SAND2015-7833C

Extending LDMS to Enable Performance Monitoring in Multi-Core Applications

Steven Feldman, Deli Zhang,
University of Central Florida
Orlando, FL
Email: sfeldman@knights.ucf.edu

Damian Dechev, and James Brandt
Sandia National Laboratories
Livermore, CA
Email: brandt@sandia.gov

Abstract—**Identifying design patterns that limit the performance of multi-core algorithms is a challenging task. There are many known methods by which threads synchronize their actions and each method may exhibit different behavior in different use cases. These use cases may vary in regards to the workload being executed, number of parallel tasks, dependencies between these tasks, and the behavior of the system scheduler. Restructuring algorithms to overcome performance limitations requires intimate knowledge on how these algorithms behave in hardware. In our experience, we have found a lack of adequate tools to gain such knowledge.**

To address this, we have enhanced and implemented additional sampler modules for OVIS's Lightweight Distributed Metric Service (LDMS) [1] to monitor hardware performance counters. These modules provide an interface by which LDMS can utilize the PAPI library, Linux perf tools, or RAPL to collect hardware performance counters of interest. Using these samplers, we plan to monitor the intra-node behavior, including contention for node level shared resources, of multi-core applications for a diverse set of use cases. We are currently exploring how the values reported are affected by the level of concurrency, the synchronization methodologies, and progress guarantees. We hope to use the information to identify ways to restructure the tested algorithms to increase their performance.

I. Introduction

Developing high performing multi-core algorithms is a challenging task. This task is complicated by numerous factors that may impact the performance of a multi-core application. These factors include the hardware the application is executed on, the number of executing threads, memory access patterns, and how the algorithm is currently being used. To better understand the impact of these factors we are extending existing and developing new modules for OVIS's Lightweight Distributed Metric Service (LDMS) [1]. These modules are designed to monitor both hardware and software events and by analyzing the results of experiments, we hope to identify ways to improve the performance of multi-core algorithms.

The goal of the OVIS project [2] is to enable more effective use of high performance computational clusters by providing a greater understanding of how applications use resources. Use cases of interest include competition for shared resources, detection of abnormal system conditions, and responding intelligently to conditions of interest. The *Lightweight Distributed Metric Service (LDMS)* [1] module is responsible for data collection, transport, and storage for consumption by OVIS, and/or third party analysis, visualization, and response modules. LDMS contains many sub-modules, referred to as samplers, which collect data that is of interest.

Our contribution is the enhancement of the perf_event [3] sampler and implementation of two additional samplers for the PAPI [4] and RAPL [5] libraries. These libraries provide access to a variety of hardware performance monitoring units and power consumption monitors. We have performed preliminary tests using these samplers and have identified patterns that may explain certain performance behavior of multi-core applications. With this knowledge, we hope to identify ways to restructure algorithms to overcome performance limitations.

II. Sampler: perf

Linux's perf tools, also referred to as perf_event [3], is a tool that provides access to CPU performance counters, tracepoints, kprobes, and dynamic tracing. These metrics are accessed through a generalized abstraction layer that removes the need to modify code when moving from one architecture to another architecture that supports similar metrics.

Events can be tracked globally or limited to events triggered by a specified process and they can be further refined to events that occur on a specified core. Because this tool can be utilized by *root* for monitoring of any supported events, it can be used for global periodic monitoring as a system service. The monitored information, taken in conjunction with scheduler and resource manager logs, can provide valuable insight into how a user application is utilizing node level resources on a per-core/per-subsystem granularity and how this varies across the user application's node allocation.

A. Sampler Enhancement

This sampler implementation enables LDMS to monitor all hardware and software events supported by the perf_event tool. While this sampler had already been written, the user interface for configuration was difficult to use and it lacked the ability to monitor the *uncore* counters. Thus our contribution to this sampler is a simplified script interface for configuration and extension to the *uncore* counters.

After loading the perf_event sampler module (*ldmsctl$ load name=perfevent*) and initializing it (*ldmsctl$ config name=perfevent action=init component_id=<int> set=<string>*), a user can track a particular event by calling the configuration option, specifying the event codes, process id, cpu core id, and lastly an identifying name for the event (*ldmsctl$ config name=perfevent action=add pid=<int> cpu=<int> type=<int> id=<int> metricname=<string>*). If the developer specifies a cpu core value of -1, it will track the specified process across all cpu cores and if a pid of -1 is specified,all processes on a single cpu core will be tracked.

The number of events and processes which can be tracked by this sampler is only limited by the number supported by the perf_event library, which may vary on the hardware architecture. Perf_event provides a utility program, *perf list*, that displays a list of supported events for the current architecture.

III. Sampler: PAPI

The Performance API or PAPI project is aimed at developing a standard programming interface by which hardware performance counters are accessed [4]. One of PAPI's most significant features is its portability; source code which uses its interfaces can be run on multiple different architectures with minimal concern for compatibility. Additionally, PAPI provides tools to determine the availability and compatibility of various hardware counter events supported on a particular system. One of PAPI's limitations, however, is that it can only be programmed by a user to collect information related to that users processes and their children. It does not allow user *root* to monitor globally and thus cannot be used to provide system wide monitoring.

A. Sampler Implementation

Our sampler implementation enables OVIS to monitor all hardware and software events supported by the PAPI library. After loading (*ldmsctl$ load name=spapi*) and initializing (*ldmsctl$ config name=spapi action=init component_id=<int> set=<string>*) the PAPI sampler module, a user can track a particular event by calling the configuration option, specifying the event name, process id, and an identifying name for the event (*ldmsctl$ config name=spapi action=add pid=<pid> event=<string> metricname=<string>*).

The API to PAPI differs to that of perf_event in two regards. The first is that it does not require a numerical event code; instead a user is able to use a string to identify the event to track. The second is that it does not allow event tracking to be limited to a specific core.

The number of events and processes which can be tracked by this sampler is only limited by the number supported by the PAPI library, which may vary based on architecture. PAPI provides two utility programs, *papi_avail* and *papi_component_avail*, that display a list of supported events for the current architecture.

PAPI is capable of automatically monitoring all threads of a forked process, but not of an attached process, which is how our sampler uses PAPI to monitor an application. To overcome this, a user can explicitly configure the sampler to track each child process. For applications that use a large number of threads or for applications that create and destroy threads, this is not an applicable solution. We are currently investigating alternative libraries and tools that may provide a means by which to overcome this limitation.

IV. Sampler: Running Average Power Limit

Running Average Power Limit or *RAPL* is an interface available on Intel Sandy Bridge or newer processors that provides the ability to monitor, control and receive notifications on CPU power consumptions.

A. Sampler Implementation

Our implementation relies on PAPI's RAPL component [6], which requires root privileges and perf tools 3.14 or newer. This component reads the RAPL values directly from the model-specific registers by using the x86-msr driver. It tracks RAPL measurements on a per CPU socket basis, but not a per-process basis.

After loading the RAPL sampler module, a user can track power consumption after an initial configuration (*ldmsctl$ config name=rapl action=init component_id=<int> set=<string>*).

V. How we plan to use the samplers

Having completed the implementation of the sampler modules, we have identified the following hardware events that we believe may provide insights into the behavior of multi-core algorithm:

- Instructions: all, load, store, branch, failed conditional instructions...
- Cache behavior: hits, misses, reads, writes, ...
- Cycles: total, stalled
- Branch miss predictions
- Hardware interrupts
- Power consumption

To facilitate our experimental evaluation we use a synthetic tester designed to simulate how multi-core applications may use a concurrent container. During experimental evaluation, we will test how different use cases, levels of concurrency, synchronization techniques, and container types affect performance.

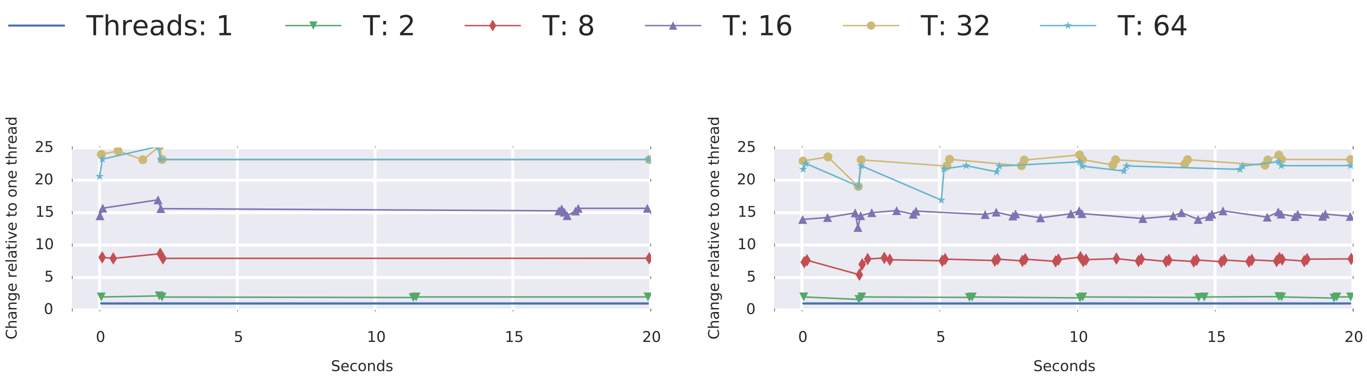
While performing these experiments, we will track several of the aforementioned hardware and software counters then compare the reported values from different use cases. Use cases may differ in a number of ways, such as the number of executing threads, the types of operations being executed by each thread, or even the container implementation used. We also plan to explore the effect of having multiple sets of threads, each executing operations with different probabilities.

VI. Initial Insights

We have performed some initial experiments to evaluate the effectiveness of the PAPI sampler. To enable sampling of multi-threaded applications, we explicitly add the process ids of each thread created by the application. In these experiments, we explore how the number of cycles and instructions the performance of the container to the number of cycles and instructions consumed by the application.

#Threads	Change in Work		Change in Cycles		Change in Stalled		Change in Instructions	
Stack	Hash Map	Stack	Hash Map	Stack	Hash Map	Stack	Hash Map	
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	-0.24	0.80	-0.09	0.78	-0.26	0.96	-0.31	0.91
8	-0.57	5.93	0.59	5.90	-0.55	6.37	-0.53	5.58
16	-0.81	10.24	0.62	9.71	-0.81	12.81	-0.83	11.84
32	-0.91	13.37	0.28	11.61	-0.90	14.57	-0.91	13.04
64	-0.90	25.07	-0.02	10.50	-0.90	31.05	-0.90	26.72

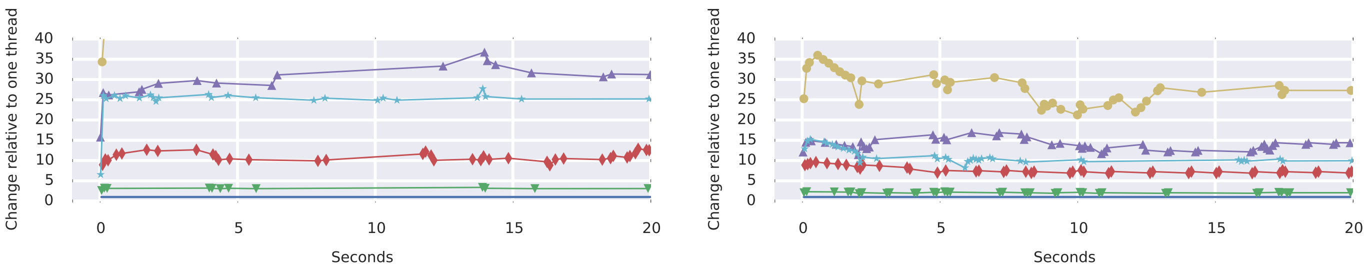
Fig. 1: Algorithm Performance Comparison



(a) Lock-Free Stack

(b) Wait-Free Hash Map

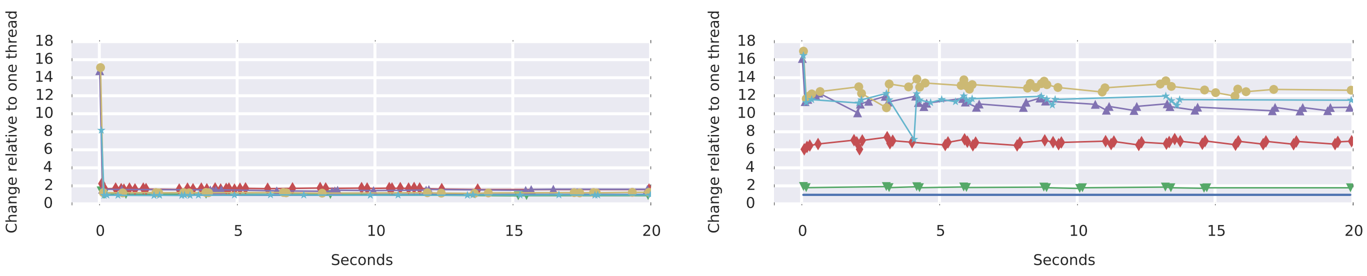
Fig. 2: Relative change in cycles compared to single thread execution.



(a) Lock-Free Stack

(b) Wait-Free Hash Map

Fig. 3: Relative change in stalled cycles compared to single thread execution.



(a) Lock-Free Stack

(b) Wait-Free Hash Map

Fig. 4: Relative change in instructions compared to single thread execution.

Our test begins by having a main thread construct and initialize a container. Next, it creates a set of worker threads and then sleeps for a few seconds. While the main thread sleeps, we attached LDMS samplers to each thread and set them to sample every $100ms$. Upon waking, the main thread signals the worker threads to begin execution, after which it sleeps for 20 seconds, and then when it wakes it signals the end of execution. Each worker thread executes operations based on the typical use case of the container being used.

Fig. 1 presents cumulative change in tracked metrics in each experiment and Fig. 2-4 depicts the values reported by different hardware metrics throughout the experiment. Each line is the performance at a specific level of concurrency *relative* to single thread execution and each point represents a 2% change in reported value from the previous point.

We see in Fig. 1 that the stack's performance decreases as the number of threads accessing it increases and, conversely, the hashmap's performance increases as the number of threads increases. Through manual examination, we attribute the poor performance of the stack to the contention created by using a single shared pointer. This is in contrast to the hashmap's implementation, which diffuses contention across a region of memory. This diffusion, creates disjoint parallelism, which we attribute to the hashmap's performance scalability.

Using Fig. 2 we see that the hashmap cycle usage varies significantly more than the stacks usage, especially at higher thread levels. However, they both exhibit roughly the same relative change in cycles compared to single thread execution.

Even though the containers consume relatively the same number of cycles, we see in Fig. 4 that, on average, the hashmap's relative increase in instructions is twice that of the stack's increase. Fig. 3 reveals that this discrepancy maybe caused by stalled cycles. We see that the stack increases in stalled cycles much faster than the hashmap. But this increase does not appear to explain all of the performance differences.

When the number of threads increase from 1 to 64, the number of operations completed decrease by 90%. If we divide the number instructions and cycles by the number of operation, we see an increase in instructions per operation and cycles per instruction. On average it takes 3,000 instructions and 3,100 cycles to perform one operation with one thread and with 64 threads, it increases to 260,000 and 696,000, respectively.

Unlike the stack, the performance of the hashmap increases with the number of threads. Increasing the number of threads form 1 to 64, leads to a factor of 26 increase in number of operations completed. Interestingly, the total number of instructions and cycles only increased by a factor of 15 and 21, respectively. This surprising reduction means that the the average number of instructions and cycles needed to execute an operation was reduced by 42.3% and 32.4%. We are unsure as to the cause of this decrease, but will be investigating this behavior further.

VII. Future Work

We are in the initial stages of our research and are currently exploring different methodologies and technologies. A priority of ours is to identify suitable multi-core applications to augment our synthetic testing. At the same time, we are expanding our synthetic tests to gather data form a wider variety of use cases. Our initial insights are promising, but more work needs to be done to determine how this information can be applied to improve the design and implementation of multi-core algorithms.

In addition to the samplers that we have already implemented, we plan to implement additional samplers to provide access to more hardware monitors. We are currently exploring the applicability of the powerAPI [7] library to overcome some limitations in the RAPL library.

Acknowledgment

Our enhancement of OVIS is funded by National Science Foundation grants NSF ACI-1440530 and NSF CCF-1218100. OVIS is a project of Sandia National Laboratories, Albuquerque NM, 87123 and collaborative partner Open Grid Computing, Austin TX., SAND 2006-2519W. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 154–165. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.18

[2] J. M. Brandt, B. J. Debusschere, A. C. Gentile, J. R. Mayo, P. P. Pébay, D. Thompson, and M. H. Wong, "Ovis-2: A robust distributed architecture for scalable ras," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[3] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, 2013, p. 80.

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.

[5] V. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with papi," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, Sept 2012, pp. 262–268.

[6] H. McCraw, J. Ralph, A. Danalis, and J. Dongarra, "Power monitoring with papi for extreme scale architectures and dataflow-based programming models," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, Sept 2014, pp. 385–391.

[7] A. Bourdon, A. Noureddine, R. Rouvoy, and L. Seinturier, "Powerapi: A software library to monitor the energy consumed at the processlevel," *ERCIM News*, vol. 2013, no. 92, 2013.