# PerfBound: Conserving energy with bounded overheads in On/Off-based HPC Interconnects

Karthikeyan P. Saravanan† and Paul M. Carpenter†
†*Barcelona Supercomputing Center (BSC)* and *Universitat Politècnica de Catalunya (UPC), Spain*

**Abstract**—Energy and power are key challenges in high-performance computing. System energy efficiency must be significantly improved, and this requires greater efficiency in all subcomponents. An important target of optimization is the interconnect, since network links are always on, consuming power even during idle periods. A large number of HPC machines have a primary interconnect based on Ethernet (about 40% of TOP500 machines), which, since 2010, has included support for saving power via Energy Efficient Ethernet (EEE). Nevertheless, it is unlikely that HPC interconnects would use these energy saving modes unless the performance overhead is known and small. This paper presents PerfBound, a self-contained technique to manage on/off-based networks such as EEE, minimizing interconnect link energy consumption subject to a bound on the performance degradation. PerfBound does not require changes to the applications and it uses only local information already available at switches and NICs without introducing additional communication messages, and is also compatible with multi-hop networks. PerfBound is evaluated using traces from a production supercomputer. For twelve out of fourteen applications, PerfBound has high energy savings, up to 70% for only 1% performance degradation. This paper also presents DynamicFastwake, which extends PerfBound to exploit multiple low-power states. DynamicFastwake achieves an energy–delay product 10% lower than the original PerfBound technique.

**Index Terms**—Energy Efficient Interconnects, Energy Efficient Ethernet, Fast-Wake, Deep-Sleep

## 1 Introduction

Energy and power are key challenges in high-performance computing. Top HPC machines require many megawatts of power, incurring millions of dollars in energy costs. The proposal by DARPA in 2008 envisioned an exaflop supercomputer by 2018 with a power budget of about $20\,\mathrm{MW}$. The top supercomputer as of November 2016 the Sunway TaihuLight with $93\,\mathrm{PF}$ in performance, which consumes $15\,\mathrm{MW}$ of power. Substantial improvements in energy efficiency and power consumption are therefore required to reach $1\,\mathrm{EF}$ within the $20\,\mathrm{MW}$ goal by 2018–2020.

High-performance interconnects are known to consume a significant portion of the total system energy: up to 12% at full load, and more when the CPU and memory are not fully loaded [7]. Of this, up to 65% is due to the interconnect links, which are essentially always on, continually transmitting signals for link alignment and synchronization [10], and consuming full power even when idle [8], [11]. Advances in the energy efficiency of compute and memory will only increase the proportion of system energy consumed by the interconnect.

There are clear opportunities for interconnect energy savings, due to the network behavior of HPC applications. While HPC applications require a high-performance interconnect, to support their peak communications demand, their average utilization of the network is low. Several proposals attempt to exploit this opportunity to save energy [7], [8], [17], [20], [26]. These proposals are built upon one of the following underlying mechanisms. The first approach is on/off-based networks, where the links are powered down during idle periods. An important example of this approach is IEEE 802.3az Energy Efficient Ethernet (EEE) [4], [10]. Approved in 2010, EEE was primarily designed to save network power consumption in homes, offices and data centres. Alternatively, *bandwidth-tunable links* adapt the network bandwidth to the communication requirements, reducing the frequency or number of channels when demand is low, thereby reducing the power consumption. InfiniBand is an important example, which implements variable bandwidth as well as varying the number of active $1\times$ links [7].

Figure 1 presents the system share of interconnects in the Top500 list [3] over the past five years, specifically from June 2011 to June 2016. It is clear that Ethernet and InfiniBand dominate the TOP500 list, each with about 40% share. It is also interesting to note that dip and recovery of Ethernet (1G+10G) seen between June 2014 and June 2016 comes from the falling in the number of machines using 1G Ethernet and the rapid growth in the adoption of 10G. A similar rate of adoption can be expected from the upcoming 40/100 Gb/s Ethernet standards.

Although EEE is built into the standards and therefore available for use in HPC, it is likely to be disabled by default until its performance overheads and energy savings are well understood. Effective use of EEE-based power saving mechanisms is especially important in HPC since, while saving power, these techniques also introduce an increase in network latency [9], which could lead to an unacceptable increase in execution time. Although energy efficiency is increasingly important in HPC, the primary design objective is still performance, so energy saving mechanisms are unlikely to be adopted unless their overheads are controlled.

The Energy Efficient Ethernet protocol was proposed in 2008 for 10 Gb/s and lower Ethernet speeds with an on/off-based power saving mode known as Deep-Sleep. The current Ethernet task force in charge of standardizing 40 and 100 Gb/s Ethernet is now adopting EEE alongside an additional sleep state, known as Fast-Wake, trading off

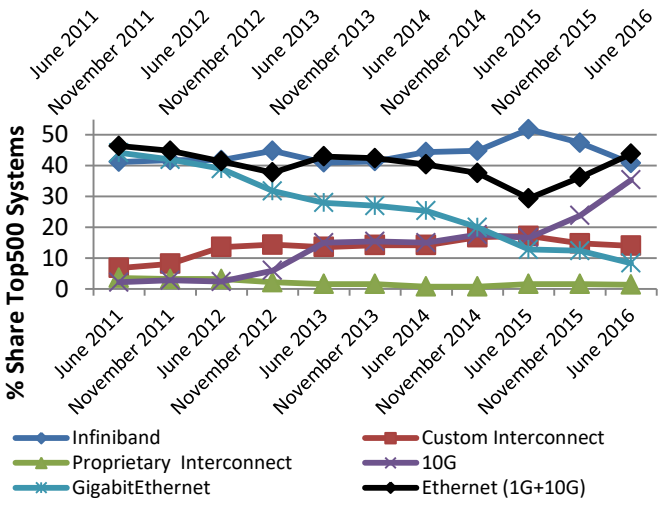


Fig. 1: **Top500 Interconnect System Share [3]**

higher energy consumption for a faster wake-up time.

This paper presents PerfBound[1] and its extension DynamicFastwake, both on/off control mechanisms for network links that adjust link parameters for energy savings while bounding the performance overheads to acceptable levels. PerfBound works to control link on/off states such that its effect on performance is controlled while saving energy. PerfBound has been shown to work well using the original sleep mode of EEE, Deep-Sleep. Recent analysis [13], on the other hand, shows that a combination of both Deep-Sleep and the newly introduced Fast-Wake, provide an opportunity for larger energy savings compared to Deep-Sleep alone. The analysis [13] shows that higher energy savings are only possible through a careful configuration of Fast-Wake and Deep-Sleep parameters. Furthermore, it finds that incorrect configurations could lead to both lower energy savings and higher performance overheads. In light of this, this paper shows how simple modifications to the original PerfBound [12] mechanism could allow for controlling both Deep-Sleep and Fast-Wake. The results presented show that DynamicFastwake consumes 10% lower energy at the same performance, compared to PerfBound using a single sleep mode, Deep-Sleep.

The mechanisms, PerfBound and DynamicFastwake, discussed in this paper have the advantage of being self-contained and application agnostic, in that the application is not modified and all decisions are taken locally at the links, without any additional communication between nodes and/or switches. The only parameter required is an acceptable overhead of performance. In this paper it is set to 1%, but it should be clear how this can be changed for lower target overheads. Furthermore, the ideas presented are applicable to non-Ethernet based systems that rely on an on/off-based system for link energy proportionality.

In summary, in order for HPC to adapt energy-proportional interconnects, it is crucial that any resulting performance overheads are controlled. In this regard, this paper discusses EEE on HPC, PerfBound and DynamicFastwake, relating these proposals for link energy savings in the context of controlled performance degradation. Enabling EEE with controlled performance overheads could directly improve energy savings of the large number of machines in the Top500 list that use Ethernet, as shown in Figure 1.

1. PerfBound was originally published at the International Supercomputing Conference 2014 [12]

In focusing on EEE, the results and ideas proposed in this paper could benefit vendors and the Ethernet task-force in designing for HPC. Although this work does not directly discuss InfiniBand, the results should also be applicable to projecting energy savings and controlling performance overheads of bandwidth-tunable links.

## 2 Background

### 2.1 Energy Efficient Ethernet (EEE) with Deep-Sleep

The Internet and specifically data centers accounted for 1.1% to 1.5% of global energy consumption in the year 2010, numbers that have doubled since 2005 [5]. The increasing energy cost of Internet infrastructure and data centers encouraged a push for energy efficiency across the computing spectrum.

To address the above, the IEEE 802.3az Energy Efficient Ethernet (EEE) Task Force was established, which subsequently in the year 2010, published the standard for Ethernet energy efficiency [4], [10]. The goal of the standard was to reduce the contribution of network devices to the national power budget, especially since large sections of the Internet and data center infrastructure are built using Ethernet [10]. After considering various proposals, including adaptively changing the link rate and bandwidth, the task force adopted the proposal known as Low Power Idle (LPI) [4], [10].

Low Power Idle, which is now known as **Deep-Sleep**, is an extension to Ethernet that allows the link to switch between "sleep" and "wake" modes on demand, in order to save energy. Deep-Sleep (or LPI) was considered straightforward to implement, because it freezes the state of the transceiver whenever the link enters sleep and it restores the state when it wakes [4]. In Deep-Sleep mode, the link is periodically refreshed and is not completely switched off. Arrival of a frame at the transmitter side of a link triggers the signaling of a wake-up transition to return to active mode. Frame transmission can start once the transceiver and receiver PHYs are both active. At the next hop, any arriving frames have to be buffered whenever the next link is in sleep or in the process of waking up. The IEEE 802.3az standard was published for 10Gb Ethernet, and the wake-up and sleep timings specified are $4.48\,\mu s$ and $2.88\,\mu s$ respectively, with about 90% energy savings at low power mode compared with active [1].

### 2.2 Energy Efficient Ethernet (EEE) with Fast-Wake

Subsequent standardization efforts, which focused on the specifications for 40Gb, 100Gb and 400Gb backplane and optical Ethernet, adopted the energy savings mechanisms from EEE. In this process, the IEEE 802.3bj task force in charge of 100Gb Backplane and copper cable standardization introduced an additional sleep state known as **Fast-Wake** [2]. In comparison with Deep-Sleep, Fast-Wake has a higher energy consumption and a faster wake-up time.

The motivation for the Fast-Wake mode came from the relatively high wake-up time of Deep-Sleep, whose effect on performance would be more pronounced at higher link speeds. The possible increase in latency due to Deep-Sleep was considered to be too high, so Deep-Sleep was expected to be disabled for 100Gb links. The long wake-up delay

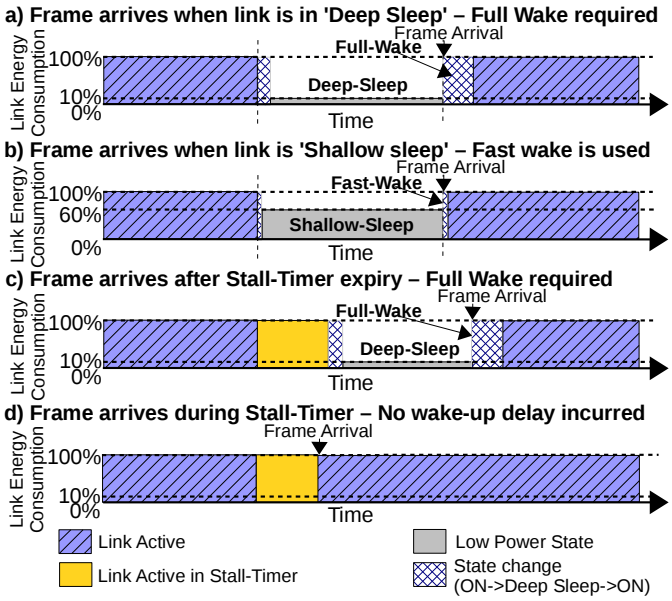


Fig. 2: **Example timeline illustrating various low power and Stall-Timer states of an EEE link**

in the original EEE standard comes from signaling components, specifically the Physical Medium Attachment (PMA) and Physical Medium Dependent (PMD) components in the PHY, that are required to synchronize the transmitting and receiving links before data transmission. In the case of Fast-Wake, while some components are powered down, the PMA and PMD remain active, continually transmitting signals between transceiver and receiver, thereby maintaining synchronization. This allows for a link wake-up in a few hundred nanoseconds, rather than in microseconds. Specifically, studies show that the link could wake from Fast-Wake mode in 250 ns with energy savings of about 40%, compared to an active link [14].[2]

### 2.3 Energy Efficient Ethernet in HPC

While EEE provides specifications for turning links on and off, the mechanisms to decide when to do so are left to the vendor. Early evaluations proposed the use of Frame Buffering, also known as Packet coalescing [11], which buffers frames that arrive while the link is in sleep. The link is woken after a certain number of frames arrive or a time-out expires, in an attempt to amortise the wake-up energy over multiple frames. With appropriate frame limits and time-outs, the technique was shown to bring links closer to energy proportionality. This mechanism was, however, proposed for Internet and data-center workloads that could tolerate significant increases in transmission latency caused by buffering frames [10], [11].

HPC applications are characterized by long compute periods (where the network is idle) fragmented by short bursts of communication. These short bursts however, require peak bandwidth and are generally latency sensitive. Buffering frames for 100 µs, as recommended for Internet and data-center workloads [10], [11], can cause high and unacceptable performance overheads in HPC. For this reason, an effective mechanism for HPC has been the **Stall-Timer**,[3] which, instead of buffering frames, leaves the link on (while idle) for a defined period after every frame transmission. This ensures that subsequent frames can be transmitted without incurring any wake-up delay.

Figure 2 illustrates the working of the above described mechanisms using an EEE-based on/off link. In Figure 2(a), the link remains active during frame transfer, which is followed by a state change that turns off the link, reducing power consumption to 10%. The later frame arrival during Deep-Sleep requires a **Full-Wake** (4.48 µs) to activate the link before transmission. Figure 2(b) is similar, except that the link powers down to **Shallow-Sleep**, which has a higher power consumption of 60%. In return, the link powers back to active mode much faster, doing so in a few hundred nanoseconds. Figure 2(c) and (d) show the working of Stall-Timers. In both cases, the link remains on at 100% after frame transmission, as opposed to Figure 2(a) where the link initiates the state change to Deep-Sleep immediately. In Figure 2(d), the frame that arrives before the Stall-Timer expires can be transmitted immediately, with no wake-up delay. Further details of Stall-Timer and the relationship between link energy and performance are discussed in Section 4.

A large share of TOP500 systems (about 40%) use 1Gb or 10Gb Ethernet. This popularity of Ethernet in HPC, coupled with the need for energy proportionality, makes a strong case for using the devices' inherent power-saving techniques. Products supporting Fast-Wake may be deployed in HPC systems within a year,[4] but without further investigation, Fast-Wake will likely be disabled by default, since, although energy efficiency is important, performance is the primary design objective for HPC. It is important, therefore, to design and employ control algorithms that maintain a low performance overhead. To this end, the insights presented in this paper could help vendors in designing EEE technology for HPC.

## 3 Methodology

The experiments and analysis in this paper were done using the Dimemas cluster simulator, which has been found to be accurate to within 10% and validated against production supercomputers, including Blue Gene/L, P, Q, and three generations of the MareNostrum supercomputer [28], [29], [30]. The network model was modified to support a hierarchical network, with on/off links as specified by the EEE standard. The simulation infrastructure is driven by traces, which record CPU intervals and MPI events, measured from a real execution on the MareNostrum II supercomputer. The CPU intervals are scaled by relative CPU performance. Link energy consumption is modeled as

2. Recent specifications show that Deep-Sleep may not yet be compatible with Optical Transport Network (OTN) based optical networks, but Deep-Sleep is expected to be incorporated into these devices in the future.

3. The Stall-Timer essentially *stalls* the link from going to sleep immediately after a frame transfer.

4. The standards supporting Fast-Wake are IEEE 802.3bj and 802.3bm, for 40Gb/100Gb backplanes and optical Ethernet, respectively, ratified as recently as March 2015, and IEEE 802.3bs for 400Gb, which is expected to be ratified in 2017. Switches that support EEE, targeting data centers, were commercially available within a year from the date of standardization.

TABLE 1: HPC WORKLOADS USED IN THIS STUDY

| Name | Num. nodes | Class | Description |
|---|---|---|---|
| ALYA [31] | 256 | Biology | Biomechanics |
| BT, CG, MG [32] | 256 | Fluid Dynamics | NAS Parallel Benchmarks |
| SP, LU [32] | 64 | Fluid Dynamics | NAS Parallel Benchmarks |
| CPMD [33] | 128 | Chemistry | Molecular Dynamics |
| GADGET [34] | 128 | Astrophysics | Dark-matter simulation |
| GROMACS [35] | 128 | Biology | Biomolecular dynamics |
| LINPACK [36] | 256 | Benchmark | Linear algebra solver |
| MILC [37] | 128 | Physics | Subatomic Interactions |
| NAMD [38] | 64 | Biology | Biomolecular simulations |
| PEPC [39] | 64 | Mathematical | Parallel Coulomb Solver |
| QUANTUM [40] | 128 | Chemistry | Nanomaterials modeling |
| WRF [41] | 128 | Meteorology | Weather Forecasting Model |

100% when "on" or during transition between on/off states, 60% in Fast-Wake mode and 10% in Deep-Sleep mode.

The simulator is configured to model a cluster with a three-level hierarchical network. Applications are executed on 64, 128 or 256 nodes, grouped into 8, 16, or 32 nodes per rack, respectively, forming eight racks in total. Nodes are connected to the top-of-rack switch (level 0), which is in-turn connected to a two-level fat tree (4-ary 2-tree). The architecture of the network and the link bandwidths are shown in Figure 3. The network is statically routed with cut-through flow-control and full-duplex links (each direction of which can be turned on and off separately). The routing decision uses a static choice of shortest path between source and destination (if there are multiple equal-cost paths, the route is chosen based on the destination address).

The system and network parameters were chosen to emulate a high-end HPC system based on analysis of systems in the TOP500 list. Each node is a two-socket high-end CPU with 225 GF (based on June 2012 TOP500 list machines with two Intel Xeon sockets). The switch latency is configured at 320 ns for the first hop and 80 ns for subsequent hops to emulate about 1 µs in end-to-end worst-case network latency. Edge links are configured at 20 Gb/s, and the higher two levels are 40 Gb/s and 100 Gb/s respectively. Wake-up and sleep timers for low power states were obtained from Energy Efficient Ethernet specifications and were set to 4.48 µs and 2.88 µs for Deep-Sleep [4], [10] and 250 ns for Fast-Wake [14] respectively. Fourteen HPC applications (shown in Table 1) were used for analysis in this paper. The original traces were large, on the order of hundreds of gigabytes, so simulation was done for a few iterations of the outer loop. The original traces were obtained from the applications executed over as many nodes as possible depending on system availability.

## 4 MOTIVATION

DynamicFastwake automatically adjusts two Stall-Timers subject to a bound on the performance overhead. Before discussing DynamicFastwake and how PerfBound was modified to control two Stall-Timers, this section discusses the motivation behind this work.

### 4.1 Performance overheads and wake-up delays

Although the EEE standard defines mechanisms for entering and leaving the sleep mode (Low Power Idle), it does not define how to decide when to do so. A naive, and aggressive, technique is to always turn the link off as soon as it becomes idle and to turn it back on only on demand. There is, however, a trade-off between energy savings and performance overhead: aggressive techniques, such as the above save more energy but may introduce too much network latency, whereas conservative techniques incur a low performance overhead but they achieve little energy savings.

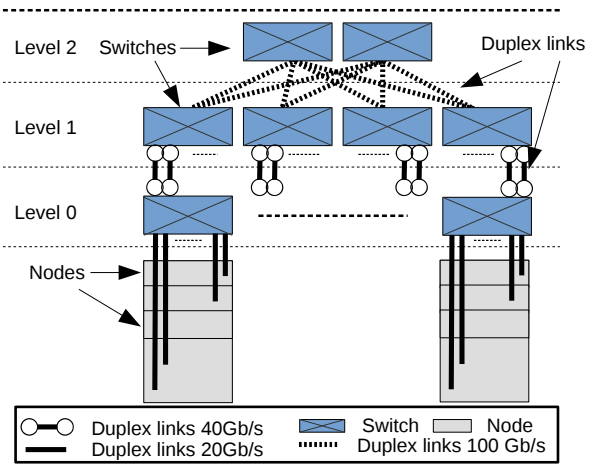


Fig. 3: **Network organization used in the discussion and analysis of PerfBound, Fast-Wake and Double PerfBound**

One difficulty in HPC is that different applications react differently to increases in latency. Figure 4 shows a sensitivity analysis of application performance to wake-up latency, assuming the naive management technique. The $x$-axis is the wake-up latency (which for EEE is 2.88 µs/link). The least latency-sensitive applications, including Quantum and BT, can potentially tolerate an aggressive energy saving technique, since the naive approach incurs only about 2% performance overhead. In contrast, GROMACS and NAMD are seen in the figure to have unacceptable performance degradation, with their execution time roughly doubled, so they require a rather conservative energy saving scheme.

There are two questions related to the management of on/off links: when to turn the link off, and when to turn it back on. An ideal solution, which obtains maximum energy savings, would turn the link off immediately after each message and turn it back on at the correct time in anticipation of the next message (if the idle period is shorter than the sum of the sleep and wake times, then the link is, of course, not switched off). This scheme, however, requires an accurate and precise prediction of the arrival time of the next message. If the prediction is wrong, then, either the link is woken up too late, incurring a performance overhead, or too soon, wasting potential energy savings.

A simple mechanism that can work well is to turn the link off only after a specific duration of idle time, which we call the **Stall-Timer**, and to turn it back on when the next message arrives. The naive approach described above corresponds to Stall-Timer=0. Our previous study [9] found that this mechanism can work well in HPC. Since different applications have different sensitivities to increases in latencies, the optimal value of the Stall-Timer depends on the application. PerfBound, discussed in Section 5, automatically determines the correct Stall-Timer to obtain high energy

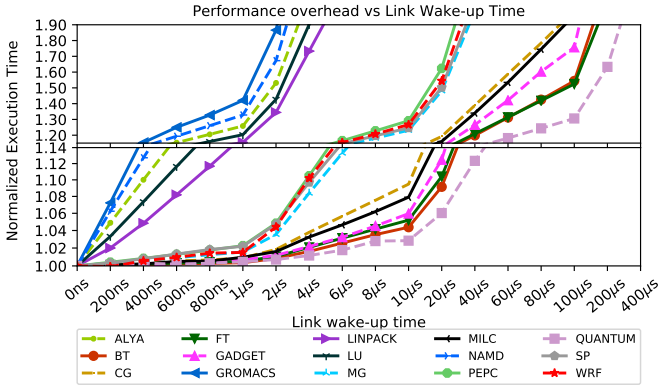


Fig. 4: **Application performance overhead as a function of wake-up delay.**

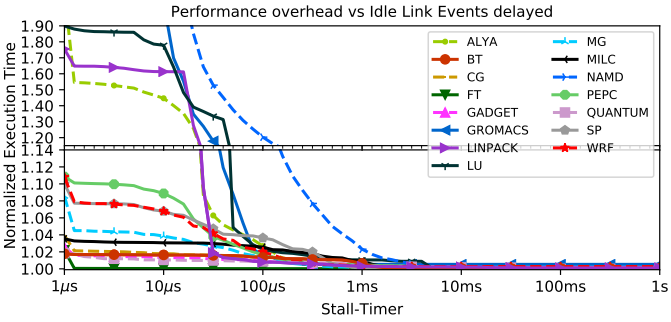


Fig. 5: **Application performance overhead as Stall-Timer is varied - normalised to execution over an always on network.**

savings subject to a target performance bound. Section 6, proposes DynamicFastwake, which extends PerfBound to control the multiple Stall-Timers.

The first key insight, in the development of PerfBound, is that, since every time the link is switched off one message will later be delayed by the wake-up time, the performance overhead is approximately proportional to the number of times that the link is switched off. This is an approximation, since the method cannot track chains of dependencies among nodes. Tracking dependency chains requires either that the user or compiler annotates the application, or that additional messages are sent by the run-time system and monitored by the switches. Either approach adds complexity, with the result that such a proposal would be unlikely to be adopted in practice. On balance, as the results show, the approximation is generally sound, and the PerfBound approach gives the right compromise. The performance overhead bound translates to a fixed number of messages, per unit time, that can be delayed. The following heuristics ensure that this number of delayed messages is not exceeded, and that the right choice of messages to delay is made, to get the high energy savings.

### 4.2 Understanding the overhead of link wake-up and idle time predictability

In order to make overhead-aware decisions for link energy savings, it is important to first understand how wakeup latencies translate to performance overheads. Figure 4 showed how different applications have different sensitivities to wake-up latencies. To look at this question in more detail, we evaluated the sensitivity of application overheads to wake-up delays. From this point in this paper, we refer to "message inter-arrival periods" as **idle link events**.[5]

Figure 5 shows a sensitivity analysis plot relating the Stall-Timer value, on the $x$-axis, to application performance. The Stall-Timer causes wake-up delays to only be incurred by messages that arrive after the Stall-Timer expires, during which the link is idle. Hence, if the Stall-Timer value is zero, then all messages would be delayed by the wake-up latency whereas if the Stall-Timer value is infinite, no messages would be delayed. At low values, as seen in Figure 5, nearly all messages have a wake-up overhead, causing large increases in the application execution time. However, as the Stall-Timer value is increased, the number of messages that are delayed reduces, and the overheads are seen to drop down to zero. This essentially makes the case for a Stall-Timer, which causes only messages that arrive after idle durations larger than itself to be delayed, for use in controlling performance overheads. Figure 5 therefore shows how a relationship between overheads and Stall-Timer can be established. Hence, if an acceptable level of performance overhead for an application is 5% (say), then Figure 5 could be used to determine an application-dependent static value for a Stall-Timer. For application LU, for instance, we can see that an appropriate value of the Stall-Timer would be $50\,\mu s$. In this case, the link remains on for the first $50\,\mu s$ in each idle period, saving power on all idle link events that are longer than $50\,\mu s$, but maintaining performance overhead inside the specified bound of 5%.

The HPC application behavior can be understood in greater detail, from the perspective of idle link events, by looking at the heatmaps in Figure 6. All sub-figures show the length of the current idle link event on the $x$-axis and the length of the next idle link event on the $y$-axis, for LINPACK and BT according to the title. Figures 6(a) and 6(c) are colored according to the *number of events*, whereas Figures 6(b) and 6(d) are colored according to the total idle time contributed by those events. That is, if in Figure 6(a) there are 100 events in position ($2\,ms$, $2\,ms$), then their total idle time would be $200\,ms$. The idle link event heat-map gives a sense of the most common idle durations and the total idle time is helpful in understanding how the idle time translates to energy savings. The results present averages across all edge links in the network.

LINPACK and BT are both typical examples of HPC applications, and the difference between Figures 6(a) and 6(c) is in the clustering of idle link events. In the case of LINPACK, in Figure 6(a), the events are clustered at around $10\,\mu s$, while the events in BT are clustered at around $1\,ms$. Another key difference between these applications is clear from Figures 6(b) and 6(d): the majority of the idle link events of LINPACK are of $10\,\mu s$, but most of its total idle time comes from events that are longer than $10\,ms$, even though there are few of them. A similar behavior can be seen in BT, where a small number of events longer than $10\,ms$ also contribute to a significant amount of total idle time. The main difference for BT is that its most common idle link event duration also contributes significantly to the

5. In this paper, we term any duration during which no data is transmitted over a link as an *idle link event*.

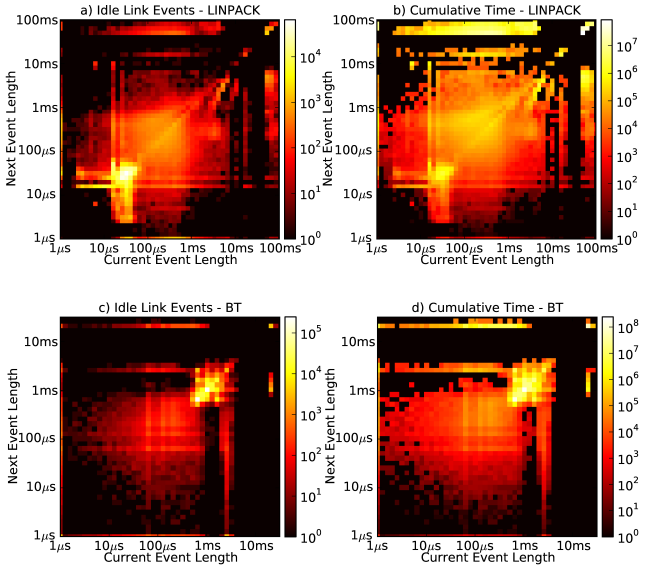


Fig. 6: **Idle Link Event distributions of LINPACK (a),(b) and BT (c),(d). The heat maps show the number of events (a),(c) and the total idle time (b),(d).**

total idle time.

Comparing Figures 5 and 6, we can explain the observed performance overheads. Firstly, note from Figure 5, that the performance overhead of LINPACK remains at about 60% until the Stall-Timer is increased to 10 µs, and it drops to about 2% between 10 µs and 100 µs. Comparing that to Figure 6(a), the performance overhead has clearly dropped as the Stall-Timer crossed the cluster between 10 µs and 100 µs. In other words, if the link remains on for about 100 µs, none of the events in the cluster in Figure 6(a) would incur performance overheads. Similarly, in the case of BT, comparing Figures 5 and 6(c), we can see that the performance overhead of BT, starting from 2%, drops to near zero at about 1 ms; this correlates to the clustering found at 1 ms in Figure 6(c). BT has low performance overhead, even at low values of the Stall-Timer, because, first, at low threshold values in Figure 6(c), no events exist to incur performance delays. Since, for BT, the number of events that exist between 1 µs and 1 ms is low, the reduction in the performance overhead is gradual. Secondly, for large events, the ratio of event size to delay incurred is very low. To illustrate, when a delay of 1 µs is applied to an event of 1 ms, the added delay corresponds to 0.1%. For LINPACK, most events are clustered at 10 µs, so if a 1 µs delay is added to them, each delay adds 10% of the idle time, translating to large performance overheads.

The insights presented are important in the discussion of PerfBound and its extension DynamicFastwake. The following section discusses the primary motivations behind the need for DynamicFastwake with an analysis comparison of Fast-Wake and Deep-Sleep.

### 4.3 Stall-Timer for Deep-Sleep and Fast-Wake

Figure 7 illustrates the trade-off between execution time and energy for Deep-Sleep alone in Figures 7 (A)(i) and (ii) and Fast-Wake alone in Figures 7 (B)(i) and (ii). For all four figures, the $x$-axis is a sweep of Stall-Timer values.

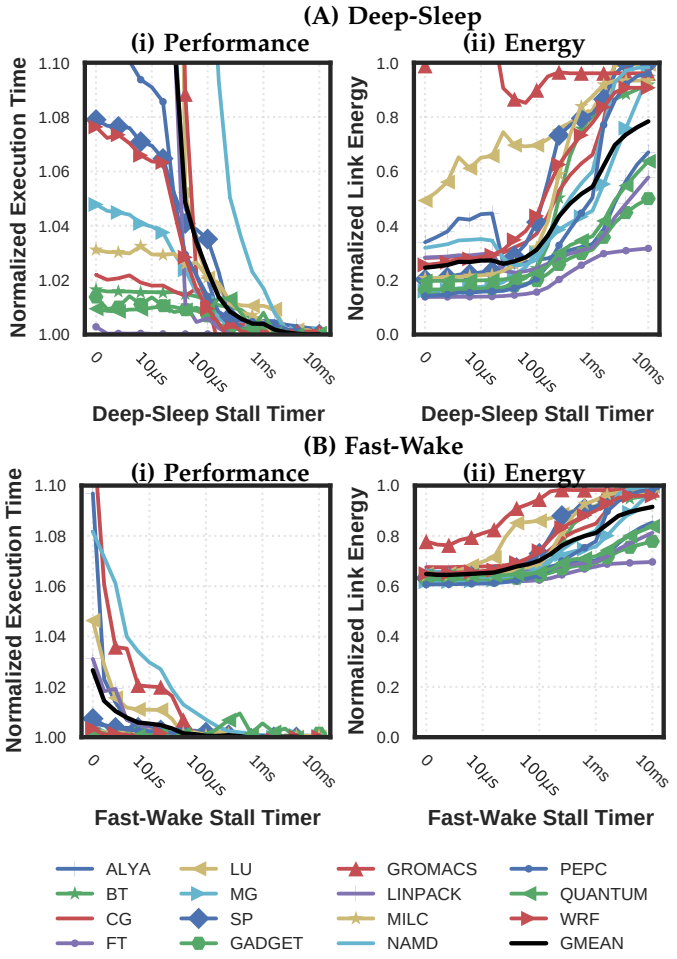


Fig. 7: **Energy and performance as a function of Stall-Timer**

In Figures 7 (A)(i) and (B)(i), the $y$-axis is the execution time, and for Figures 7 (A)(ii) and (B)(ii), the $y$-axis is the link energy. The figures are all normalized for each application referenced in Section 3, relative to an always-on interconnect. Furthermore, the energy consumption is the average across all edge links of the network, but the below conclusions apply similarly for higher-level links.

With Figure 7, a comparison between Deep-Sleep and Fast-Wake over HPC applications is presented. As discussed previously, increasing the Stall-Timer value decreases the execution time but it increases link energy.[6,7] In Figure 7, it is interesting to note that some applications benefit more from either using Fast-Wake alone or Deep-Sleep alone, i.e., they have lower performance overhead and higher energy savings with only either one of them. For example, for

6. Note that in Deep-Sleep and Fast-Wake, the links consume at least 10% and 60% energy respectively, as shown in Figures 7 (A)(ii) and (B)(ii), and cannot save energy beyond that point.

7. In Figure 7 (A)(ii), a dip in the energy curve for GROMACS between the Stall-Timer values 50 µs and 500 µs can be observed. This is because at small values of Stall-Timers (less than 50 µs) excessive number of sleepwake transitions, which in addition to increasing the execution time increases the energy consumption. Hence, curve between 50 µs and 500 µs represents optimal points between higher energy consumption due to increased execution time and lower energy savings due to a large Stall-Timer.

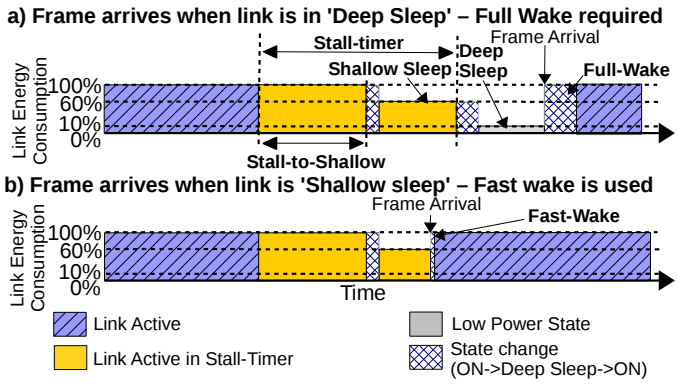


Fig. 8: **Timeline illustrating hybrid Fast-Wake+Deep-Sleep**

a performance overhead of 1%, NAMD with Deep-Sleep consumes 85% energy, while with Fast-Wake, at the same overhead, it consumes only 65% energy. Clearly, NAMD benefits from using Fast-Wake alone. In contrast, PEPC at the same 1% overhead consumes only 20% with Deep-Sleep, while it consumes 60% with Fast-Wake. These examples, along with GADGET, FT and QUANTUM, tend to have better energy savings with Deep-Sleep while others such as GROMACS, SP and MILC benefit more from Fast-Wake. This analysis clearly makes a case for using an approach that combines Deep-Sleep and Fast-Wake.

### 4.4 Stall-Timer for hybrid Fast-Wake + Deep-Sleep

Figure 8 shows the timeline of a link with hybrid Deep-Sleep and Fast-Wake. In Figure 8(a), the link is initially active and transmitting data, after which the link remains on, consuming full (or 100%) power, until the Stall-to-Shallow timer expires. When this happens, the link switches to Shallow-Sleep, during which it consumes 60% energy. The link remains in Shallow-Sleep state until the original Stall-Timer expires, at which time it drops to Deep-Sleep, consuming 10% power. The arrival of a frame during Deep-Sleep triggers a full wake-up, before the link can transmit the frame. Figure 8(b), shows the case where the next frame arrives during the Shallow-Sleep period, so that only a Fast-Wake is needed to quickly power up the link.

Figure 9 shows the energy–execution time trade-off for the hybrid scheme. The $x$-axis is execution time and the $y$-axis is average link energy (both normalized to non-EEE). The blue line is for Deep-Sleep only, and is a one-dimensional sweep of varying values of a single Stall-Timer from small to large, as seen in Figure 7. The scatter plot is an exhaustive search for the hybrid fast-wake + Deep-Sleep scheme, covering the two-dimensional space given by the two Stall-Timers. The dark line connects the Pareto-optimal points from the scatter plot, which are roughly the points with lowest energy consumption for a given performance overhead.

Figure 9 is used to motivate the need for combining Fast-Wake and Deep-Sleep using a sweep over the two-dimensional space of Stall-Timer values. The following sections describe how PerfBound and DynamicFastwake automatically find the right values, and the evaluation of these techniques is presented in Section 7. All applications evaluated in this work show enormous potential for link energy savings compared to the traditional always-on network to which these results are normalized. Most applications (as presented in Section 7) are similar to SP and ALYA shown in Figure 9, saving between 50% to 80% link energy at an overhead of 1%. This makes a strong case for the use of these on/off schemes in HPC environments.

Comparison of the hybrid scheme to Deep-Sleep only, is as follows. First, SP has a clear gap between Deep-Sleep only and the hybrid Pareto-optimal curve, so that, for example at an overhead of 1%, there are potential savings of 20%, assuming that the appropriate Stall-Timer values can be identified. Similar to SP, GROMACS is another example of an application that benefits from the hybrid scheme with 12% better energy savings compared to Deep-Sleep. ALYA on the other hand has about 5% improvement over Deep-Sleep, but in particular emphasizes the need to carefully choose the Stall-Timer values. In application SP, it can be seen that a bad combination of Stall-Timers would result in sub-optimal energy–performance, but it would still be at least as good as Deep-Sleep. For ALYA, in contrast, there are many combinations of Stall-Timer values that are significantly worse, in both energy and performance, than Deep-Sleep. Furthermore, for the same overhead, 1% say, there are multiple pairs of Stall-Timers with large differences in energy consumption among themselves. A bad choice of Stall-Timers could result in up to 20% higher energy consumption compared to the Pareto-optimal curve, and 15% worse than Deep-Sleep.

In summary, while significant energy savings are available from using Deep-Sleep and Fast-Wake, Stall-Timers must be carefully chosen. Their optimal values are application-dependent, with potential values in a two-dimensional space that includes points that are far from being Pareto optimal. Moreover, for a given performance overhead there are many choices of Stall-Timer pairs, each consuming different amounts of energy. Hence, finding the Pareto-optimal solution can only be done based on an estimate of the energy consumption. The following sections show how DynamicFastwake automatically finds optimal Stall-Timer pairs for a given performance overhead bound to minimize energy consumption.

## 5 PerfBound: Bounding performance overheads in on/off HPC links

Section 4 provided several key insights, which we use as the basis for the design of PerfBound. Firstly, the application overhead is roughly proportional to the number of delayed idle link events. Secondly, the application overhead can be adjusted using the Stall-Timer. Thirdly, the best Stall-Timer depends on the application, so the algorithm itself must be dynamic, adaptive and application independent. Finally, from an energy standpoint, it is best to delay the events of longest duration.

Based on the above, this section introduces PerfBound and PerfBoundRatio. The only parameter to the algorithms is a limit on the acceptable performance degradation. For the purpose of the exposition, we assume in the following discussion that the limit is 1%, but it should be clear how to make the bound into a parameter.

The overall approach is to first determine how many idle link events can be delayed per unit time before the overhead reaches 1%, and then to ensure that the right

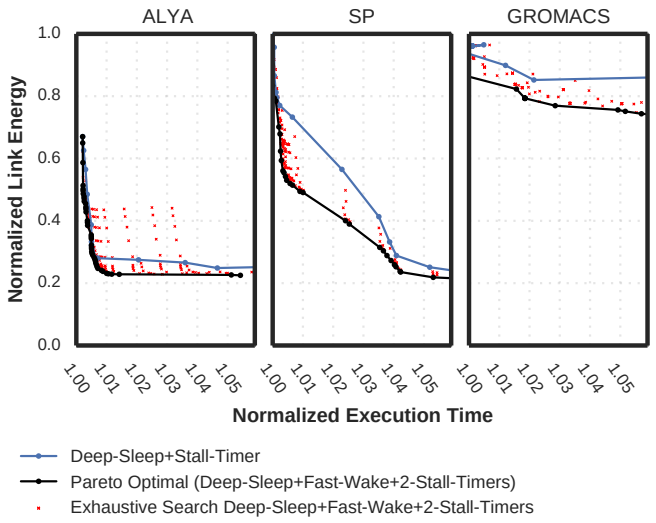


Fig. 9: **Pareto-optimal analysis of performance/energy using both Deep-Sleep and Fast-Wake vs. only using Deep-Sleep with Stall-Timers**

number of events are delayed and that they are the longest ones. This is done by maintaining a histogram of the lengths of the idle periods, and using it to dynamically adjust the Stall-Timer value. This approach maintains the performance bound without requiring any changes to the application.

### 5.1 Calculating the #events that can be delayed

We first analyze the case where there is a single hop between two nodes. Since the overhead is assumed to come only from delayed wakeup events, the maximum number of them that can be tolerated, within a 1% bound, in a period of length $X$ is simply $0.01X/T_w$, where $T_w$ is the wakeup delay and $0.01$ corresponds to the 1% bound. As $X$ increases, the total number of events that can be delayed also increases, in proportion. This is the value used by **PerfBound**, when configured with a local performance bound of 1%. The next section will describe how the Stall-Timer is adjusted to delay the correct number of events.

In a multi-hop network, each link in the route may implement PerfBound, each contributing to the resulting performance overhead. A three-level network has longest routes containing three upward links and three downward links, for a maximum hop count of six, so a single message may incur six cumulative wakeup delays. Using the above equation directly leads to a total overhead of up to 6%.

The simplest solution is to divide the global 1% performance bound equally among the links, so that each link uses PerfBound with a local performance bound of 0.166%. This is, however, unnecessarily conservative. An application that mainly communicates at Level 0 (say), would rarely incur overheads at the upper levels, meaning that the overhead is actually being constrained to 0.33%. Although a lower overhead is better, all else being equal, the configured 1% performance bound would probably have led to greater energy savings.

**PerfBoundRatio** is the solution to multi-hop networks, which is configured with a global performance bound. It adapts dynamically to the locality of the application's communication pattern, using only the information that is

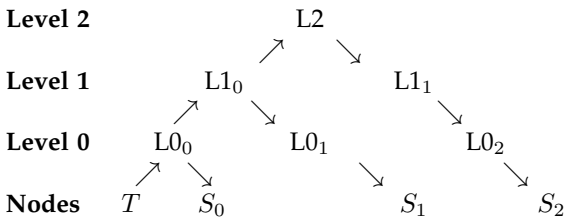


Fig. 10: **Example network topology**

TABLE 2: PERFBOUNDRATIO: EXAMPLE CALCULATION OF LOCAL STATE, WHEN 50%, 40% AND 10% OF MESSAGES REACH LEVELS 0, 1 AND 2, RESPECTIVELY.

| Link | Total messages | Messages/level | | | Proportion to level | | |
|---|---|---|---|---|---|---|---|
| | | L0 | L1 | L2 | L0 | L1 | L2 |
| $T$ to L0$_0$ | 1000 | 500 | 400 | 100 | 0.5 | 0.4 | 0.1 |
| L0$_0$ to $S_0$ | 500 | 500 | 0 | 0 | 1.0 | 0 | 0 |
| L0$_0$ to L1$_0$ | 500 | 0 | 400 | 100 | 0 | 0.8 | 0.2 |
| L1$_0$ to L0$_1$ | 400 | 0 | 400 | 0 | 0 | 1.0 | 0 |
| L0$_1$ to $S_1$ | 400 | 0 | 400 | 0 | 0 | 1.0 | 0 |
| L1$_0$ to L2 | 100 | 0 | 0 | 100 | 0 | 0 | 1.0 |
| L2 to L1$_1$ | 100 | 0 | 0 | 100 | 0 | 0 | 1.0 |
| L1$_1$ to L0$_2$ | 100 | 0 | 0 | 100 | 0 | 0 | 1.0 |
| L0$_2$ to $S_2$ | 100 | 0 | 0 | 100 | 0 | 0 | 1.0 |

available locally at the switch. In order to use PerfBound, each switch must be given enough information about the network topology to be able to calculate the level of the highest switch in the route between any pair of source and destination IP addresses. This may require specific configuration, but for an HPC system, such configuration is tolerable.

We explain PerfBoundRatio using the example three-level network in Figure 10. The switches are labeled with the level and a unique number; e.g. L1$_1$ is one of the switches in level 1 and nodes are labeled T and $S_0$ to $S_2$. Let us assume, node $T$ transmits 1000 messages in total, to $S_0$, $S_1$ and $S_2$, in proportion 50%, 40% and 10%, respectively. In a real application, all nodes will transmit, with different distributions to various nodes, but the total counts are simply the sums of the various contributions, and the algorithm still works. It can be best understood by looking at a simple case.

Each link has four counters, one that counts the total number of messages over the link, and three messages/level counters, each corresponding to a level in the network. The messages/level counter for level $n$ counts the number of messages seen whose highest level in the network is exactly $n$. This information is summarized, for all links, in Table 2. This table also shows the proportion of messages that go to each level, found by dividing by the total number of messages. For example, the link between L0$_0$ and L1$_0$ sees all messages from $T$ that go to either $S_1$ or $S_2$. There are 500 such messages, of which 400 go to $S_1$, reaching level 1, and 100 go to $S_2$, reaching level 2. The ratios of messages that reach levels 0, 1 and 2, respectively, are 0, 0.8 and 0.2.

The key idea is to divide the global performance bound according to the behavior of the communication traffic. Of the 500 messages that are seen over the link between L0$_0$ and L1$_0$, 80% of the messages that reach network level 1, have four hops on their route, whereas the 20% of messages that

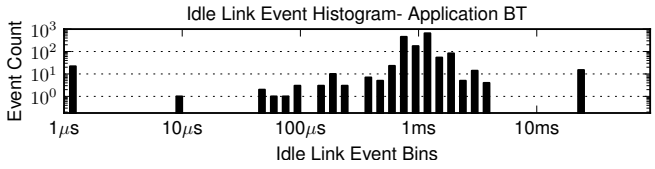


Fig. 11: **Snapshot of an Idle Link Event Histogram**

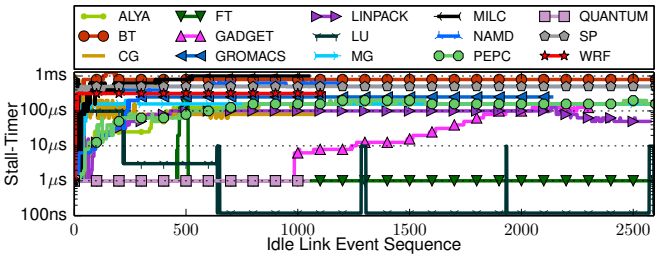


Fig. 12: **Stall-Timer convergence over time**

reach network level 2, have six hops. The local performance bound is therefore given by $0.8 \times \frac{0.01}{4} + 0.2 \times \frac{0.01}{6}$. In this equation, the weighing factors of $0.8$ and $0.2$ are given by the message statistics, $0.01$ is the global performance bound, and the denominators are the numbers of hops on the routes.

In general, for a particular link, let $MC0$ be the total number of messages that reach maximum level 0, $MC1$ be the total number that reach maximum level 1, and $MC2$ the total number that reach level 2. Let $MC = MC0 + MC1 + MC2$ be the local total message counter. Then the local performance bound (as shown in Table 2) for that link is given by

$$l = \frac{MC0}{MC} \cdot \frac{0.01}{2} + \frac{MC1}{MC} \cdot \frac{0.01}{4} + \frac{MC2}{MC} \cdot \frac{0.01}{6}$$

### 5.2 Calculating the Stall-Timer Value

After calculating the total number of events that can be delayed, per unit time, the final step is to determine the Stall-Timer. As described in Section 3.2, the Stall-Timer is the duration of time that the link must remain idle before it is switched off.

The Stall-Timer is determined from a histogram of idle link events. In detail, at the end of every idle link event, one new data point is available. This data point is the length of the previous idle link event. As shown in Figure 11, the bin corresponding to this length is determined and its histogram value is incremented. The histogram therefore keeps track of the distribution of link idle interval lengths, and its total mass increases over time.

The Stall-Timer is found by searching from the right-hand side of the histogram; i.e. from the longest idle intervals, until the correct total number of messages has been found. That is, if the histogram has been collected for total time $X$, then the previous section gives the number of messages to delay as $N = lX \;/\; T_w$, where $l$ is the local performance bound. The threshold is given by the max-point of the smallest bin that has a total of at most $N$ messages in all bins to its right.

The amount of work per message is constant and rather small, since it is only necessary to update the histogram and search for the correct value of the Stall-Timer. In our experiments, the Stall-Timer value is updated after every idle link event, but it can be updated less frequently if desired. Alternatively, the algorithm can easily be optimized to take advantage of the fact that the correct value of the Stall-Timer seldom moves by more than one bin at a time.

Figure 12 shows three important characteristics of the algorithm. The $x$-axis is time, or more accurately a sequence number for the idle link event, and the $y$-axis is the value of the Stall-Timer, measured for a particular, but arbitrary, edge link (other edge links had similar behavior). Firstly, the correct value of the Stall-Timer differs dramatically between benchmarks—notice the logarithmic scale on the $y$-axis. Secondly, most applications rapidly converge to a stable value of the Stall-Timer, within just 200 events. This stable value can be compared with the point in Figure 5 where the overhead drops below 1%. Thirdly, for some benchmarks, most clearly LU, the Stall-Timer value is seen to adapt to varying application phases. The fast convergence of the histogram allows for its periodic refresh in order to account for major changes in application behavior. Refresh periods for the histogram is discussed in more detail in Section 6.3.

## 6 DynamicFastwake: Extending PerfBound to Multiple Sleep States

This section presents DynamicFastwake, an extension to PerfBound to control hybrid Fast-Wake + Deep-Sleep whose benefits were outlined in Section 4. The discussion in this section primarily focuses on a single-hop two node system for simplicity, however it should be clear how PerfBoundRatio can directly be applied for use in multi-hop networks.

### 6.1 Extending the histogram to support hybrid Fast-Wake + Deep-Sleep

The previously mentioned PerfBound heuristic discusses how a target number of delayed messages can be mapped to a single Stall-Timer value. This is done using a histogram[8] that records for each link, the lengths of their idle periods. Figure 13 reproduces the histogram discussed in Section 5 to compare with the operation of DynamicFastwake. With PerfBound, Stall-Timer values are found by searching from the right-hand side of the histogram, starting from the longest idle period, until the target number of messages has been counted. With hybrid Fast-Wake and Deep-Sleep, the *number* of messages to delay is not fixed, as it depends on how many messages cause a wake from Fast-Wake, each incurring a delay of $T_{\mathrm{FW}}$, and how many wake from Deep-Sleep, each incurring a delay of $T_{\mathrm{DS}}$. The first step should therefore identify the total of the wakeup times, $T_{\mathrm{D}} = lX$, rather than the number of messages to delay.[9]

8. In this study, the possible Stall-Timer values are divided logarithmically into one hundred bins between 1 µs to 100 ms. Increasing the number of bins will allow for finer-grain Stall-Timer choices, but experiments with larger histograms showed no major benefit. A hardware timer tracks in microseconds the duration of link idle periods (after the frame buffer is empty). When a new frame arrives, the histogram is updated. All idle periods less than 1 µs are ignored since the Stall-Timer is likely to be larger than 7 µs (sleep + wake time of a EEE link is 7 µs).

9. $X$ is the total time the histogram was collected and $l$ is the local overhead performance bound. The local performance overhead bound could be calculated in the same way as PerfBoundRatio as described in Section 5.1.

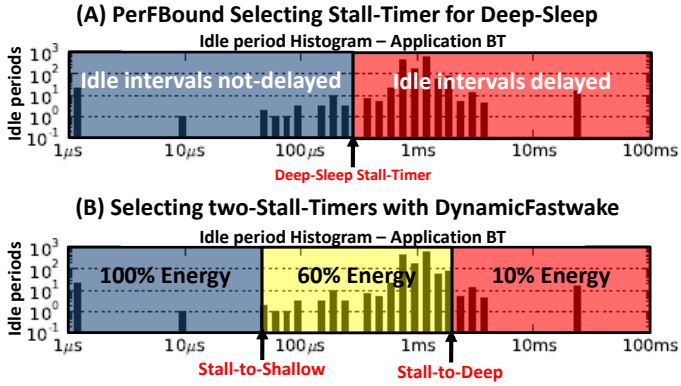


Fig. 13: **Idle period histograms for Application BT, illustrating effect of Stall-Timer placement on link delay and energy savings**

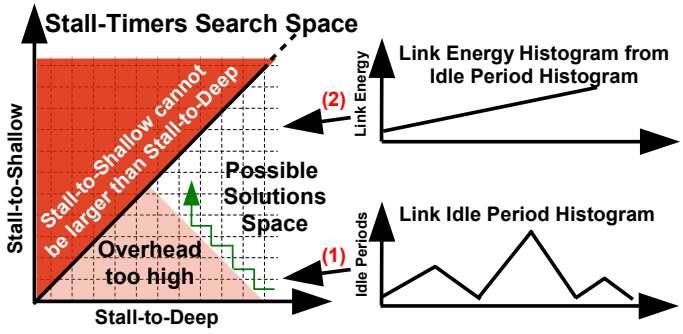


Fig. 14: **Stall-Timer search logic of DynamicFastwake; (1) Idle period histogram used to find possible solutions and (2) for each possible solution, energy histogram is used to find lowest energy solution**

Figure 13(B) shows how the idle period histogram is used by DynamicFastwake. Instead of a single Stall-Timer, the algorithm must determine two Stall-Timers, Stall-to-Shallow and Stall-to-Deep. Similarly to Figure 13(A), the left of the histogram has short idle intervals, of length less than Stall-to-Shallow, during which the link remains active, at 100% power, and there is no wake-up penalty. In the middle of the histogram are intervals larger than Stall-to-Shallow but less than Stall-to-Deep, during which the link (eventually) enters Shallow-Sleep, consuming 60% power but later incurring a Fast-Wake penalty of $T_{\text{FW}}$. The right of the histogram has intervals longer than Stall-to-Deep, during which the link enters Deep-Sleep, consuming just 10% power but later incurring the full penalty of $T_{\text{DS}}$.

### 6.2 Stall-Timer Search for DynamicFastwake

The search for the optimal pair of Stall-Timer values is illustrated in Figure 14. The two-dimensional space on the left of the figure contains all potential pairs of Stall-Timer values. Only the pairs in the lower-right triangle are valid, since Stall-to-Shallow must always be less than or equal to Stall-to-Deep. The (pink) region in the lower-left contains valid Stall-Timer pairs that should be excluded because the estimated performance overhead is greater than the link's local performance bound, $l$. The algorithm then chooses the acceptable solution with the greatest estimated energy savings. As indicated at the right, and explained

**Algorithm 1** DynamicFastwake search algorithm

$B_{\text{D}} = D_{\text{MAX}}$ ▷ Stall-to-Deep bin index
$B_{\text{S}} = 0$ ▷ Stall-to-Shallow bin index
$E_{\text{BEST}} = 0$ ▷ Best energy savings so far
$B_{\text{BEST}} = (B_{\text{D}}, B_{\text{S}})$ ▷ Best Stall-Timers so far
**do**
  Calculate $T_{\text{overh}}$ and $E_{\text{sav}}$
  **if** $T_{\text{overh}} < T_{\text{D}}$ **then** ▷ Acceptable, so Move left
    **if** $E_{\text{sav}} > E_{\text{BEST}}$ **then**
      $E_{\text{BEST}} = E_{\text{sav}}$
      $B_{\text{BEST}} = (B_{\text{D}}, B_{\text{S}})$
    **end if**
    $B_{\text{D}} = B_{\text{D}} - 1$
  **else** ▷ Rejected, so Move up
    $B_{\text{S}} = B_{\text{S}} + 1$
  **end if**
**while** $B_{\text{S}} \leq B_{\text{D}}$
**return** $B_{\text{BEST}}$

in detail below, the performance overhead and energy savings are estimated using histograms derived from that of Figure 13(B). Also, two "monotonicity" properties, also described below, mean that the search is one-dimensional, not two, and this is suggested by the (green) arrow.

The performance overhead is estimated using the idle period histogram of Figure 13(B), illustrated at the bottom-right of Figure 14 and marked (1). Given the two Stall-Timer values, the histogram is used to determine the number of idle periods that are followed by a Fast-Wake, denoted $M_{\text{FW}}$, and the number of idle periods followed by a full-wake, denoted $M_{\text{DS}}$. The estimated overhead is then $T_{\text{overh}} = M_{\text{FW}} \cdot T_{\text{FW}} + M_{\text{DS}} \cdot T_{\text{DS}}$. The first step of DynamicFastwake calculated the total acceptable wakeup time as $T_{\text{D}} = lX$, so the Stall-Timer pair is excluded whenever this bound is exceeded; i.e. if $T_{\text{overh}} > T_{\text{D}}$. Otherwise, the pair of Stall-Timers give an acceptable solution, although it may not be optimal in terms of energy.

The energy savings are estimated as shown at the top-right of Figure 14 and marked (2). This step is done using the *energy histogram*, which measures the total idle time in each bin, and is proportional to the energy savings, rather than the idle period histogram, which measures the *number* of idle intervals in the bin, and was proportional to the performance overhead. In fact, instead of separately collecting the energy histogram, we found that it was sufficient to approximate it from the idle period histogram in Figure 13(B), simply by multiplying the number of idle periods in each bin by the mid-point of the bin. If $I_{\text{FW}}$ is the total idle time in idle periods followed by a Full-Wake, determined from the energy histogram, and $I_{\text{DS}}$ is the total idle time in idle periods followed by a Deep-Sleep, also from the histogram, then the energy savings are estimated to be $E_{\text{sav}} = \frac{40}{100} I_{\text{FW}} + \frac{90}{100} I_{\text{DS}}$.

Combining the above, the optimal pair of Stall-Timers is chosen to maximise $E_{\text{sav}}$ subject to $T_{\text{overh}} \leq T_{\text{D}}$, and this can be naively done using a two-dimensional search over all pairs of valid Stall-Timers.

The optimised one-dimensional search is given in Algorithm 1. This algorithm manipulates the bin indexes for the two Stall-Timers, labelled $B_{\text{D}}$ and $B_{\text{S}}$, rather than the Stall-Timers themselves. It starts with Stall-to-Deep at

its maximum value ($B_D = D_{MAX}$) and Stall-to-Shallow at zero ($B_S = 0$). Whenever the performance overheads are acceptable, then the best energy savings is updated if necessary, and Stall-to-Deep is reduced (by decrementing $B_D$), moving to the left in Figure 14. If the overheads are too high, then Stall-to-Shallow is increased (by incrementing $B_S$), moving up. The algorithm terminates when the Stall-Timers cross, which is when the solution enters the invalid upper-left triangle in Figure 14. Since it is always true that $0 \leq B_S \leq B_D \leq D_{MAX}$, both Stall-Timers stay in bounds.

This algorithm can be proved to be correct, using two "monotonicity" properties. These are easier stated if the values of Stall-to-Shallow and Stall-to-Deep are abbreviated as $S_S$ and $S_D$, respectively. Firstly, if $(S_D, S_S)$ is excluded because the overhead is too high, then clearly all $(S'_D, S'_S)$ with $S'_D \leq S_D$ and $S'_S \leq S_S$ have smaller Stall-Timers, and therefore larger overheads, so they must be excluded also. Secondly, if $(S_D, S_S)$ has acceptable overheads and the energy savings have been estimated, then all $(S'_D, S'_S)$ with $S'_D \geq S_D$ and $S'_S \geq S_S$ have larger Stall-Timers, and therefore smaller energy savings, so they cannot be closer to optimal.[10]

Using these properties, the algorithm can be shown to be equivalent to scanning the two-dimensional space in Figure 14, starting from the lower-right, and moving right-to-left then bottom-to-top, excluding pairs that do not need to be checked due to the monotonicity relationships. When a pair is rejected because the performance overheads are unacceptable, then performance monotonicity implies that all values to the left can be ignored, as the performance overheads will also be unacceptable. Moreover, all values one row up and strictly to the right can also be excluded due to energy monotonicity applied to the previously visited Stall-Timer pair. Hence the next potentially optimal pair of Stall-Timers is found by moving up.

Since $B_D - B_S$ starts at $D_{MAX}$, and it decreases by one in each iteration of the loop, the number of iterations is simply $D_{MAX}$. This is a one-dimensional search with the same worst worst case as the original PerfBound. It is also worth noting that it is not necessary to recalculate $T_{overh}$ and $E_{sav}$ from scratch on each iteration, since, each iteration, only the contributions from a single bin are changed. This optimization is not included in Algorithm 1 for the sake of simplicity.

### 6.3 Stall-Timer Error Correction

This section describes a low-complexity feed-forward control mechanism that improves the accuracy of PerfBound and in extension of DynamicFastwake in reaching a target performance overhead bound. As described above, PerfBound translates its performance overhead bound to a target number of messages that are allowed to be delayed and then chooses Stall-Timer values to match this target. Although the algorithm is reasonably accurate, the target number of messages that can be delayed does not always exactly match the actual number of messages delayed, as observed at the links. This is mainly because of the discrete nature of histograms. In particular, the Stall-Timers are always chosen to be at a max-point of one of the bins.

10. The phrase "has acceptable overheads" is necessary because when the overheads are large, increasing Stall-Timers may actually save energy indirectly through a decrease in execution time.

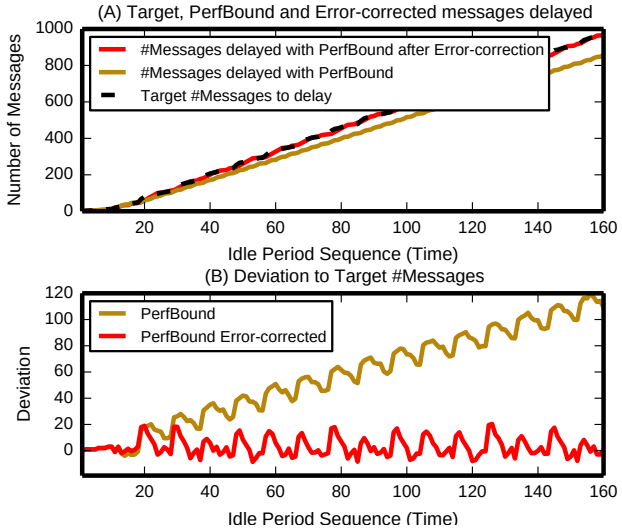


Fig. 15: **Stall-Timer Error Correction and #Target messages to actual messages delayed by PerfBound and Dynamic-Fastwake for BT**

Increasing the number of bins reduces the error, at the cost of increased computation time.

Figure 15 (A) illustrates this tracking error, for application BT. The dashed black line is the target number of delayed messages, and the yellow line is the actual number of messages delayed by PerfBound. As seen, there is a consistent error of about 20%, with the actual number of delayed messages lower then the target, meaning that PerfBound achieves lower energy savings than it could have. This tracking error was similarly observed in other examined applications.

Figure 15 (B) plots the error, given by the difference between actual number of messages delayed and target number of messages delayed as presented in Figure 15 (A). Here, zero represents no deviation between target and actual number of messages delayed. It is clear from Figure 15 (B) that this deviation increases without bound for the previous PerfBound (in red).

This is solved with the use of a simple feed-forward controller that monitors the difference between the actual and target numbers of delayed messages. If this difference exceeds a fixed value of twenty messages, it applies an offset to the Deep-Sleep Stall-Timer to force the difference towards zero. In Figure 15(A), the red line is the number of messages delayed after this error correction and in Figure 15(B), the red line shows the deviation from the target. As shown, the error after correction stays close to zero.

**Histogram Refresh:** The possible Stall-Timer values in this study were divided into 100 histogram bins between 1 µs to 100 ms. Increasing the number of bins would allow for finer-grain Stall-Timer choices, but experiments with larger histograms showed no major benefit. A hardware timer tracks in microseconds the duration of link idle periods (after the frame buffer is empty). When the next frame arrives, the histogram is updated. All idle periods less than 1 µs are ignored since the Stall-Timer is likely to be larger than 7 µs (sleep + wake time of a EEE link is 7 µs).

Although this discussion focuses on *unit-time* for sim-

plicity, histograms are refreshed in *idle link events*, e.g. every 20,000 idle link events, irrespective of the elapsed time or application. Figure 12 shows that, for all applications, the algorithm converges within 200 events, which is only 1% of 20,000, hence a negligible fraction. When a new application begins, only the first refresh cycle has events from the old application. In the worst case, at $4\,\mu s$ and 6 hops/message incurred on all 20,000 events in the first refresh cycle, the overhead is $\leq 480\,ms$, which is negligible compared with typical application execution times.

## 7 Results

The performance–energy trade-offs for the DynamicFastwake algorithm are shown in Figure 16. These results were obtained using the methodology and applications described in Section 3. As for Figure 9, for each application, the $x$-axis is the execution time and the $y$-axis is the average link energy, both normalized to the corresponding values for an always-on network. In these plots, moving down and/or left is preferred, since this corresponds to decreasing energy and/or improving performance, respectively. In practice, the mechanism offers a trade-off between performance and energy; hence the curves.

The baseline results are as follows. The dashed blue line is a static sweep of a single Stall-Timer, and the large yellow triangle is the result from PerfBound, configured for a performance overhead of 1%. The smaller yellow triangles are results for PerfBound configured for performance overheads of 0.5%, 2%, 4% and 6%.

The Fast-Wake results are as follows. The dashed black line is the Pareto-optimal curve from an exhaustive static search over all pairs of Stall-Timers, obtained as explained in the description of Figure 9. Finally, the large red square is the result from DynamicFastwake configured for a performance overhead of 1%. The small red squares are obtained by varying the performance overhead parameter as for PerfBound. This section focuses on discussing the 1% performance overhead configurations, which were highlighted by the larger points. The conclusions from the other configurations are similar.

Before further discussion of Fast-Wake and DynamicFastwake, it is important to visit the savings obtained from Deep-Sleep and PerfBound alone. Deep-Sleep as seen in Figure 16 already obtains more than 60% link energy savings for nine out of the fourteen applications. PerfBound automatically finds its energy-performance curve on the Deep-Sleep curve, restricting performance overheads while also obtaining the potential energy savings.

The first observation from Figure 16 for DynamicFastwake is that for ten of the fourteen applications it finds a result on or below the static Pareto-optimal curve. Specifically, the DynamicFastwake curve overlaps the static Pareto-optimal curve. Applications SP and MILC has DynamicFastwake values below the Pareto-optimal curve, indicating higher energy and lower performance overheads than obtainable with a sweep. The reason for this is that all links of a sweep are configured with a single fixed Stall-Timer. An ideal sweep would vary the Stall-Timers for all links independently, but this would have translated to an enormous number of parameters to explore, which is not feasible. Since PerfBound and DynamicFastwake are independent and local to each link, the heuristics naturally configure different Stall-Timers for each link, depending on locally visible traffic and can thus be better than the sweeps.

In Figure 16, for a 1% overhead, SP shows about 20% improvement in energy from DynamicFastwake. This was the dangerous example discussed in Section 4.4, for which the Stall-Timer speed contains values far from the Pareto-optimal curve. DynamicFastwake for SP has its value on the Pareto-optimal curve itself. Similarly, BT and MILC also show high energy savings from DynamicFastwake. The 5% higher energy saving potential observed for MG in Section 4.4 is captured by DynamicFastwake. For application ALYA however only 2% is saved from its possible 5%. Although even ALYA, for performance overheads larger than 1%, have DynamicFastwake values overlapping the Pareto-optimal curve. Section 4.4, specifically discusses ALYA for its many combinations of possible Stall-Timer values that are significantly worse than the Pareto-optimal curve or even the Deep-Sleep sweep. To this end, for ALYA, both DynamicFastwake and PerfBound have nearly optimal energy to configured performance overheads. Similar to ALYA and MG are applications, CG, WRF and GROMACS with the similar 5% improvement in energy savings.

As remarked in Section 4.4, some applications, specifically GADGET, LINPACK and QUANTUM, have no opportunity to obtain additional energy savings from Fast-Wake. Therefore, for the three applications where Fast-Wake offers no additional energy savings, DynamicFastwake found the same Pareto-optimal solution as PerfBound.

Figure 16 also presents the accuracy of DynamicFastwake in maintaining its configured performance overhead bound. **For twelve out of fourteen applications, the deviation from the configured performance overhead was less than 1%**. This error is potentially due to the inherent application behavior. For example, when an application ends immediately after a network long idle period, with no further messages, an opportunity to delay messages is lost. This is because, with the PerfBound heuristic, the number of messages delayed and in extension incurred performance overhead is proportional to time. A large idle period introduces time that could potentially be used to delay more messages for energy savings. This however cannot be exploited if the application immediately ends with no new impending messages. Hence in this scenario expected overhead is slightly lower than the configured overhead bound. Similarly a large burst of messages towards the end of an application could introduce many delayed messages, and may not give the heuristic time to react or adjust its Stall-Timers to correspondingly control overheads caused by the same. This may cause a small increase in overhead compared to the bound. Two applications, specifically NAMD and LU have higher overhead than the configured bound but still less than 5% for the configured 1% bound. These applications have high inter-message dependencies that PerfBound does not account for in its calculation. For a 1% target overhead, the average actual overhead across all applications, including NAMD and LU, is 1.1% for PerfBound, and 1.4% for DynamicFastWake.

Out of fourteen, DynamicFastwake saves 70% energy for six applications and 40–70% for another six, with a performance overhead bounded to only 1%. The two ap-

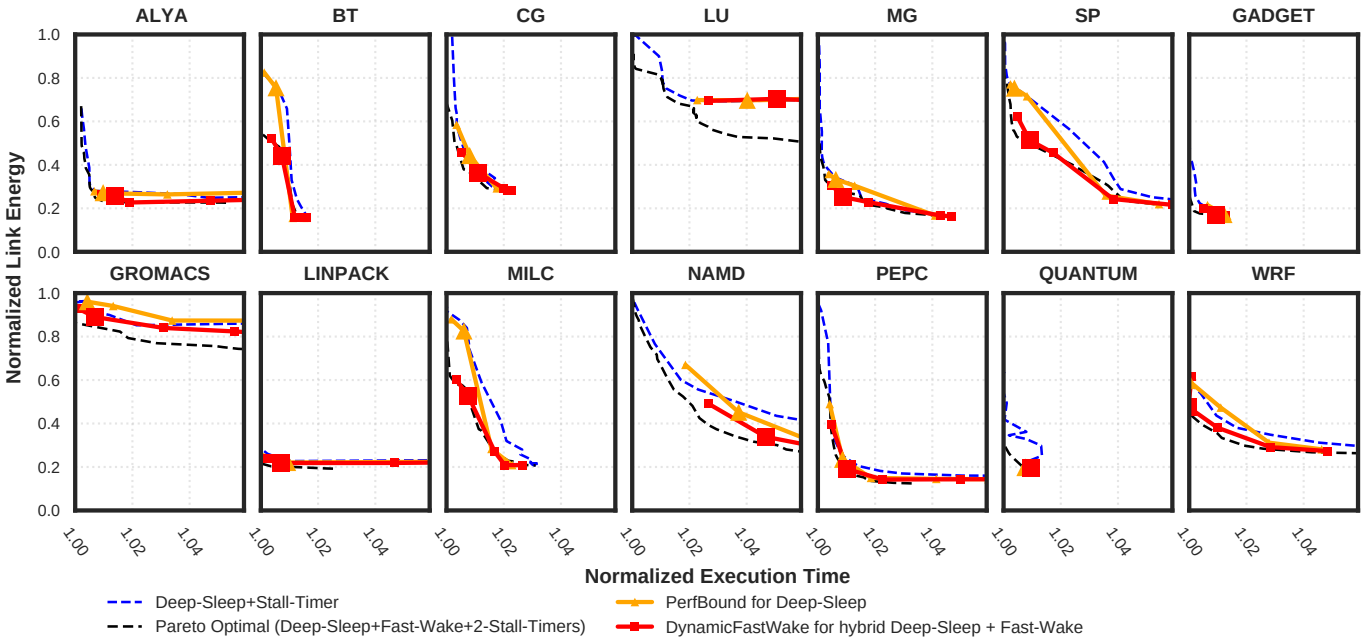


Fig. 16: **Energy and Performance of various configurations of DynamicFastwake (0.5%, 1% (large square), 2%, 3% and 4%) compared to an exhaustive search and Pareto-optimal curve of the hybrid Deep-Sleep+Fast-Wake mechanisms**

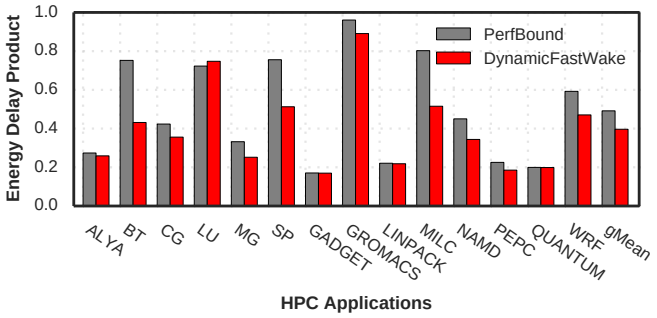


Fig. 17: **Energy Delay Product obtained from Figure 16**

plications that have lower than 40% energy saving, LU and GROMACS both have high network usage. GROMACS in specific is also latency sensitive and a bad choice of Stall-Timers have the potential for large performance overheads [9]. While DynamicFastwake does not save energy when the opportunity is limited, it still bounds performance overheads.

Figure 17 compares the energy–delay product (EDP) for DynamicFastwake and PerfBound, with applications configured at a 1% performance overhead bound. Applications GADGET, LINPACK and QUANTUM obtain no additional benefit from Fast-Wake over Deep-Sleep, so their results are not improved by DynamicFastWake. Applications CG, MG, NAMD, PEPC show an improvement of between 5% and 10%. Applications BT, SP, MILC and WRF show large improvements over PerfBound of 35%, 25%, 30%, 12% respectively. On average **DynamicFastwake has EDP 10% better than PerfBound**.

As mentioned in the introduction, interconnects consume about 12% of the system energy, and about 60% of this is due to the interconnect links. This means that 7.2% of the total system energy costs are due to the interconnect links. DynamicFastWake saves 70% of the link energy, which is 5% of the system energy. Assuming that the 1% increase in execution time translates to a 1% increase in the energy consumption of the rest of the system, which is pessimistic as it ignores energy-proportionality of memory, CPUs and GPUs, the overall saving in system energy consumption from DynamicFastWake is therefore 4%. This is an improvement of 0.4% over PerfBound alone.[11]

This 4% reduction in system energy consumption has a significant impact on the total cost of ownership. For example, a large-scale HPC system consuming 20 MW costs about 20 million USD per year,[12] which is a total of 80 million USD over a four-year lifespan [6]. This means that adopting DynamicFastWake instead of the current approach would provide 3.2 million USD in savings per machine over its 4-year lifetime.[13] The improvement of DynamicFastWake over PerfBound is 0.57 million USD over its lifetime.

## 8 Related Work

Jian Li, et al., [8] discuss on/off networks that use snoop messages that arrive at the NICs as an indication of an impending message. In nodes that have snoop-based coherence, snooping messages would arrive at the link before an impending message, which could be used to turn the link on, before the actual arrival of the message. They also propose the use of an always-on control network that sends control signals through the routing path of a message to wake up subsequent links. They further propose software enhancements which would have programmers annotate

11. An improvement of 10% of link energy is 0.72% of system energy. DynamicFastWake increased execution time by 0.3% above PerfBound, so the overall saving in system energy is 0.4%.

12. The fastest TOP500 supercomputer consumes 22 MW at 35 PFLOPS, and an exaflop machine is expected to consume 20 MW to 60 MW.

13. This is 4% of 80 million USD.

the code signalling an impending message. Similarly, Soteriou, et al., [15] show that on/off networks incur a large performance penalty and hence, they propose software mechanisms such as parallelizing compilers for network power savings. Work by Saravanan, et al., [9], [13] analyzes HPC applications over on/off based links. Their work shows energy efficiency benefits in using on/off based links, however applications can suffer from performance overheads. R. Bertran, et al., [25], evaluate the power and performance trade- offs on the Blue Gene/Q. Among other results, their work shows that the links of Blue Gene/Q are also 'always-on' similar to that of Ethernet and InfiniBand. Their work shows scope for the ideas presented in this paper to be extended to non-Ethernet but on/off based networks.

Gupta, et al., in their work [19], show that opportunistic sleeping of links is possible, but their technique increases the mean latency. Vassos, et al., [17] discuss a design space analysis for on/off based links. They propose using multiple routing paths available in torus like networks to shut down parts of the network during periods of low load. They evaluate their proposal with message arrivals following a Poisson process. Similarly, Alonso, et al., [18] propose shutting down redundant links (sub-trees) in their fat-tree system to save energy. These proposals do not discuss the performance impact of bursty communications that are typical of HPC.

Silva, et.al., [16] target Data-Centers with evaluation of Energy Efficient Ethernet based networks over MapReduce. This work evaluates Stall-Timers to show low performance or energy benefit with MapReduce clusters. The work shows that MapReduce is not sensitive to latency and is not affected by wake-up delays and in extension does not require Stall-Timers. Similarly Ethernet evaluation reports [10], [11] use synthetic benchmarks to evaluate on/off networks. Relevant work on Energy Efficient Ethernet [4], [10], [11], [21], [22], [23] provide detailed evaluations on EEE for its potential for desktop and IT based systems, but they do not target HPC workloads. Totoni, et al., [24] show that not all links of a network executing an HPC application are utilized, so they propose runtime techniques to find links in the network that are never utilized, in order to turn them off. Their work, however, does not adaptively turn on/off links.

D. Abts, et al., [7] proposed energy proportional interconnects based on reducing the link rates of aggregated links. In their approach, during periods of inactivity, link rates are reduced to a lower link bandwidth to save energy. B. Dickov, et al., [27] present power savings for InfiniBand networks. They show similar energy saving potential is available in HPC systems and use prediction based methods to save link energy. Work by Kim, et al., [20] evaluate energy proportional networks and compare links based on dynamic voltage scaling and on/off links. They show that dynamic voltage scaling in links causes a significant increase in latency and show that on/off based techniques perform comparatively better.

## 9 Conclusions

Energy consumption is a key challenge in high-performance computing, but the primary goal will continue to be performance. Mechanisms to reduce system energy consumption will, therefore, only be employed if the impact on performance is known and small. Switches and NICs implementing the new Energy Efficient Ethernet standards, with support for Fast-Wake, will soon be deployed in HPC systems, but, without continued investigation, these features will likely be disabled by default.

This paper PerfBound and DynamicFastwake, heuristics to dynamically manage on/off links, minimizing the interconnect link energy consumption with a bound on the performance degradation. PerfBound and DynamicFastwake were evaluated using traces from a production supercomputer. For twelve of fourteen applications, the heuristics presented finds energy/performance results on or below the static Pareto-optimal curve. Overall, PerfBound obtained up to 70% savings on link energy. Furthermore, DynamicFastwake achieves on top of PerfBound 10% better EDP. The proposed techniques require no changes to the application, and, since it uses only local information already available at the switches and NICs, there is no additional communication. It is also compatible with multi-hop networks. With the ratification of Fast-Wake in March 2014 and 2015 for 40/100Gb Ethernet, and the ongoing standardization effort for 400Gb Ethernet, this work could help interconnect vendors to build energy-efficient switches and NICs that target HPC.

## Acknowledgments

This research was supported by European Union's 7th Framework Programme [FP7/2007-2013] under the Mont-Blanc-3 (FP7-ICT-671697) and EUROSERVER (FP7-ICT-610456) projects, the Ministry of Economy and Competitiveness of Spain (TIN2012-34557 and TIN2015-65316), Generalitat de Catalunya (FI-AGAUR 2012 FI B 00644, 2014-SGR-1051 and 2014-SGR-1272), the European Union's Horizon2020 research and innovation programme under the HiPEAC-3 Network of Excellence (ICT-287759), and the Severo Ochoa Program (SEV-2011-00067) of the Spanish Government.

## References

[1] "IEEE Standard for Information Technology Local and Metropolitan Area Networks Specific requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications Amendment 5: Media Access Control Parameters, Physical Layers, and Management Parameters for EnergyEfficient Ethernet," *IEEE Standard 802.3az-2010 (Amendment to IEEE Std 802.3-2008)*, 2010, pp. 1–302.

[2] A. Marris, "Comments on EEE Operation," *IEEE P802.3bj Task Force*, Sep 2013; http://ieee802.org/3/bj/public/sep13/marris_3bj_01_0913.pdf.

[3] *TOP500 Supercomputers*; http://www.top500.org/.

[4] R. Hays, A. Wertheimer, and E. Mann, "Active/Idle Toggling with Low Power Idle," *IEEE 802.3az Task Force Group Meeting*, Jan 2008; http://www.ieee802.org/3/az/public/jan08/hays_01_0108.pdf.

[5] J. Koomey, *Growth in Data Center Electricity Use 2005 to 2010*, Analytics Press, 2011.

[6] T. Ludwig and M. Dolz, Total Cost of Ownership in High Performance Computing, 2014; https://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2014/tco-14-intro.pdf.

[7] D. Abts, M.R. Marty, P.M. Wells, P. Klausler, and H. Liu, "Energy Proportional Datacenter Networks," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010, pp. 338–347.

[8] J. Li, W. Huang, C. Lefurgy, L. Zhang, W.E. Denzel, R.R. Treumann, and K. Wang, "Power Shifting in Thrifty Interconnection Network," *Proc. 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 156–167.

[9] K.P. Saravanan, P.M. Carpenter, and A. Ramirez, "Power/performance Evaluation of Energy Efficient Ethernet (EEE) for High Performance Computing," *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[10] K. Christensen et al., "IEEE 802.3 az: The Road to Energy Efficient Ethernet," *IEEE Communications Magazine*, vol. 48, no. 11, 2010.

[11] P. Reviriego et al., "Performance Evaluation of Energy Efficient Ethernet," *IEEE Comm. Letters*, vol. 13, no. 9, Sep. 2009, pp.697–699.

[12] K.P. Saravanan, P.M. Carpenter, and A. Ramirez, "A Performance Perspective on Energy Efficient HPC Links," *Proc. 28th ACM International Conference on Supercomputing (ICS)*, 2014, pp. 313–322.

[13] K.P. Saravanan, P.M. Carpenter, and A. Ramirez, "Exploring Multiple Sleep Modes in On/Off Based Energy Efficient HPC Networks," *Proc. 33rd IEEE International Conference on Computer Design (ICCD)*, 2015, pp. 54–61.

[14] H. Barrass, "Options for EEE in 100G", *IEEE P802.3bj Meeting*, Newport Beach, CA, USA, 2012.

[15] V. Soteriou, N. Eisley, and L.-S. Peh, "Software-Directed Power-Aware Interconnection Networks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 1, 2007.

[16] R. Fischer e Silva and P.M. Carpenter, "Exploring Interconnect Energy Savings Under East–West Traffic Pattern of MapReduce Clusters," *Proc. 40th IEEE Conference on Local Computer Networks (LCN)*, 2015, pp. 10–18.

[17] V. Soteriou and L.-S. Peh, "Design-Space Exploration of Power-Aware On/Off Interconnection Networks," *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, 2004, pp. 510–517.

[18] M. Alonso et al., "Dynamic Power Saving in Fat-Tree Interconnection Networks Using On/Off Links," *Proc. 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[19] M. Gupta and S. Singh, "Dynamic Ethernet Link Shutdown for Energy Conservation on Ethernet Links," *Proc. IEEE International Conference on Communications (ICC)*, 2007, pp. 6156–6161.

[20] E. Kim et al., "Energy Optimization Techniques in Cluster Interconnects," *Proc. 2003 International Symposium on Low Power Electronics and Design (ISLPED)*, 2003, pp. 459–464.

[21] H. Anand et al., "Ethernet Adaptive Link Rate (ALR): Analysis of a MAC Handshake Protocol," *Proc. 31st IEEE Conference on Local Computer Networks (LCN)*, 2006, pp. 533–534.

[22] F. Blanquicet and K. Christensen, "An Initial Performance Evaluation of Rapid PHY Selection (RPS) for Energy Efficient Ethernet," *Proc. 32nd IEEE Conference on Local Computer Networks (LCN)*, 2007, pp. 223–225.

[23] M. Koibuchi et al., "An On/Off Link Activation Method for Low-Power Ethernet in PC Clusters," *Proc. IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009, pp. 1–11.

[24] E. Totoni, N. Jain, and L.V. Kale, "Toward Runtime Power Management of Exascale Networks by On/Off Control of Links," *Proc. IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2013, pp. 915–922.

[25] R. Bertran et al., "Application-Level Power and Performance Characterization and Optimization on IBM Blue Gene/Q Systems," *IBM Journal of Research and Development*, vol. 57, no. 1/2, 2013, pp. 4:1–4:17.

[26] V. Soteriou and L.-S. Peh, "Dynamic Power Management for Power Optimization of Interconnection Networks Using On/Off Links," *Proc. 11th Symposium on High Performance Interconnects (HOTI)*, 2003, pp. 15–20.

[27] B. Dickov et al., "Software-Managed Power Reduction in InfiniBand Links," *43rd International Conference on Parallel Processing (ICPP)*, 2014, pp. 311-320.

[28] R.M. Badia et al., "DIMEMAS: Predicting MPI Applications Behavior in Grid Environments," *Workshop on Grid Applications and Programming Tools (GGF8)*, vol. 86, 2003, pp. 52–62.

[29] S. Girona, J. Labarta, and R.M. Badia, "Validation of Dimemas Communication Model for MPI Collective Operations," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2000, pp. 39–46.

[30] J. Gonzalez et al., "Simulating Whole Supercomputer Applications," *IEEE Micro*, vol. 3, pp.32–45.

[31] E. Casoni et al., "Alya: Computational Solid Mechanics for Supercomputers," *Archives of Computational Methods in Engineering*, vol. 22, no. 4, 2015, pp. 557–576.

[32] *NAS parallel benchmarks*; www.nas.nasa.gov/publications/npb.html.

[33] D. Marx and J. Hutter, "Ab-initio Molecular Dynamics: Theory and Implementation," *Modern Methods and Algorithms of Quantum Chemistry*, Forschungszentrum Jülich, 2000.

[34] V. Springel, "The Cosmological Simulation Code GADGET-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, no. 4, 2005, pp. 1105–1134.

[35] B. Hess et al., "GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation," *Journal of Chemical Theory and Computation*, vol. 4, no. 3, 2008, pp. 435–447.

[36] T. Davies et al., "High Performance Linpack Benchmark: a Fault Tolerant Implementation Without Checkpointing," *Proc. International Conference on Supercomputing (ICS)*, 2011, pp. 162–171.

[37] *MIMD Lattice Computation (MILC) Collaboration*; http://physics.indiana.edu/~sg/milc.html.

[38] R.K. Brunner, J.C. Phillips, and L.V. Kalé, "Scalable Molecular Dynamics for Large Biomolecular Systems," *Scientific Programming*, vol. 8, no. 3, 2000, pp. 195–207.

[39] *PEPC: Pretty Efficient Parallel Coulomb-solver*, Tech Rep., Zentralinstitut für Angewandte Mathematik.

[40] P. Giannozzi, et al., "QUANTUM ESPRESSO: A Modular and Open-Source Software Project for Quantum Simulations of Materials," *Journal of Physics: Condensed Matter*, vol. 21, no. 39, 2009.

[41] J. Michalakes et al., "The Weather Research and Forecast Model: Software Architecture and Performance," *Proc. 11th ECMWF Workshop on the Use of High Performance Computing in Meteorology*, 2005, pp. 156–168.


**Karthikeyan P. Saravanan** obtained his PhD in Computer Architecture from the Universitat Politècnica de Catalunya (UPC), Spain in 2016. During his PhD he also worked as a researcher at the Barcelona Supercomputing Center (BSC), Spain. He holds an Undergraduate degree in Computer Science and Engineering (obtained in 2010) from Anna University, India.

**Paul M. Carpenter** graduated from the University of Cambridge in 1997 and received his PhD in computer architecture from the Technical University of Catalonia in 2011. Prior to starting his PhD, he was Senior Software Engineer at ARM in Cambridge and technical lead for audio/video codecs. He is Senior Researcher at BSC, where he is Principal Investigator of the EuroEXA and ExaNoDe projects. He is co-chair of the ETP4HPCs Working Group on programming models and (co-)director of four PhD students. His research interests include server system architecture, programming models and virtual machine resource sharing.