I/O system behaviors. For I/O profiling, there are many system events that need to be recorded, which leads to high volume of raw profiling data. How to store these data efficiently is an important issue to I/O profiling tools. Most existing profiling tools write raw profiling data to local storage for post-processing, which adds extra burden to already heavy-loaded storage systems and also causes skew on measured profiling data. Considering this overhead, people can only carry out I/O profiling during a short period. By offloading overheads caused by profiling tools, ULF supports continuous I/O profiling in production systems.

3.1 Methodology and Experimental Setup

We have implemented a ULF prototype based on an Intel IOP310 processor under Linux platform. We export a set of APIs to host systems by a kernel module. We modified a popular I/O profiling tool, LTT [17], to utilize our ULF. We measured and compared the results of popular benchmarks running under different LTT configurations. We modify the LTT and let LTT operate in three different ways. All these modifications only involve minor change on LTT code and we will discuss this integration cost issue later. Along with an unmodified Linux kernel, we have four different configurations as follows:

NTNR (Neither Traced Nor Recorded). This is the original Linux 2.4.16 kernel without any overhead. This configuration is the baseline for comparison purpose.

TNR (Traced and Not Recorded). This configuration uses modified Linux 2.4.16 kernel and logs system events. All these events are discarded when the event buffer is full so it does not generate any recording overheads.

TDR (Traced and Disk Recorded). This is a full-featured LTT system with modified Linux 2.4.16 kernel. It logs all system events and records events to a local disk when the event buffer is full.

TFR (Traced and ULF Recorded). This is a ULF enhanced full-featured LTT system with modified Linux 2.4.16 kernel. It logs all events and records events using ULF when the event buffer is full.

The first benchmark program we used in our test is PostMark, a popular file system benchmark developed by Network Appliance, and used by many researchers [18, 19]. It measures performance in terms of transaction rates in an ephemeral small-file environment by creating a large pool of continually changing files. PostMark generates an initial pool of random text files and carries out a specified number of transactions. Each transaction consists of a pair of smaller transactions that are chosen randomly. On completion of each run, a report is generated showing some metrics such as elapsed time, transaction rate and so on. Another benchmark we used is IoZone, which is a widely used [20, 21] file system benchmark tool. The benchmark generates and measures a variety of file operations carried out on a single large file.

We carried out experiments on a Dell PowerEdge 1400 server with hardware configuration as listed in Table 1. We ran all tests on the hardware RAID that consists of 4 Quantum SCSI disks using RAID5 configuration and recorded raw profiling data to a separate IBM SCSI disk via independent SCSI controller.

Components	Description
CPU	Pentium III 866
Memory	256MB PC133
SCSI Controller	Adaptec AIC7899
RAID Controller	LSI MegaRAID
SCSI Disks	Quantum Atlas10K2-TY184L IBM DNES-309170W

Table 1. Hardware Configurations

3.2 Measured Results

In order to measure overheads caused by different LTT configurations, our first experiment is to run PostMark with 20K initial files and different transactions ranging from 150K to 300K. All other parameters of PostMark are set as default. The read and write data traffic generated by these transactions are listed in Table 2, which are much larger than the system RAM size (256MB).

	Data Read (MB)	Data Written (MB)
150K	465.84	600.33
200K	629.36	763.14
250K	793.88	926.05
300K	956.59	1087.95

Table 2. PostMark Data Traffic

PostMark results are shown in Figure 3. Figure 3(a) shows the total elapsed time for each LTT configuration against different transactions. We treat NTNR as our baseline here since it is unmodified Linux kernel without any extra overhead. We can see that TNR, which discards the raw profiling data when the event buffer is full, shows unnoticeable difference with the base configuration. The total elapsed time of TDR that writes raw profiling data to a local disk is always much longer than the baseline configuration. Compared to TNR, the extra operation is to write raw data to a local disk and it becomes the main cause of the overhead. We also noticed that TFR does not show much extra overhead by utilizing ULF. Using the total elapsed time, we compute overheads of different LTT configurations as shown in Figure 3(b) by setting our baseline configuration as 0. We find that compared to the baseline configuration, both TNR and TFR show less than 1.2% overhead while TDR shows 32.98%, 35.43%, 38.34% and 42.86% overheads corresponding to different transactions, respectively. Since the only difference between TDR and TFR is to use a local disk or ULF to persist the raw profiling data, ULF-enhanced LTT reduces overheads brought to the host system from more than 32% to less than 1.2%. In Figure 3(c), we observed that NTNR, TNR, and TFR show similar throughput in all scenarios while TDR shows much lower throughput. We computed the performance improvement and plotted it in Figure 3(d). It is obvious that the performance gain of using ULF ranges from 31% to 42% under different number of transactions.

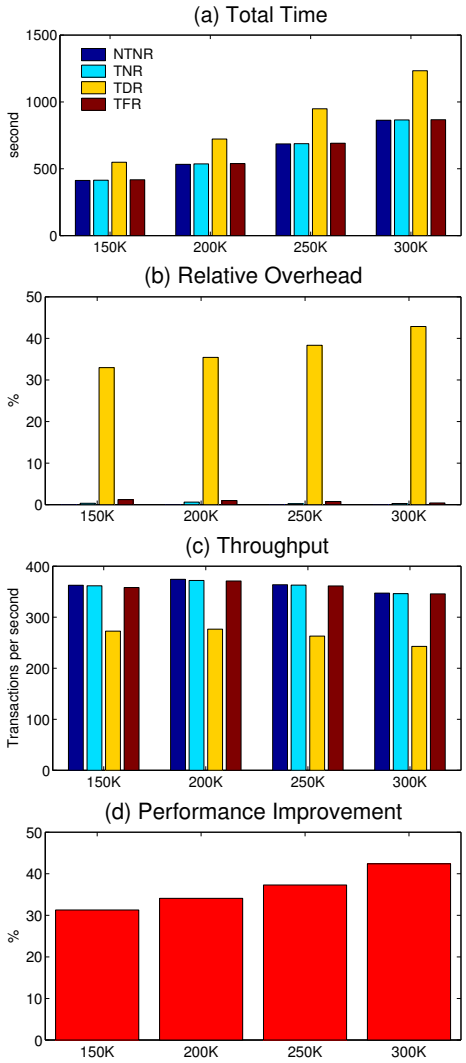

Figure 3. PostMark with different transactions under LTT

Our next experiment is to measure the influence of read/write ratio by changing the write biases of PostMark while keeping the initial files as 20K and total transactions as 200K unchanged. Higher write bias means more write operations. We computed the overheads of different LTT configurations under different PostMark write biases and piloted results in Figure 4(a). The overhead of TDR changes dramatically from 4.6% to 59.5% while that of TNR or TFR

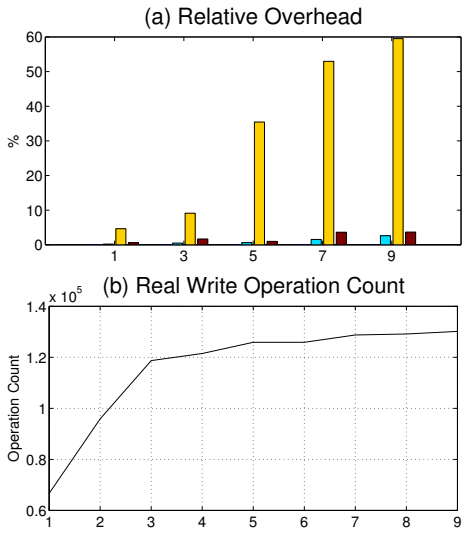

Figure 4. PostMark with different write biases

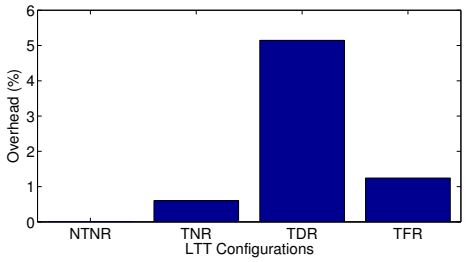

Figure 5. IoZone under different LTT configurations

almost keep unchanged with the change of write bias from 1 to 9. In order to find the reason, we modified the Linux kernel and add two counters to record the number of physical read and write operations to the RAID. These numbers are different from the count of read and write transactions since all transactions are handled by the file system layer and some read and write transactions can be satisfied by the file system cache, which does not generate physical disk operations. By using these two counters, we measured numbers of physical read and write operations of PostMark under different write biases. We find that for all different write biases, the numbers of read operations are almost same (around 720) while the numbers of write operations change dramatically as listed. When the write bias changes from 9 to 1, the write operations count change from 130,172 to 66,710. Since the main overhead caused by the TDR is the I/O subsystem contention between host system original I/O operations and profiling data write operations, we believe that with the decrease of the write bias, original write operations of host system also decrease, which relieves I/O contention. Because the overhead of TFR changes little with the load of I/O subsystem, we believe that TFR is more suitable for I/O profiling than TDR.

Our third experiment is to test different LTT configurations using IoZone. We run IoZone with different data set and record the total elapsed time for each LTT configuration. Figure 5 shows the relative overhead of different configurations on average. The results show that TFR reduces the overhead of TDR from 5% to less than 1% on average, which is around 4 times improvement. Using our read/write counter, we find that IoZone generates around 179 transactions per second as opposed to around 233 transactions per second generated by PostMark for the write bias of 5. Since IoZone generates much less requests than PostMark, the overhead under IoZone is less than that under PostMark.

In all above experiments, there is no apparent increase in number of system interrupts. Although our ULF Board uses interrupts to communicate with host system, it also reduces the number of disk controller interrupt implying that the communication between host system and ULF does not introduce new overhead to host system.

4 Cost and Usability

Compared to traditional profiling tools usually in the form of software, ULF uses additional hardware that increases cost. But we believe that this cost is worthwhile from our experience of building the prototype of ULF. The basic prototype board costs less than $200 and contains an Intel IOP310 processor, 128M memory, and a 100M Ethernet controller. With the optional IDE disk controller and a small size IDE disk, the total cost can be less than $300. The software running on board is openly available embedded Linux and other open source tools. Since our ULF is especially suitable for high-load server systems, compared to the total system cost, ULF is still a cost-effective solution.

Another important issue needs to be considered is how easily ULF can be integrated into existing or new profiling tools and optimizers. Since our ULF provides a unified interface and encapsulates low-level hardware details in API routines, it is very easy to be integrated. In our case study, we used the source code of LTT and only modified less than 20 lines of code in order to utilize ULF. A simple plug-in with less than 150 lines of code runs on ULF Board and its function is to save raw profiling data to disk. All these works are finished in one day after understanding the source code of LTT. So we believe that with the source code of profiling tool in hand, this will not be a big issue.

5 Related work

There are many profiling tools reported in the literature. Digital Continuous Profiling Infrastructure (DCPI) [12] is a profiling system that monitors system activity continuously on production systems. Morph [13] provides a framework for automatic collection and processing of profile information and profile-driven optimizations. Both tools support continuous profiling and share some of the characteristics of ULF. DCPI provides accurate instruction level information, attributing dynamic stalls to the instructions that actually incur those stalls. DCPI also uses hardware-specific performance counters, which limits its usage. The Morph system provides on-line optimization to program by making use of idle time so that it can process profiling data and recompile optimized code offline.

Multi-ICE and MultiTrace [14] are two debug tools for ARM core based system on chip (SOC) device, where Multi-ICE is an ARM's JTAG-based in-circuit emulator. MultiTrace is used to passively collect data from ARM SOC. Although Multi-ICE and MultiTrace share some features with ULF, such as real-time and embedded system support, ULF provides continuous profiling and on-line optimization while Multi-ICE and MultiTrace are mainly used for debug purpose.

There are several hardware based profiling techniques such as the programmable co-processor proposed by Zilles and Sohi [15] that assists profiling by performing local post-processing on the profiling data, thereby reducing the overhead of collecting each data sample. Conte et al [22] proposed a hardware-based profiling using traditional branch handling hardware to generate profile information in real time. Merten et al [4] presented a framework for identifying program hot spots for run-time optimization. A small-sized frequently active region of code (hot-spot) is detected by the hot spot detector and used to support runtime optimization. All these hardware based profiling techniques aimed at instruction-level CPU profiling whereas ULF provides a tool for overall system profiling by using an embedded processor board that can be plugged into a system bus such as a PCI or PCI-X.

Using an embedded board, ULF reduces the overhead of many existing profiling tools. In addition, ULF reduces requirement of system resources such as host memory, network bandwidth, and disk I/O that are already overloaded by offloading the post-processing proofing data to ULF board. Both existing and newly proposed profiling tools can make use of ULF to off load profiling and optimization functions from host system and perform runtime optimization in production systems.

6 Conclusions

In this paper we have proposed a unified low-overhead framework (ULF) that helps profiling tools to save profiling data rapidly and perform run-time parallel processing. We provide a set of APIs to allow an easy integration of profiling tools with our ULF. By offloading the post-processing of profiling data, ULF supports on-line optimization and parallel processing, and dramatically reduces the overhead caused by profiling tools. We have conducted an extensive case study for I/O profiling. The measured results show that our framework reduces the system overhead caused by the existing profiling tool LTT from 40% to 0.41%.

Areas of future investigation include issues related optimizing the framework APIs, designing algorithms for system optimizers, and developing a full-feature ULF.

Acknowledgments

This research is sponsored in part by National Science Foundation under grants MIP-9714370 and CCR-0073377. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The second author's research is partially supported by the Manufacturing Center at Tennessee Tech University.

References

[1] P. Crowley and J.-L. Baer, "On the use of trace sampling for architectural studies of desktop applications," in *Proceedings of the 1999 SIGMETRICS Conference*, May 1999.

[2] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos, "ProfileMe : Hardware support for instruction-level profiling on out-of-order processors," in *International Symposium on Microarchitecture*, 1997, pp. 292–302.

[3] S. Graham, P. Kessler, and M. McKusick, "Gprof: A call graph execution profiler," in *SIGPLAN Symposium on Compiler Construction*, June 1982, pp. 120–126.

[4] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu, "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization," in *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999, pp. 136–147.

[5] S. S. Sastry, R. Bodik, and J. E. Smith, "Rapid profiling via stratified sampling," in *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, July 2001.

[6] L. Schaelicke, A. Davis, and S. A. Mckee, "Profiling I/O interrupts in modern architectures," in *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2000.

[7] Z. Wang and M. D. Smith, "Progressive profiling: A methodology based on profile propagation and selective profile collection," in *Proceedings of 4th Workshop on Feedback-Directed and Dynamic Optimization*, Dec. 2001, pp. 105–116.

[8] A. Goldberg and J. Hennessy, "Mtool: An integrated system for performance debugging shared memory multiprocessor applications," *IEEE Transactions on Parallel Distributed Systems*, vol. 4, no. 1, pp. 28–40, 1993.

[9] J. Reiser and J. Skudlarek, "Program profiling problems, and a solution via machine language rewriting," *SIGPLAN Notices*, vol. 29, no. 1, pp. 37–45, Jan. 1994.

[10] Intel. Vtune performance analyzers. [Online]. Available: http://developer.intel.com/software/products/vtune/index.htm

[11] Compaq Corp. iprobe tool suite. [Online]. Available: http://freshmeat.net/projects/iprobetoolsuite/

[12] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?" *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 357–390, Nov. 1997.

[13] X. Zhang, Z. Wang, and N. Gloy, "System support for automatic profiling and optimization," in *Proceedings of the 16th Symposium on Operating Systems Principles*, Oct. 1997, pp. 15–26.

[14] ARM Ltd. Multi-ICE and MultiTrace. [Online]. Available: http://www.arm.com/devtools/debug_tools

[15] C. B. Zilles and G. S. Sohi, "A programmable co-processor for profiling," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA 7)*, Jan. 2001.

[16] T. Anderson and E. Lazowska, "Quartz: A tool for tuning parallel program performance," in *Proceedings of the 1990 ACM SIGMETRICS Conference*, 1990, pp. 115–125.

[17] K. Yaghmour and M. Dagenais, "Measuring and characterizing system behavior using kernel-level event logging," in *Proceedings of 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[18] J. L. Griffin, J. Schindler, S. W. Schlosser, J. S. Bucy, and G. R. Ganger, "Timing-accurate storage emulation," in *Proceedings of the Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[19] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber, "Structure and performance of the direct access file system(DAFS)," in *Proceedings of USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002, pp. 1–14.

[20] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. K. Hollingsworth, J. Saltz, and A. Sussman, "Tuning the performance of I/O intensive parallel applications," in *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*. Philadelphia: ACM Press, 1996, pp. 15–27.

[21] C. Lever and P. Honeyman, "Linux NFS client write performance," in *Proceedings of 2001 USENIX Annual Technical Conference*, June 2001.

[22] T. Conte, B. Patel, K. Menezes, and J. Cox, "Hardware-Based profiling: An effective technique for profile-driven optimization," *International Journal of Parallel Programming*, vol. 24, no. 2, Feb. 1996.