Evaluating A Prototype KnightShift-enabled server

Daniel Wong
University of Southern California
wongdani@usc.edu

Murali Annavaram
University of Southern California
annavara@usc.edu

***Abstract*—Energy cost of operating servers is a growing concern, particularly in data center settings. Even though servers have low utilization they are rarely shut down due to data and performance requirements. Previous work has shown that management processors can be used to satisfy I/O only requests while allowing servers to sleep, significantly reducing energy usage. By increasing the role of management processors to include the handling of requests with limited computing demands, these "assist nodes" can process all request during low-utilization periods while allowing the primary server to be placed in low power mode. We call this approach KnightShift. In this research we implemented a fully-functional prototype of KnightShift to demonstrate its feasibility. We run a real-world workload, WikiBench, and measure a 34% energy savings while only increasing average request latency by 4%.**

I. Introduction

Energy consumption of computing servers is a critical concern, particularly in datacenter settings where thousands of servers are deployed to meet the demands of modern service-oriented architectures. Server energy costs comprise a significant fraction of the total operating cost of datacenters. However, many servers operate at low utilization and still consume significant energy due to the lack of ideal energy proportionality [4]. Server virtualization coupled with workload migration and consolidation can boost utilization on some servers and allow others to be placed in sleep mode. Unfortunately, server shutdown/sleep is not always possible for two reasons. First, servers may continuously service many low utilization requests interspersed occasionally with high utilization requests. It is necessary to service low and high utilization requests without paying the full penalty of sleep and wakeup delays, which are in the order of tens of seconds. Second, many large datacenters use direct-attached storage (DAS) architecture. A DAS architecture distributes data across disks attached to many separate servers. DAS architectures scale well with growing data size as commodity disks can be added to increase storage without the need for expensive I/O solutions. The disadvantage of this architecture is that nearly all data-intensive applications, such as web search, e-commerce, and social networking necessitate access data stored on multiple servers. The need to service remote requests for data precludes servers from being powered down or placed in low-power modes due to their long wake-up latencies. In this research we focus on reducing energy consumption of servers used in DAS design. An alternative to DAS design is network-attached storage (NAS) architecture. NAS relies on high bandwidth central storage to satisfy requests from any server. Hence, conceivably servers can be placed in low power state when idle. However, NAS scalability is a concern since costs grow super-linearly with bandwidth.

Previous work has shown that management processors can be used to satisfy I/O only requests while allowing servers to sleep, significantly reducing energy usage [5]. The approach was termed as KnightShift. In this paper, we extend the KnightShift work in two ways. First, we use a very low power compute assistant, rather than a management processor, to front power hungry primary server. The assistant node has sufficient capabilities to handle low-utilization computation and I/O requests directed toward the primary server. By shifting the burden of servicing low utilization and I/O access requests to an assist node, called the Knight, we can place primary servers in low power state and wake them up only to service requests that require the full computing resources of the server, namely the computing core and large amounts of main memory. KnightShift provides significant power savings but requests may see an increased latency if service by Knight. Hence, KnightShift trades-off increased latency with significant power savings. In [5], KnightShift was evaluated using a simple queueing model. In this paper, we prototype a KnightShift system using commodity parts to validate previous claims and demonstrate the feasibility and real-world performance of a KnightShift system. We run a real-world workload, WikiBench [11], and measure a 34% power savings while only increasing average request latency by 4%. We also evaluated KnightShift against a variety of datacenter workload traces to see the affect of different workload patterns on KnightShift.

The rest of the paper is organized as follows. Section II provides an overview of KnightShift. Section III describes how KnightShift can be implemented using existing hardware and operating system services and describes one potential mechanism to enhance KnightShift with additional hardware support. Section IV shows the results of KnightShift running under a real-world workload, WikiBench. Section VI describes the results of our approach using a collection of large scale datacenter traces from our institute's production facility. Section VII describes related work and KnightShift's contribution, and we conclude in section VIII.

II. The KnightShift approach

Figure 1 shows a high-level overview of how KnightShift works. A KnightShift enhanced server has two components. A primary server, which is the default server for processing requests, and the Knight server, which acts as a compute assistant to the primary server. This system receives a request for service from an external source, such as a load balancer in a

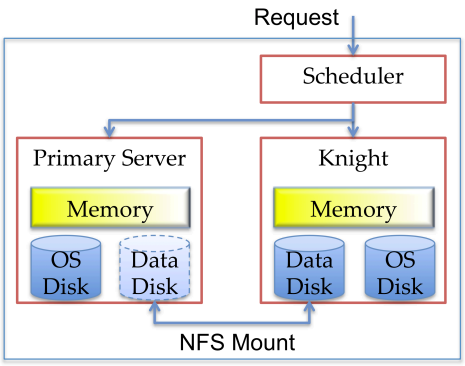


Fig. 1: KnightShift Overview

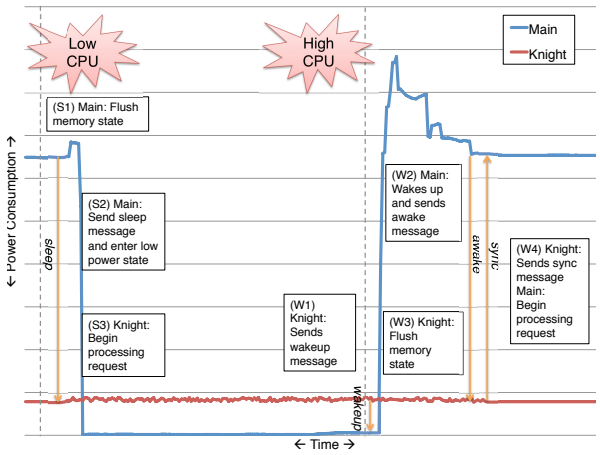


Fig. 2: Operation of a KnightShift system

datacenter. A KnightShift enhanced server has a simple front-end scheduler that runs on the Knight system which determines whether the incoming request is handled by the primary server or the Knight. If the primary server is active and operating at a load that Knight cannot handle, the request will be routed to the primary server. If the primary server is inactive and the incoming request requires larger computing resources than the Knight's capabilities, then the primary server is woken up and assigned the request. In all other cases Knight will handle the request without routing the request to the primary server. Thus Knight can allow the primary server to stay in a low-power state (or even turned OFF) for extended periods of time by handling the low utilization requests.

In Figure 1, note that Knight and primary server can both access the disk data. To preserve data consistency either Knight or primary server can modify the disk content at any time, but not both. This issue will be further discussed in a later section.

Figure 2 illustrates the operational details of the KnightShift system. The Y-axis represents power consumption and X-axis is time. This figure is not to scale and is used solely for illustrative purposes to explain the workings of KnightShift. The blue and red line indicates the power consumption of the primary server and Knight, respectively. We collected this data during a real experiment done using our prototype implementation of KnightShift described later in Section IV. Initially, the primary server is actively processing requests. Upon encountering a period of low utilization, KnightShift will switch to Knight mode. To prevent the Knight from operating on stale data, the main server must first flush it's buffered disk modifications to disk . The flush event is labeled as S1 in the figure. Once the flush has completed, the main server sends a *sleep* message to the Knight (S2) and enters a low-power state. Once the sleep message is received, the Knight begins processing future requests (S3). Note that when the primary server flushes the buffered state from memory to disk there is a small spike in power consumption to the left of event S1 as it scans memory and activates disks for writing. Once the primary server moves to a low-power state, its power consumption drops significantly, even to zero, depending on the low-power state.

After event S3, all requests will be handled by Knight, while the primary server is in low-power state. When the Knight encounters a high utilization request, the Knight issues a *wakeup* message to the main server (labeled W1). The main server will wake up and send an *awake* message upon bootup (W2). A spike in power consumption is associated with waking up the primary server as seen by the large power spike to the right of event W1. While the main server is waking up, the Knight will continue to process requests until it receives the *awake* message. Upon receiving the *awake* message, the Knight will flush its buffered disk state from it's main memory to disk. Once the flush is complete Knight sends a *sync* message (W3), notifying the primary server that it can begin processing request. After event W3, the primary server handles all requests. The Knight remains powered on, routing requests to the primary server as well as listening to messages from the primary server in case the primary server wants to enter low-power state. In our current research, we leave the Knight always ON and hence during the entire duration Knight consumes a small amount of power, as seen by the continuous red line in Figure 2.

III. KnightShift Prototype Implementation

A. Hardware Emulation Setup

To measure the impact of the proposed approach using real hardware and existing operating system support, we designed an emulator that emulates the KnightShift mechanism. The emulator details are shown in Figure 3. Our KnightShift hardware consists of three compute nodes: a large powerful server that acts as the primary server, a small low-power system that acts as the Knight, and a third compute node that acts as a client to generate requests and to collect power measurement data. We use a Shuttle XS35 low-power compute node with an Intel Atom D525, 1GB of ram and 500GB hard drive as the Knight. This computer uses 15W at idle and 16.7W of power at full load. The CPU and memory consume 9W of power. 6W is the disk and the base motherboard power during idle time. The primary server is a Supermicro

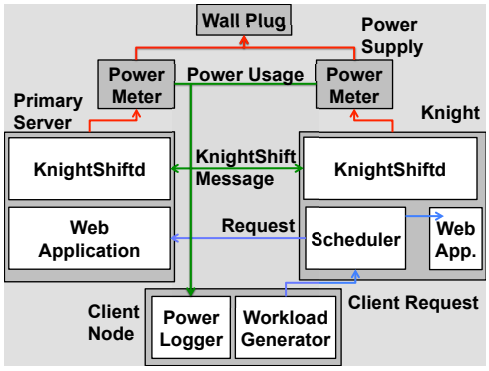


Fig. 3: KnightShift Emulation setup

server with dual 4-core Intel Xeon L5630, 36GB of ram, 500GB hard drive. The primary server uses the default DVFS scaling to reduce its power consumption during low utilization periods. DVFS is enabled at all times in our evaluation. The primary server uses 156W at idle and 205W at full load. Our primary server can enter two low-power states: a sleep state where the CPU is shutdown but the memory is refreshed, a hibernate state where the CPU and memory are both turned off by storing the memory state in disk. The delay to put the server in sleep state is 5 seconds and the delay to put the server in hibernate state is 10 seconds. Our primary server consumed significant amount of power, 120W, even during sleep state. Hence, we always put the server in hibernate state in all our results. During hibernate, our server consumes only 0.5W. The delay to wakeup the server from hibernate state is 20 seconds. We will use the term low-power state to mean hibernate throughout the rest of this paper. The client node has no impact on the power usage of KnightShift because its primary purpose is to emulate the client that generates requests and measure the power consumption of the emulation setup. The power consumption of each of the Supermicro server and XS35 node are measured using two *Watts Up? Pro* meters. The power consumption data is averaged over one minute granularity. All computers run Ubuntu 10.10.

We measured the throughput of the Shuttle XS35 and Supermicro servers to benchmark how capable the Knight is compared to the primary server. We used the apachebench test [7] that is commonly used to benchmark webservers. From this stress test we measured that our Knight is capable of providing 15% of the throughput of the primary server. We define Knight capability as simply the fraction of the throughput that Knight can provide compared to the primary server. For example, if the Knight has a capability level of 10%, it means that the Knight is capable of handling jobs that require less than or equal to 10% of the primary server utilization.

B. KnightShift Software Implementation:

To implement KnightShift on the above hardware setup, we need a software glue logic that enables the following capabilities: sharing disk, sharing network, ability to sleep/wakeup server, monitor utilization, and handle KnightShift messages. KnightShift messages (*sleep, wakeup, awake, sync*) are used to communicate between the main server and the Knight as described earlier. Software support for KnightShift is provided through a set of scripts and background processes. Our KnightShift implementation runs on Ubuntu Linux and many of our functionalities make use of Linux system calls.

Sharing Disk: The Knight must be closely coupled to the primary server so that it can concurrently access the disk storage of the primary server. If both the Knight and main server share an I/O subsystem, then any hard drive connected to the southbridge will be visible to both the Knight and main server. At the operating system level, the hard drive and its partitions will be exposed as a device file (e.g. /dev/sda, /dev/sda1,etc). However, mounting a disk as a shared read/write partition on multiple systems *at the same time* can lead to data inconsistency, if the two systems update some disk block. To solve this, both the primary server and Knight each have their own system partition (with its own operating system and applications), which avoids any potentially fatal corruption to the operating system. The data partition is located in the Knight and is made available to the primary server through NFS. To avoid corruption to the data partition (which contains the web pages, databases and all other data that is necessary to process incoming requests), *only one system* is allowed to mount the data partition at a time. Mounting and unmounting of disk partitions are achieved through the `mount` and `umount` commands.

In Figure 2, the main server would unmount the shared data partition at the end of its flush operation (S1) and the Knight would mount the data partition when it receives the *sleep* message (S3). Similarly, the Knight would flush and unmount the data partition when it receives the *awake* message from the main server (W3). The main server would mount the data partition when it receives the *sync* message. Thus we force the data partition never to be concurrently accessed by the primary server and Knight.

Sharing Network: In KnightShift, the Knight and primary server must share a network connection so as to appear as a single server to external servers. KnightShift can use sideband ethernet to share a single physical ethernet port between the Knight and primary server. Each of the two systems gets its own IP address but we made the Knight's IP address as the only publicly visible IP address. In other words, for the outside world the entire KnightShift system appears as a single server with a single IP address. For instance in a datacenter setup each KnightShift system appears as a single server to a load balancer. The load balancer assigns a request to the KnightShift system. The request is received by the Knight and the scheduler running on the Knight decides whether to allow the Knight to handle the request or to wakeup the primary server.

Ability to sleep/wakeup server: Most modern server systems allow a server to enter low-power state by controlling the power states (hibernate, sleep) and also wakeup on a network message request. The Linux operating system provides the pm-

utils [8] package to allow a server to enter low-power state. The pm-utils package contains a set of commands that allow the primary server to hibernate (using `pm-hibernate`) or sleep (using `pm-suspend`). In order to wakeup the main server from a low-power state, we use of the wake-on-lan [18] mechanism, a popular mechanism to wake up a sleeping computer by sending a "magic packet" to its network interface card. Wake-on-lan allows us to send a "magic packet" from the Knight to the main server to wake the main server from a low-power state.

KnightShift Message and Utilization Monitor: In KnightShift the primary server sleep/wakeup process is controlled by system utilization. To support utilization monitoring and KnightShift messages, both the primary server and the Knight runs a daemon, called KnightShiftd. KnightShiftd is implemented as a python script and functions as the control center for the KnightShift system. KnightShiftd monitors utilization and decides on when to switch modes. KnightShiftd running on the primary server monitors primary servers utilization and decides whether to put the primary server in low-power state. KnightShiftd running on Knight monitors the utilization of a request running on the Knight system and decides when to wakeup the primary server.

In addition to utilization monitoring, KnightShiftd listens to KnightShift messages over ethernet through defined sockets. On the Knight system, KnightShiftd listens for the *sleep* message, which indicates that the primary server entered a low-power state. The Knight system then mounts the data partition of the disk and begins handling requests. When KnightShiftd receives *awake* message it implies that the primary server is ready to handle requests. Then the Knight system flushes buffered changes from memory to disk and unmounts disks and send the *sync* message. After a synch message Knight is unable to handle any request since it does not have access to the data partition on the disk.

The Knight also sends the *wakeup* message to wake up the main processor over wake-on-lan. Upon wakeup, the primary server immediately runs the KnightShiftd daemon which sends the *awake* message. The KnightShiftd daemon on the primary server listens for the *sync* message which indicates that the Knight system has completed flushing its memory contents to disk. Hence, the primary server can now mount data partition of disks and begin handling requests.

IV. Evaluation Using Wikipedia

To verify that our hardware and software implementation details actually work correctly (no functional bugs) and to evaluate KnightShift under realistic workloads, we cloned Wikipedia [12] on our KnightShift system and benchmarked it using traces of real requests. Wikipedia consists of two main components, Mediawiki [10], the software wiki package written in PHP, and a backend mySQL database. Database dumps [6] are publicly available from Wikipedia. For our clone, we used a database dump from January 2008, containing over 7 million articles.

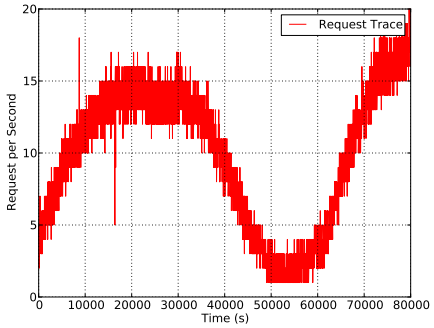


Fig. 4: WikiBench trace

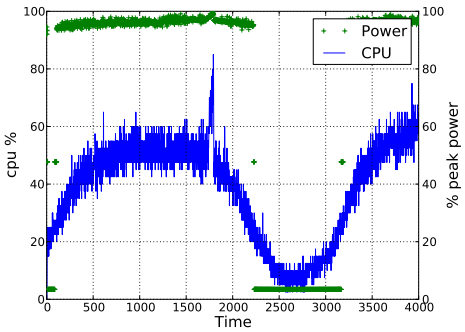


Fig. 5: Peak Power and CPU utilization With KnightShift

To subject our Wikipedia clone to conditions similar to the real Wikipedia, we use WikiBench [16], a wikipedia based web application benchmark. WikiBench provides a framework to generate real traffic by replaying traces of actual Wikipedia traffic. For our evaluation we use one day worth of requests from January 2008. We scale the request traces to a maximum of 20 requests per second to the primary server. The exact request per second distribution over a 24 hour window is shown in figure 4. The primary server utilization oscillates between 0% and 60%, with some utilization periods that are below 15%. Recall from Section III that the Knight system in our setup can provide 15% capability in terms of throughput compared to the primary server. Hence, when the utilization of the primary server falls below 15% for some period, the primary server enters low-power state and allows the Knight system to handle those requests. For the purpose of this paper, we switch to the Knight if the utilization of the primary server falls below 15% for 10 seconds, the time it takes to put the primary system to a low-power state. We did this to ensure that the system does not go to sleep and wakeup right away due to immediately encountering high utilization.

WikiBench Results: To measure the power and latency effects of KnightShift, we setup the WikiBench database on the disk, and installed Mediawiki application on both the Knight and primary server. We then allowed the client node in our setup to generate requests using the WikiBench trace. The CPU and I/O utilization as well as the power usage is logged

	Energy(KWH)	Response Time(ms)
Primary Server	23.27	144
KnightShift	15.35	150
Savings	34%	-4%

TABLE I: Energy consumption and response time of Primary Server vs KnightShift

during the benchmark. Figure 5 shows the cpu utilization of the primary server and Knight on the primary Y-axis. The secondary Y-axis plots the % peak power consumed by the Knight and primary server together.

As can be seen from the CPU utilization data the primary server goes to low-power state every time it's CPU utilization drops below 15% threshold. At that point Knight handles all requests and hence Knight's utilization increases significantly. When Knight is handling requests the aggregate power consumed drops significantly since the primary server enters a low power state. The power consumption drops from about 95% to 5% of aggregate peak power consumption. Table I shows the Energy consumption and the average response time over the benchmark run. In a KnightShift enhanced server, we are able to achieve 34% energy savings with only 4% impact on response time.

V. Trace Based Emulator

In this section we further evaluate our prototype using traces from production datacenters. We collected minute-granularity utilization traces from USC's campus-wide production datacenter for 9 days. The datacenter serves multiple tasks, such as e-mail, learning management system (e.g. Blackboard), distance education (streaming of course lectures), compute intensive application hosting such as CAD tools, and student timeshare servers. Each task is assigned to a dedicated cluster, with data spread across multiple servers. We selected 8 clusters and 4 servers per cluster for trace collection. We were assured by our datacenter operations group that the selected servers within a cluster exhibit a behavior representative of each of the server within that cluster. We used the *sar* tool to collect CPU, memory, and disk utilization, as well as paging activity, kernel and user activity. Figure 6 shows CPU utilization at a per-minute granularity of all the servers traces . We can see that some servers (aludra, nunki, scf, girtab, email) run at less than 20% CPU utilization for nearly 90% of their total operational time. In general, these traces confirm that CPU utilization never reaches 100% but at the same time they also never go down to 0% utilization for extended periods of time, reaffirming prior studies [4], [19], [15].

Traces Evaluation: The client machine emulates a datacenter workload generator using a trace driven approach provided through two components: the parser and load generator. The parser runs on the client node and parses the trace files gathered from our production datacenter to extract the CPU utilization and IO utilization for each time quantum of one minute. The client then generates a single request with the CPU and I/O utilization level that was experienced in our physical datacenter during that minute. The generated request is then sent to the Knight system only. Knight then reads

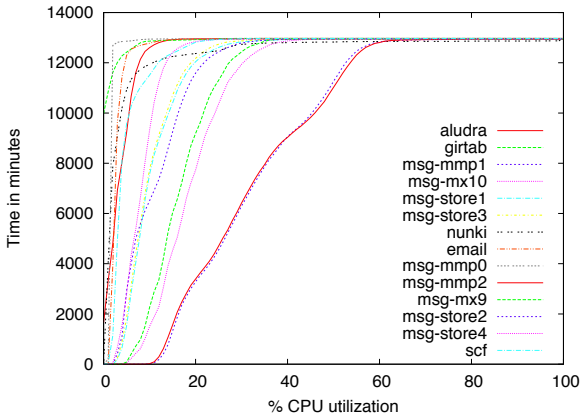


Fig. 6: CPU Utilization vs Time

the client request containing CPU and I/O utilization levels. The scheduler running on the Knight node determines which machine will service the current request from the client. The machine that is scheduled to handle the request will then generate the target CPU and I/O utilization.

The main modification to our KnightShift emulator to support trace based emulation is the inclusion of the load generator module. Instead of running WikiBench application the primary server and Knight run a load generator application capable of separately generating target CPU and I/O utilization.

Modeling Latency: In a trace driven emulation, we only have CPU and I/O utilization provided to us from the datacenter. Since our emulation setup uses a load generator to obtain realistic power measurement, it has no notion of requests, thus we cannot obtain latency impact directly from our emulation setup. By making a simple assumption, it is possible to estimate the response time of requests fairly accurately. To estimate latency impact, we feed the utilization trace into a simple queueing model with arrival rate determined by the utilization trace and constant service time. To obtain request information, we assume that every request requires a fixed utilization. By using utilization as a proxy for arrival rate, we can generate requests to our queueing system based on the utilization trace. Through this queueing model, we can obtain an average latency of a KnightShift system.

Similarly, we do not have latency information for our baseline system. Since the utilization trace is recorded from the baseline system, we simply assume that requests to the baseline system is serviced right away, thus the baseline latency is solely the service time. Recall that we assumed that each requests take up a constant utilization and the queueing model uses a constant service time. By using this as the baseline, we can get a latency impact from our KnightShift system relative to our baseline system.

Modeling Sleep/Wakeup: To accurately model the wakeup

	Baseline	Aggressive
Workload (ms)	144	150 (4.2%)
Trace (relative)	1	1.045 (4.5%)
Error		0.3%

TABLE II: Response Time of Workload vs Trace based emulation

Trace	Baseline	KnightShift	Latency Impact
aludra	34.2	4.1 (87.9%)	6.40%
email	40.0	4.9 (85.5%)	0.99%
girtab	33.8	4.3 (87.2%)	24.24%
msg-mmp1	37.8	40.2 (-6.6%)	31.21%
msg-mx10	36.3	33.8 (7.2%)	150.09%
msg-store1	35.3	23.1 (34.5%)	50.09%
nunki	34.2	11.0 (67.8%)	268.51%
scf	34.5	7.8 (77.5%)	28.19%
wikibench	23.3	15.3 (34.5%)	4.50%
Average Savings		52.8%	

TABLE III: Energy consumption in KWH and savings wrt Baseline of a 15% Capable KnightShift system

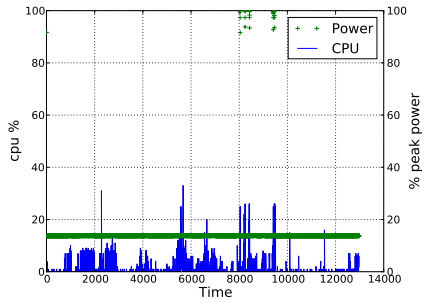
and sleep power, we gathered the power usage of the primary server during a wakeup and sleep process. Through experimentation, it was determined that the wakeup power for the primary server is 167W and sleep power is 122W. We assume that during the entire wakeup period the primary server consumes 167W. For instance, to model an aggressive wakeup time of 3 seconds for the primary server, the primary server burns 167W each second for the 3 seconds of wakeup. Since the Knight is never turned off it continues to burn its usual power all through this wakeup time. So the power usage during wakeup will be the power usage of the Knight plus the wakeup power of the primary server for the entire wakeup time. The transition power observed here can become expensive, especially if we are constantly switching between the primary server and Knight.

VI. Trace-based Results

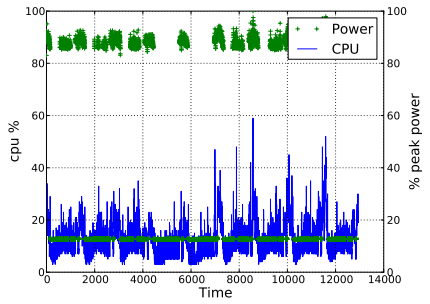
We define the baseline as a system where all requests are always handled by the primary server. We emulated both the baseline system as well as the KnightShift enhanced system to measure the impact on energy consumption and latency. We assume the Knight processor is always on and operates at a constant power level.

Validating Trace Based Emulation: We validate and give confidence that the results of the trace based emulation are similar to the results of the application run. We collected utilization traces from our WikiBench run and re-played the utilization traces through the trace emulator. Table II shows the latency results from running WikiBench on the full KnightShift implementation and the corresponding trace driven emulation run. The latency results are fairly close, within 5%. The differences are primarily due to the approximation of latency computation as described in Section V For brevity, we also validated the power consumption during the run and verified that the error is less than 3%.

Results: Table III shows the results of our trace-based evaluation. For workloads that lack low-utilization periods, we experience high latency impact and low power saving



(a) girtab



(b) nunki

Fig. 7: Power usage of KnightShift run

potentials (e.g. msg-mx10, msg-mmp1, girtab). This is due to the fact that the workload pattern may cause KnightShift to switch to Knight mode and switch back to primary right away, similar to Figure 7a. In certain cases, this can actually lead to an increase in power usage (e.g. msg-mmp1).

For bursty workloads, we experience significant power savings, but with a large latency impact (e.g. nunki). For these workloads, we are able to stay in Knight mode for significant portion of time, but we may not be able to transition quick enough to handle requests during bursts, leading to a large performance impact, similar to Figure 7b. Rather than reacting to utilization changes, it may be possible to proactively switch to a high-performance state anticipating bursts as daily utilization patterns follow the same general pattern, thus leading to significant latency improvements.

If we just account for KnightShift-friendly workloads (aludra, email, girtab, scf, and wikibench), then we can have a 74.52% savings in energy consumption with a modest 12.86% impact in latency. Our results are based only on our prototype. We expect a tightly-integrated solution would greater energy savings and lower latency penalties.

VII. Related Work

In the last few years, power and energy related issues in the context of large scale datacenters have become a growing concern for both commercial industries as well as the government [17]. This concern has become the source of much active research. Barroso [4] showed that energy-proportionality is a chief concern since most enterprise servers

operate at low average utilizations. Fan et al. [19] showed that power provisioning of a datacenter requires understanding the dynamic behavior of the servers. Specifically, they demonstrated that CPU utilization is a good proxy metric for server energy consumption. Ranganathan, et al. [15] showed that greater opportunities for power savings exist at the ensemble level rather at individual server level. They take advantage of inter-server variations to allow active servers to steal power from inactive servers while still meeting power budgets. Our KnightShift approach is motivated and guided by these prior studies. In particular, we demonstrate that significant energy saving opportunities exist even at a single system level by enabling long server sleep times., and we, too, use CPU utilization as a metric to inform transitions between the primary processor and the Knight.

Our work also benefits from techniques such as those proposed in [14]. Here Meisner et al. propose PowerNap, an approach to rapidly transition the entire system between a high and low power state and minimizing idle power. If PowerNap is used in conjunction with KnightShift more opportunities to nap the primary server for longer durations are possible thereby reducing the demand for very short sleep/wake intervals. Barely-alive servers [3] use Remote Direct Memory Access [9] capable network cards to put servers in a barely-alive state where only its memory is active and can be used for caching by other servers in the cluster. This approach essentially improves memory utilization across servers but is not targeted toward servicing any computational demands from remote nodes. Somniloquy [1] supports parts of application level protocols like file download and Instant Messaging as well on an enhanced network card. Amur and Schwan [2] proposed to operate servers in two different power modes while sharing disk through SATA switches. During mode transition, the virtual machine will migrate from a high performance node to a low performance node. While KnightShift has a similar hardware architecture, we propose to have applications running independently on each node coordinating mode transitions, avoiding the overhead of virtual machine migration. Our work is analogous to Server Consolidation [13], which dynamically size clusters to web-server workloads at the datacenter level. Server consolidation suffers from high overheads due to data migration and inefficient use of datacenter space. KnightShift aims to be integrated into a single server, offering the benefits of server consolation with less overhead and more efficient datacenter space utilization.

VIII. Conclusions

In this paper we present KnightShift, a novel approach to front a primary server with a low power compute node called the Knight. The low power compute node is capable of handling small utilization requests on behalf of the primary server. By shifting the burden of small jobs to the Knight system we enable the primary server to sleep for extended periods of time. We described the architecture of KnightShift and designed an emulation setup to closely mimic the primary server and the Knight. We used a broad selection of traces collected from a production datacenter and a Wikipedia-based benchmark to evaluate the benefits of KnightShift. Using our emulator to evaluate KnightShift, we show that we can reduce energy usage by an average of 52.8%. In addition, we demonstrate a very limited impact on job latency for jobs with long periods of low utilization and limited bursts in utilization. Overall, we believe that KnightShift holds strong promise for reducing energy consumption in enterprise-scale computing.

Acknowledgement

This work was supported by NSF grants CAREER-0954211, CCF-0834798.

References

[1] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: augmenting network interfaces to reduce pc energy usage. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 365–380, Berkeley, CA, USA, 2009. USENIX Association.

[2] H. Amur and K. Schwan. Achieving power-efficiency in clusters without distributed file system complexity. In *Proceedings of the 2010 international conference on Computer Architecture*, ISCA'10, pages 222–232, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] V. Anagnostopoulou, S. Biswas, A. Savage, R. Bianchini, T. Yang, and F. T. Chong. Energy conservation in datacenters through cluster memory management and barely-alive memory servers.

[4] L. Barroso and U. Hlzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[5] S. Ghosh, M. Redekopp, and M. Annavaram. Knightshift: Shifting the i/o burden in datacenters to management processor for energy efficiency. In *WEED 10: Workshop on Energy-Efficient Design*, 2010.

[6] http://dumps.wikimedia.org/. Wikipedia database dumps, Retrieved Aug 2011.

[7] http://httpd.apache.org/docs/2.0/programs/ab.html. ab - apache http server benchmarking tool, Retrieved Aug 2011.

[8] http://pm utils.freedesktop.org/wiki/. pm-utils, Retrieved Aug 2011.

[9] http://www.ietf.org/rfc/rfc5040.txt. Remote direct memory access protocol specification, Retrieved June 2010.

[10] http://www.mediawiki.org/wiki/MediaWiki. Mediawiki, Retrieved Aug 2011.

[11] http://www.wikibench.eu. Wikibench, Retrieved Aug 2011.

[12] http://www.wikipedia.org. Wikipedia, the free encyclopedia, Retrieved Aug 2011.

[13] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 46–56, New York, NY, USA, 2007. ACM.

[14] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. *SIGPLAN Notices*, 44(3):205–216, 2009.

[15] P. Ranganathan, P. Leech, D. Irwin, and J.Chase. Ensemble-level power management for dense blade servers. *SIGARCH Comput. Archit. News*, 34(2):66–77, 2006.

[16] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Comput. Netw.*, 53:1830–1845, July 2009.

[17] www.energystar.gov/index.cfm?c=prod_development.server_efficiency. Report to congress on server and data center energy efficiency. Retreived July 2009.

[18] www.lagoas.com/Produtos/User%20Manager/WP/Wake_On_LAN.pdf. White paper: Wake on lan technology. Retrieved June 2010.

[19] X.Fan, W. Weber, and L. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 13–23, 2007.