Low-overhead, High-speed Multi-core Barrier Synchronization

John Sartori and Rakesh Kumar

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

Abstract. Whereas efficient barrier implementations were once a concern only in high-performance computing, recent trends in core integration make the topic relevant even for general-purpose CMPs. While the nature of CMP applications requires low-latency, the cost of low-latency barrier implementations using hardware-based techniques can be prohibitive for CMPs, where die area represents opportunities for throughput and yield. Similarly, whereas traditional multiprocessor barrier implementations were developed primarily for dedicated environments, scheduling and multi-programming on CMPs require more adaptable barrier implementations.
In this paper, we present and evaluate three barrier implementations that are hybrids of software and dedicated hardware barriers and are specifically tailored for CMPs. The implementations leverage the unique characteristics of CMPs and provide low latency comparable to that of dedicated hardware networks at a fraction of the cost. The implementations also support adaptability, enabling efficient multi-programming and dynamic remapping of the barrier network.

1 Introduction

Barrier synchronization has been a well-studied problem for large-scale, traditional multiprocessors [1–4]. A wide variety of barrier implementations have been proposed, ranging from software-based [2, 5–8] to fully hardware-based [3, 9–12]. Several of these implementations have been used in the context of large-scale parallel applications with large data sizes, coarse-grained parallelism, and high computation to communication ratios.

Requirements for barrier synchronization for CMPs are different, however. In contrast to typical multiprocessor applications which target coarse-grained parallelism, multi-core applications tend to exploit fine-grained parallelism, making low-latency synchronization a primary concern. Consequently, multi-core applications can be highly sensitive to barrier performance. For example, Figure 1 shows the performance of three OpenMP NAS benchmarks [13] that exploit inner-loop parallelism. As the granularity of parallelism decreases, the overhead of barrier synchronization becomes relatively larger, and performance degrades. As the results show, performance can be very sensitive to barrier latency for applications with fine-grained parallelism. So, a barrier implementation for multi-cores should have low latency.

Also, low latency barrier implementations have traditionally been achieved through dedicated hardware support. For CMPs, however, the high area and power overheads of hardware barrier implementations are particularly taxing. Figure 2 shows how the area overhead of additional dedicated links scales with the number of cores for a dedicated hardware barrier tree implementation in 65nm technology. The area cost of dedicated links is determined by the thickness (dependent on metal layer) and length of the wires [14], and may represent a considerable fraction of the precious die area for CMPs (up to 16% assuming a $400mm^2$ die). Since die area is a precious resource for CMPs (as it can translate into higher throughput – saved area can be used for more cache or cores – or higher yield – yield varies inversely with die area), a barrier implementation for multi-cores cannot afford to ignore the area/power costs of providing low latency.

Finally, while applications for high-performance systems are typically run in a dedicated system mode, multi-core applications are often expected to run in shared environments with scheduling and multi-programming. So, barrier implementations for multi-cores should be adaptable for various levels of participation and dynamically configurable based on the mapping of threads to cores.

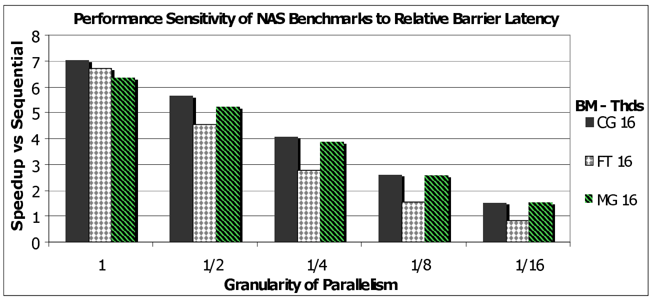

Fig. 1: The performance of three NAS parallel benchmarks degrades as the granularity of parallelism becomes finer and relative barrier latency increases on a 16-threaded workload. When relative barrier latency becomes high, as can be the case for software barrier implementations, taking advantage of fine-grained parallelism has little or no benefit.

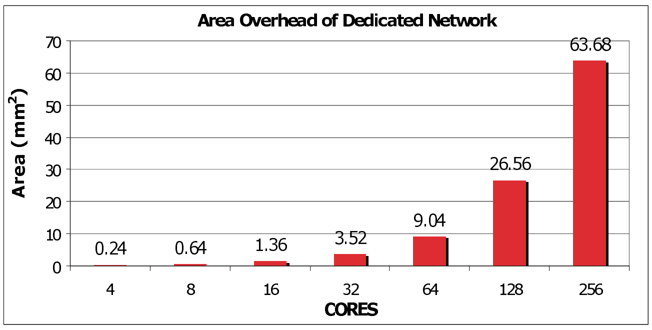

Fig. 2: A dedicated network adds considerable area overhead to the chip. The plot above shows the area overhead of a dedicated tree network in terms of additional wiring cost for 65nm technology.

In this paper, we revisit barrier synchronization for CMPs and present three CMP platforms that achieve barrier latencies close to those of dedicated hardware barrier networks at a fraction of the cost. The implementations support adaptability, enabling efficient multi-programming and dynamic remapping of the barrier network.

2 Hardware-supported Mapping of Virtual Barrier Topologies

Our first CMP-specific optimization accelerates software-based barrier implementations. One way to reduce the performance overhead of software-based barrier synchronization while keeping area overhead low is to form a virtual hierarchical network atop the physical mesh. In terms of topologies, a butterfly network can potentially achieve the lowest latency for global barrier synchronization. However, for this study, we consider a virtual tree network due to the high connectivity/area and messaging costs of the butterfly ($(N/2) \cdot log_2 N$ and $N \cdot log_2 N$, respectively) as compared to the tree ($N-1$ and $2 \cdot (N-1)$). We assume that the existing topology of the network-on-chip for general purpose communication is a mesh. The following section describes our first platform for providing low-overhead, accelerated barrier support.

2.1 Implementation

When a group of cores that will perform synchronization is mapped into a virtual barrier tree, we can reduce the software overhead of synchronization by adding a simple state machine to each router to control routing of intermediate barrier notifications in the interconnect, without involving the cores.

The state machine in each router (Figure 3) contains three registers and three notification bits. The registers store the location of the parent, left child, and right child of a node in the network. The notification bits record whether a node has received an arrival notification from left child, right child, and self. When a thread reaches the barrier, it sends a notification to itself. When all notification bits are set, the last arriving notification is forwarded to the parent node. When the barrier has been satisfied at the root, completion notices are propagated back down to the leaves of the tree.

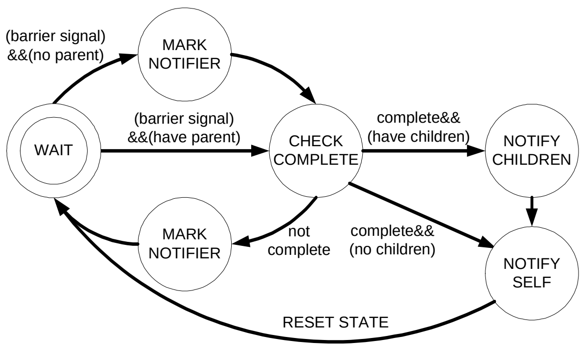

Fig. 3: This state machine describes the routing logic for barrier notifications in the virtual network. Arrival notifications are forwarded up the tree when the barrier is satisfied at the current level. Completion notifications are propagated down the tree after the barrier is satisfied at the root level.

Since propagation of the intermediate signals in the virtual barrier network can be performed in hardware, a node only needs to perform the initial arrival notification and the final check for barrier completion. Moreover, all of the notification details (such as neighbors, size, etc.) are determined and stored in the routers when the virtual network is configured. Therefore, rather than using a high overhead, generalized software procedure to send and receive notifications, we allocate a memory mapped address for use in barrier algorithms such that a store to the address sends an arrival notification to the network, and a load from the address stalls until the barrier completion notification is received at the node.

2.2 Benefits and Overheads

Adding hardware support for virtual networks reduces barrier latency by minimizing the software overhead of barrier management. Additional cost at routers consists only of a few registers and a small state machine. This approach adds no additional area overhead for communication links.

2.3 Map Optimization

While hardware support for virtual barrier networks improves the performance of barriers without adding much overhead, the performance gains are constrained if a naive strategy is used to map the virtual network onto the physical interconnect. In this section, we consider the benefits of intelligent mapping.

Determination of the optimal virtual to physical mapping involves finding the minimum depth spanning tree of the graph represented by the CMP cores (vertices) and mesh links (edges), where depth represents the longest distance from root to leaf. In general, this problem is NP-complete [15], and the goodness of a solution may depend on several factors, including the amount of time spent in computation. This may not be a problem for a statically assigned tree, but for the case of multicore processors, in which thread-to-core mappings are assigned dynamically at runtime and may change depending on availability of nodes and number of threads in a thread group, a static mapping will likely be inadequate.

The algorithm used to determine virtual to physical mappings is described in Figure 4. When a thread group will utilize barrier synchronization, this algorithm is executed by the processor's software runtime at the time a thread-to-core mapping is assigned for the task. Once the involved routers are configured by the runtime, no additional runtime support is needed during barrier execution. The algorithm selects a root near the center of the selected mesh nodes to maximize opportunities for fanout as the tree expands. Exhaustive testing over all possible tree roots confirms that the algorithm achieves minimum global barrier latency by minimizing the depth of the tree. Figure 5 gives examples of a naive mapping and an optimized mapping for a 16-core CMP.

```
NodeList.append(root); hasParent[root] = true
while NodeList not empty do
    POP Node i from NodeList with minimum depth
    for child in {left,right} do
        child = node with min distance(i, child) AND hasParent(child) = false
        NodeList.append(child); hasParent[child] = true
    end for
end while
```

Fig. 4: The virtual to physical mapping algorithm bears some similarities to Prim's algorithm for minimum spanning trees. The goal of the algorithm is to find the spanning tree with minimum root-to-leaf depth.

2.4 Platform Adaptability

The support for virtual barrier networks described above is easily adaptable to support semi-global synchronization of dynamic thread groups. When a processor's runtime schedules a group of threads, it may assign to them a dynamically computed virtual barrier tree if they will be performing barrier synchronization. In this case, the runtime selects a group of cores for the threads to use, computes a virtual to physical mapping for the graph represented by the cores and mesh links, and configures the state of the routers to connect and initialize the virtual network.

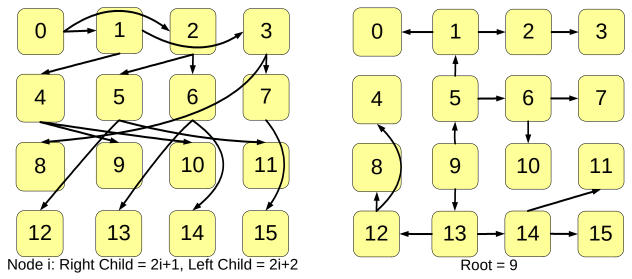

Fig. 5: In the figure on the left, an obvious (but naive) mapping strategy is used to assign tree neighbors. In this strategy, node i's children are at indices $2i + 1$ and $2i + 2$, and the resulting tree depth and aggregate hops of the mapping are 8 and 34. In an optimized approach to assigning the tree structure, a search is performed to find the best tree root, and each node's children are determined dynamically to minimize tree depth. The network on the right has a depth of 4 and an aggregate hop count of 17 – **half that of the naive mapping.**

3 Barrier Implementation using Hybrid Networks

While the previous CMP-specific barrier implementation accelerates software-based barrier implementations by providing hardware support for mapping virtual barrier topologies to physical topologies, in this section, we discuss a CMP-specific barrier implementation that tries to get the benefits of both software and hardware-based barrier implementations by creating a hybrid network.

A dedicated barrier network includes a dedicated link between two nodes that are neighbors in the topology. However, for a good mapping strategy, a direct connection may already exist between the two nodes in the form of a regular mesh link. Thus, adding an extra dedicated link does not buy additional performance in several cases. However, there *can* be a benefit for placing a dedicated link when there is not already a mesh link connecting two virtual neighbors in the barrier topology. This is the basis of a technique we call *barrier bolstering*.

3.1 Implementation

In an attempt to create a perfect barrier tree, with one hop between each level of the tree for all paths (each node is directly connected to all tree neighbors), we can add a dedicated physical link between two virtual neighbors any time a single-hop path does not exist between the nodes in the physical network. With this approach, the latency of the hybrid network would be very close to that of the dedicated network, and presumably, the cost would be lower, since some virtual links correspond directly to single physical links in the mesh. However, since wire delay depends on wire length, the effectiveness of this technique in reducing latency would be somewhat limited, since long wires (even long *dedicated* wires) would incur multiple-cycle delays. Also, due to limited connectivity at each switch in the mesh, the overhead of this strategy approaches that of a dedicated network as the number of nodes increases.

The previously mentioned approach to barrier bolstering can incur high area overheads when the number of cores is large. Also, perceived benefits may in reality be limited, since long wires require multiple cycle delays. Thus, for our actual implementation of barrier bolstering, we choose a more cost-effective technique for adding dedicated links, and we assume that link latency is proportional to link length in hops. Under this assumption, replacing long virtual links with dedicated links does not buy much performance relative to the cost.

While it may be unrealistic to assume substantial reduction in latency by replacing all multi-hop links with dedicated links, it is certainly possible to equalize the latency of links that have similar latencies by replacing the longer virtual link with a dedicated link. We define *barrier slack* as the difference in delay between two virtual links that connect sibling nodes in the topology. When barrier slack is present, the critical path of a virtual tree will be limited by the longer of the virtual links at a given level of the tree. Thus, we can reduce the critical path of the tree by adding a dedicated link to short circuit the longer path and equalize the latencies of the paths to the siblings. Figure 6 demonstrates how barrier bolstering can reduce the depth of a barrier tree by selectively adding dedicated links to equalize the latency of paths to two sibling nodes. This reduction in latency is mostly attributed to the reduced routing cost on the dedicated link.

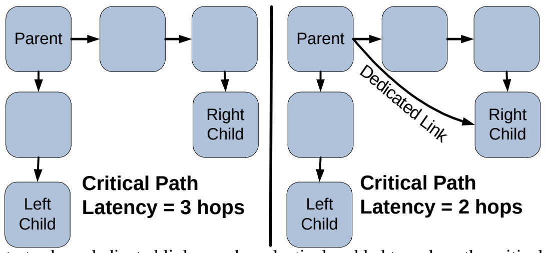

Fig. 6: This figure demonstrates how dedicated links can be selectively added to reduce the critical path of a tree. When two virtual links in the same level of the tree differ in length (in hops), barrier slack exists between the two links. Under certain circumstances, adding a dedicated link can eliminate the slack and equalize the latencies of the links to that of the shorter link, reducing the critical path of the tree network.

3.2 Benefits and Overheads

Barrier bolstering produces a very low latency barrier implementation, with performance close to that of a dedicated network (results in section 6) by selectively adding dedicated links to reduce the depth of the tree network. Different approaches to bolstering affect the number and lengths of dedicated links that are added, which determines the area overhead of the bolstering technique.

Figure 7 compares various approaches to barrier bolstering in terms of their wiring overhead costs. Even if possible to implement with acceptable link latencies, the cost of the first mentioned technique (with single-hop links for every link) approaches the cost of a dedicated network for large number of cores. This cost can be reduced somewhat by realizing that permitting a certain number of two-hop links does not increase the latency of the barrier. However, in our actual implementation of barrier bolstering, the wiring cost remains low (less than 1% die area) even for large number of cores.

3.3 Mapping Considerations for Barrier Bolstering

In the case of barrier bolstering, the virtual to physical mapping algorithm from the previous implementation (Figure 4) is modified to minimize the slack between sibling nodes in the virtual tree. Figure 8 shows the new algorithm, which is used during network design to determine the locations of supplemental dedicated links in the network for optimal global barrier performance. Although the network is optimized for global barriers, semi-global barriers can also achieve good performance, since they can be mapped to optimized subtrees of the global tree. During dynamic mapping, the final

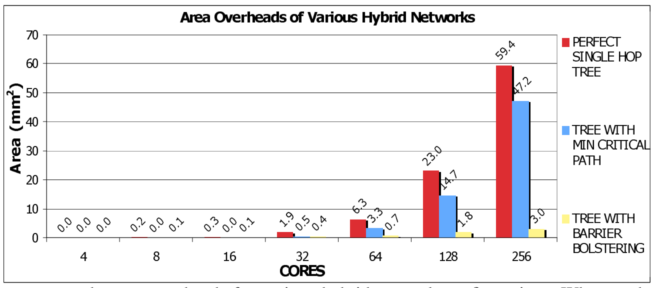

Fig. 7: This figure compares the area overheads for various hybrid network configurations. Whereas the cost of a perfect single-hop tree (where every node is directly connected to its neighbors) approaches that of a dedicated tree for high core integration (compare to Figure 2), the cost of slack elimination via barrier bolstering remains low.

if statement of Figure 8 is ignored, since the locations of dedicated links are statically assigned.

```
NodeList.append(root); hasParent[root] = true
while NodeList not empty do
    POP Node i from NodeList with minimum depth
    select children with: min(max(distance(i,left),distance(i,right))) AND min(|distance(i,left)-distance(i,right)|) AND hasParent(left,right) = false
    NodeList.append(left,right); hasParent[left,right] = true
    if |distance(i,left)-distance(i,right)| < max correctable slack then
        mark longer link as dedicated
    end if
end while
```

Fig. 8: The virtual to physical mapping algorithm for barrier bolstering attempts to minimize slack between siblings. This algorithm is used to select the locations of dedicated links for best global barrier performance.

3.4 Platform Adaptability

Since some dedicated links are used in barrier bolstering, the adaptability of the platform for dynamic thread mapping is somewhat lessened. For semi-global synchronization, the best case is when a thread group can be mapped to a subtree of the originally mapped tree. In this case, the threads receive the full benefits of the bolstering. In the worst case, the virtual to physical mapping for a thread group may not be able to use any of the dedicated links. In this case, the performance of the bolstered network is equivalent to that of the unbolstered virtual network.

4 Reducing Virtual Link Latency with Router Bypassing

While the first technique (hardware-supported mapping of virtual barrier topologies) required no additional link area overhead, the previous technique (barrier bolstering) allowed closer approximation of the performance of a dedicated hardware barrier network. In this section, we discuss a way to get most of both benefits by allowing direct virtual connections instead of physical connections in the case of bolstering.

4.1 Implementation

Using a well-mapped virtual topology in conjunction with barrier state machines at the routing nodes significantly reduces the latency of barrier synchronization. Note, however, that for a virtual link in a virtual tree, there may be multiple hops between successive levels of the tree. This occurrence adds latency to the critical path of the tree, and a significant portion of this latency is due to the packet being routed at multiple routers along its path between tree levels. A recent work suggests the use of *express virtual channels* [16] to mitigate the cost of routing packets that travel multiple hops.

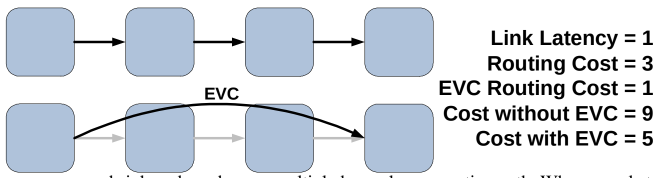

Fig. 9: An EVC has a source and sink node and spans multiple hops along a routing path. When a packet allocates an EVC, it skips the routing stage at intermediate routers and a routing decision only needs to be made again once the packet exits the EVC.

Figure 9 explains the concept of an express virtual channel (EVC). When the downstream destination of a packet is further than one hop away, an EVC may be allocated, spanning all intermediate routers so that the packet can continue on the same virtual channel without being routed at the intermediate nodes. Routing only needs to be performed again when the packet reaches the terminus of the EVC.

We use EVCs as a way to set up virtual connections between nodes that are logical neighbors in the virtual barrier topology. Since the set of routing paths to be accelerated for a given tree mapping are fixed, EVCs can be planned and allocated efficiently according to the state of the configuration registers in the routers. When configured by the runtime for a specific virtual topology, the registers store the location of virtual neighbor nodes, describing the parameters for an EVC between the nodes.

4.2 Benefits and Overheads

Using EVCs to enhance routing between virtual neighbors maintains all the benefits of the virtual network platform and also adds the benefit of reduced routing latency for some multi-hop virtual links. The extent to which this benefit can improve performance depends on how well EVCs can be utilized.

The costs incurred to obtain this additional benefit are increased router complexity to add support for EVCs and potential degradation of other network traffic, since EVCs suppress communication on intermediate routers when they are allocated until they are freed. Use of EVCs does not add any area/power cost in terms of communication links.

4.3 Mapping Considerations

The use of EVCs adds a new dimension to the virtual to physical topology mapping problem. Since EVCs are only allowed to travel along a single routing dimension [16], the selection criteria for choosing the children of a node when forming a virtual tree are somewhat different. Whereas in the original algorithm, children were chosen to minimize the number of hops between successive tree levels, they are now selected to simultaneously minimize both the number of hops and the number of directional changes. This implies a different distance function that accounts for the relative costs of link traversal and routing latency in EVCs. Because the cost of routing is higher than the cost of link traversal, situations arise in which it is more efficient to travel more hops in a single direction than fewer hops in multiple directions.

When an EVC flow passes through a router, any other traffic at that router is suppressed. This situation reveals a tradeoff for EVCs. While they exhibit potential to expedite communication along the EVC path, they also potentially slow down other traffic that uses any of the routers in the EVC path. We account for this by minimizing the occurrence of crossing paths in the same level of the tree whenever possible. This means that if multiple potential children are located at the same distance, children are selected to avoid crossing paths in the same tree level. Figure 10 gives the EVC-aware virtual to physical mapping algorithm.

```
let distance(i,j) = (link latency)·hops(i,j)+(routing latency)·dirChanges(i,j)
NodeList.append(root); hasParent[root] = true
while NodeList not empty do
    POP node i from NodeList with min depth
    select children with: min distance(i,child) AND hasParent(child) = false
    if multiple potential children with min distance then
        select children to minimize crossing paths
    end if
    NodeList.append(children); hasParent[children] = true
end while
```

Fig. 10: The EVC-aware virtual to physical mapping algorithm minimizes directional changes and same-level crossing paths, features that inhibit the utilization or performance of EVCs.

Figure 11 demonstrates that a mapping algorithm that is aware of the tradeoffs inherent in the use of EVCs can produce a different mapping than the normal mapping algorithm, which considers the cost of all hops along a virtual path to be the same, independent of the directional implications.

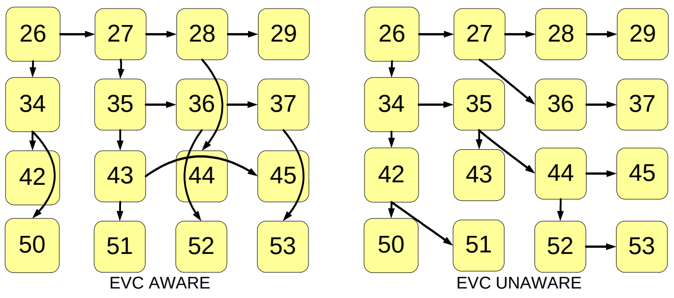

Fig. 11: The figure above compares the same subsection of a 64-core mesh for two mapping strategies – one that considers EVC tradeoffs during the mapping phase (left) and one that does not (right). When EVC tradeoffs are considered, virtual paths that change directions are less desirable.

4.4 Platform Adaptability

Since EVCs are allocated dynamically, not statically like the dedicated links in barrier bolstering, this platform shares all adaptability features of the original virtual network platform.

5 Methodology

To obtain our experimental results, we use a modified version of the M5 simulator [17] that has been adapted to support communication over a network-on-chip (NoC) rather than a shared bus. In our baseline architecture, the NoC has a rectangular mesh topology, with link latency equal to 1 cycle and routing latency equal to 4 cycles. Routers

Clock	2GHz	Mem Latency	300 cyc	Execution	In-order
L1 Icache	32KB	L1 Dcache	64KB	L2	4MB/CORE, 10 cyc

Table 1: Architectural Details.

are pipelined for increased throughput. Table 1 lists some additional details about the simulated architecture.

To model a dedicated hardware barrier network, we add an extra set of physical links to the chip and arrange them in a tree topology. Routing for the dedicated network is simple, deterministic, and suffers no resource contention. Thus, we specify single cycle latency for routing on the dedicated network. Link latencies for the dedicated links are assigned based on the Manhattan distance between the connected nodes.

For virtual barrier configurations, described in section 2, we model routers with supplemental routing logic equivalent to the state diagram of Figure 3. The state machine logic requires only a few latches and gates. For an N-core processor, the additional state required at each router is composed of $3 \cdot logN$ bits to store the indices of the parent, left child, and right child, and 3 bits to track whether arrival notifications have been received from left child, right child, and self.

In section 3, we model the addition of select dedicated links to the regular mesh. All routing latencies remain the same, and link latencies of the dedicated links are determined by the lengths of the links.

In section 4, we assume the availability of express virtual channels (EVCs). Necessary support for EVCs is described in [16]. When an EVC is allocated, we assume normal routing latency at the source and termination of the EVC and routing latency of 1 cycle for intermediate routers.

6 Analysis of Results

In this section, we first reinforce the point made in section 1 that barrier synchronization mechanisms for CMPs need to be different from those for traditional multi-processors. Then we compare the performance of the various CMP-specific barrier implementations presented in this paper. Finally, we discuss the implication of new barrier implementations on performance and design of parallel applications.

6.1 Traditional Barrier Mechanisms in the Context of CMPs

In our revisitation of barrier mechanisms for CMPs, we looked at three categories of software barrier implementations, categorized based on their communication patterns – centralized barriers, decentralized barriers, and hierarchical barriers.

Centralized barriers are most commonly found in cache coherent shared memory systems. In this style of barrier, all participating threads communicate with a central entity to make known their arrival at the barrier. When the centralized entity receives notifications from all threads, it responds in turn by sending barrier completion notifications to the participants. While this type of barrier is simple to implement, obvious performance and scalability detriments are inherent in the design.

Decentralized barrier algorithms differ from centralized algorithms in that all threads determine completion of the barrier locally. Thus, decentralized algorithms perform notification and completion phases in parallel, at the expense of extra communication between participants. An example of a decentralized barrier that we evaluated is a broadcast barrier.

In a **hierarchical barrier** algorithm, each participant synchronizes with some subset of the global nodes and subsequently propagates the local synchronization state to a higher level until global synchronization can be determined. Some example implementations of this type are tree and butterfly barriers [2].

Figure 12 shows the performance of various standard software barrier implementations on multi-core architectures with different number of cores.

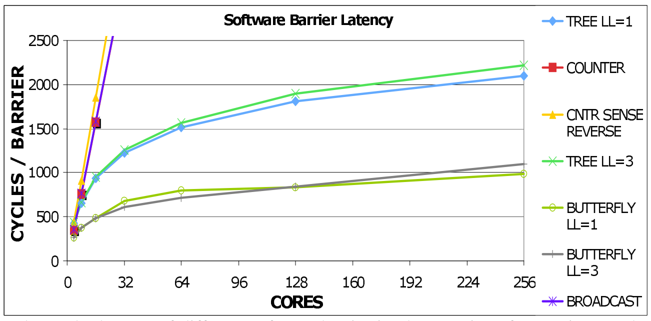

Fig. 12: This figure shows the latency of different software barrier implementations for varying number cores. While the hierarchical software implementations show some promise, the other approaches are constrained by software overheads on the critical path. For implementations that are sensitive to link latency, LL=X denotes the link latency.

Figure 12 demonstrates a few key CMP-specific points. First, there is no difference between the performance of the centralized and decentralized software barriers on the CMP due to the relative expense of barrier management in software as compared to the relatively low cost of communication between cores. Essentially, notifications arrive faster than the software is able to process them.

Another observation is that changing the latency of the communication links for a multi-core architecture has almost no effect on the performance of centralized and decentralized barriers. This further demonstrates that these implementations are constrained by software overhead and shared resource constraints and are unsuitable for use in CMPs. On the other hand, the hierarchical barriers do respond to changing link latency, with a more pronounced effect as the depth of the hierarchy grows. This is because the critical paths of these algorithms depend more directly on link latency. Since notifications do not always queue up for threads, they must spend time waiting for notifications to travel from one level of the hierarchy to the next.

The final observation to be drawn from the figure is that even though some software approaches are not completely dominated by software overhead, the barrier synchronization overhead is still unacceptably high for all software barrier implementations in scenarios where medium to fine grain synchronization is desirable. So, barriers for CMPs should preferably not be implemented in software and should have low latency.

We also investigated using dedicated hardware barrier tree networks.

Figure 13 shows performance scalability for dedicated tree barrier networks. The two curves represent different assumptions about link latencies. The lower curve corresponds to the situation where routing can be done through different metal layers to normalize the link latency. For the upper curve we assume pipelined links where the length determines the number of latches necessary and thus the end-to-end latency of the link. As the latter assumption represents a more realistic scenario, we use this approach for comparison in the rest of our discussion.

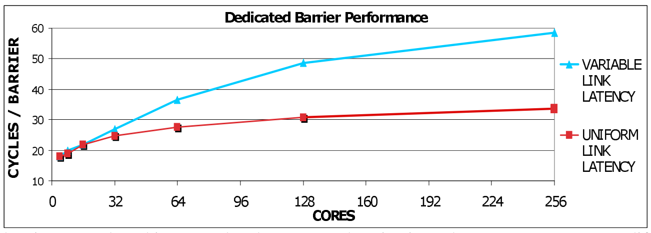

Fig. 13: Dedicated barrier networks achieve very low latency synchronization. The two curves represent different assumptions about the routing and latency determination for network links. In one, uniform link latency is assumed. For the other curve, we assume link latency proportional to link length.

Comparison of Figure 12 and Figure 13 demonstrates that dedicated hardware barrier implementations have much smaller performance overhead than software implementations. In fact, compared to the dedicated barrier tree implementation, the best performing software implementation exhibits up to **22.82x increased latency**. However, dedicated hardware barrier networks have prohibitive area overhead (see section 1). So, new CMP-specific implementations are needed.

6.2 Performance Benefits of CMP-specific Barriers

Figure 14 compares the latencies of the efficient barrier techniques to those of the dedicated hardware barrier network. The results demonstrate that with efficient barrier notifications and minimal support in the NoC, we can achieve close to dedicated performance at a greatly reduced cost.

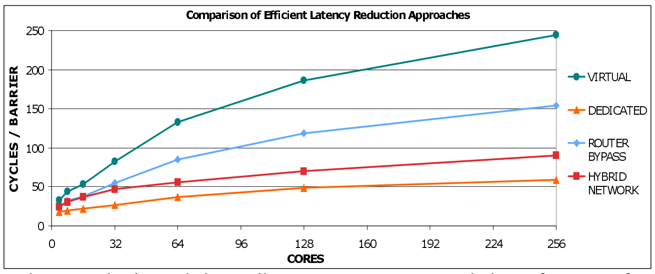

Fig. 14: The efficient latency reduction techniques all represent ways to approach the performance of a dedicated network without paying the associated overhead cost. The hybrid approach comes very close to matching the performance of a dedicated synchronization network with a sizable reduction in overhead.

There are several sources of benefits for each of the three barrier implementations. For example, our first technique (hardware-supported mapping of virtual barrier topologies to physical topologies) uses loads/stores instead of sends/receives. The formula for latency of the tree barrier has the form $X + Y \cdot log_2 N$, where X and Y represent the notification/completion cost and the inter-level traversal latency, respectively. Thus, adding load/store support for barrier notifications reduces latency by cutting down the value of X, as demonstrated in Figure 15.

Similarly, optimizing the virtual to physical mapping for a synchronization network can significantly affect performance. Figure 16 compares the latency of a naively mapped virtual tree, in which children are located at the obvious node indices, to an optimized virtual mapping, in which the children of each node in the tree are selected intelligently based on the algorithm outlined in Figure 4.

Figure 17 shows that if EVCs are employed to expedite routing on virtual links, then it becomes necessary to consider the limitations of virtual channels when deciding on the optimal virtual to physical mapping. Mappings that are unaware of these

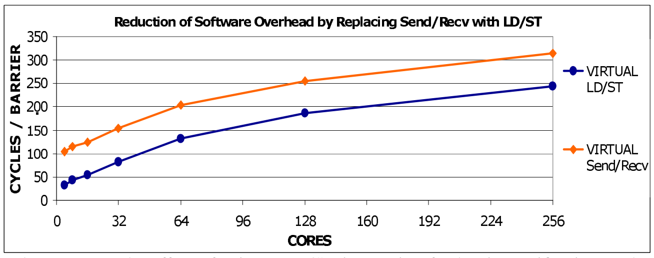

Fig. 15: This figure demonstrates the effect of using a LD/ST instruction for barrier notifications rather than a send/recv operation. Since the barrier notification message does not need the general functionality of a full-fledged send operation, the latency can be reduced considerably. Likewise, latency of completion query can be similarly reduced. The aggregate effect is to shift the latency curve down by a constant offset.

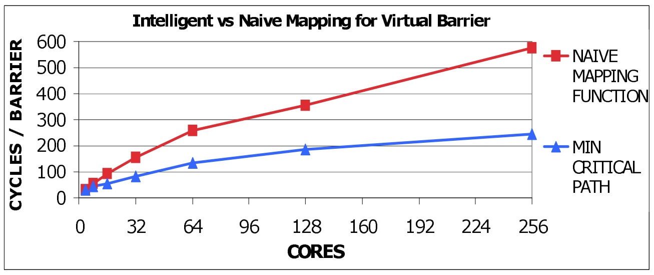

Fig. 16: This figure compares barrier latency for two virtual barrier networks – one that is configured naively, and one that is configured intelligently (e.g. the two configurations of Figure 5). The latency observed by a naive virtual mapping can be considerably greater than that of an intelligently mapped virtual topology.

considerations are not able to make the most efficient use of available EVCs. As Figure 17 demonstrates, the benefit of adding EVC capability to the NoC is small unless an EVC-aware mapping policy is used.

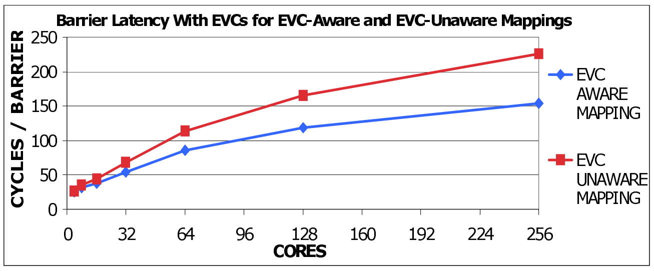

Fig. 17: This figure shows the difference in barrier latency for an EVC-aware mapping strategy and a strategy that does not optimize for best EVC utilization. The results demonstrate that the benefit of adding EVCs is small unless the existence of EVCs is accounted for during the virtual to physical mapping process.

6.3 Implications for Performance and Design of Parallel Applications

To further validate the usefulness of our barrier techniques, we demonstrate how fast barriers enable large-scale CMPs to exploit fine-grained parallelism and achieve speedups on challenging benchmark applications. As in [18], we evaluate the performance of our barrier techniques for two of the Livermore loops [19] and two benchmarks from the EEMBC suite [20]. Figure 18 compares the performance of parallel versions of the Livermore loops that employ software and hardware barrier techniques on a 128-core CMP against the sequential performance for varying vector lengths.

Although the granularity of parallelism is very fine, the efficient barrier techniques allow the large-scale CMP to achieve substantial speedups. For larger granularity of parallelism, software barriers can have benefits, but the benefits are very limited compared to those afforded by our efficient barrier techniques. These results demonstrate the need for low-overhead, CMP-specific barrier techniques.

Figure 19 shows the benefits of efficient barrier approaches for EEMBC Autocorrelation (32 lags, input=xspeech) and Viterbi decoder (input=getti.dat). The Autocorrelation results demonstrate that while software barriers leave performance on the table, virtual and hybrid approaches can nearly achieve the performance of a dedicated synchronization network. For the Viterbi decoder, using a software barrier implementation actually results in a slowdown with respect to sequential, while hybrid approaches achieve modest gains. For these benchmarks, performance variation between our efficient techniques was small. This variation increases with increased application dependence on barriers and increased number of cores performing synchronization.

In a nutshell, using CMP-specific barrier implementations allows existing parallel applications to exploit fine-grained parallelism more effectively. It also allows applications to be parallelized at a finer granularity, potentially resulting in significant application speedups.

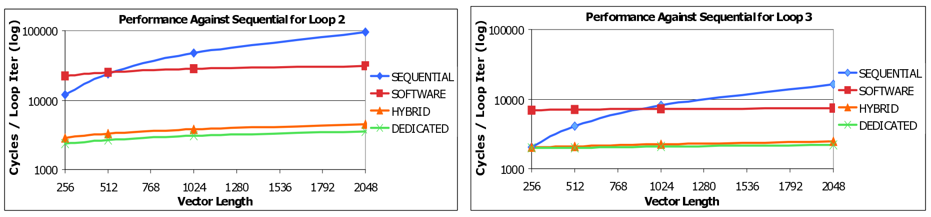

Fig. 18: Performance comparison of software, hybrid, and hardware barrier implementations for Livermore loops 2 and 3. Speedups are relative to sequential execution.

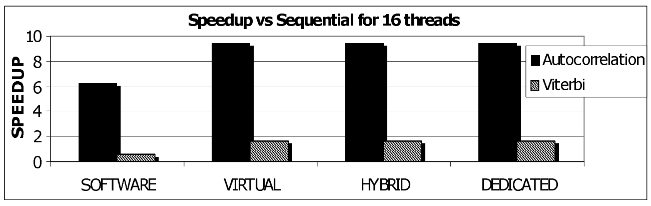

Fig. 19: Performance comparison of software, hybrid, and hardware barrier implementations for EEMBC Autocorrelation and Viterbi decoder. Speedups are relative to sequential execution.

7 Related Work

Barrier synchronization in the context of large scale multiprocessors has been a well-studied problem [1–4]. Several approaches target efficient software algorithms [2, 5–8], but dedicated hardware synchronization networks have also been deployed in some systems. Notably, IBM's Bluegene/L [9] contains multiple interconnect networks, each with a dedicated purpose. Both the global interrupt network and the collective communication networks of BG/L can be used to achieve low latency barrier synchronization [3]. Targeting low latency synchronization, other systems have also used dedicated networks, including the AND-tree barrier synchronization circuits of the Cray T3D [10], the network-supported fetch-and-add approach of the NYU Ultracomputer [11], and the barrier register proposed by Beckmann, et al. [12]. While we evaluate the performance of a dedicated hardware barrier network, we do so in the context of a CMP, where we observe a different set of constraints and design considerations than those found in previous large-scale multiprocessors.

More recent works have looked at the topic of synchronization in CMPs. Zhu, et al. propose a synchronization state buffer [21] for reducing the overhead of fine-grain synchronization support by tracking only actively synchronized data. Sampson, et al., suggest the use of barrier filters [18] that are implemented in the memory controllers of a shared memory processor. These efforts both represent centralized, memory-based approaches, whereas our techniques are inherently decentralized in nature and focus on support in the NoC.

Another recent work [22] evaluates barrier performance in CMPs with the intent of determining how well various barrier algorithms perform on different NoC topologies. This is similar to the way in which we map virtual topologies onto disparate physical topologies, however, the mapping strategies used in [22] are naive, leading to a different set of conclusions. Another Cray multiprocessor, the T3E [23], uses configurable routers, equipped with barrier synchronization units to map a virtual topology onto a separate physical topology.

Our optimization of virtual barrier networks through the use of express virtual channels (EVCs) is based on the work of Kumar, et al. [16], who propose EVCs as a technique for approaching an ideal interconnect fabric for NoCs.

8 Conclusion

In this research, we have revisited the subject of barrier synchronization for many-core CMPs. First, we established that the unique characteristics and constraints of CMPs dictate that software-only barrier implementations perform poorly relative to implementations that utilize a dedicated synchronization network. Then we observed that the overhead of adding a dedicated synchronization network to a chip can be high, especially as core integration continues to increase. Based on these observations, we suggested several techniques that utilize the existing network on chip with slight modifications to allow dramatically increased barrier performance without paying the price of a dedicated network. Our techniques allow us to achieve near-dedicated barrier performance for minimal cost.

References

1. Shang, S., Hwang, K.: Distributed hardwired barrier synchronization for scalable multiprocessor clusters. IEEE Trans. Parallel Distrib. Syst. 6(6), 591-605 (1995)
2. Hoefler, T.: A survey of barrier algorithms for coarse grained supercomputers. Chemnitzer Informatik-Berichte (2004)
3. Almasi, G., et al.: Optimization of MPI collective communication on Bluegene/L systems. ICS '05. 253-262 (2005)
4. Ramakrishnan, V., Scherson, I.D.: Efficient techniques for nested and disjoint barrier synchronization. J. Parallel Distrib. Comput. 58(2), 333-356 (1999)
5. Chen, J., Watson, W.: Software barrier performance on dual quad-core Opterons. NAS '08. 303-309 (2008)
6. Nikolopoulos, D., Papatheodorou, T.: Fast synchronization on scalable cache-coherent multiprocessors using hybrid primitives. IPDPS '00. 711 (2000)
7. Lee, J.B., Jhon, C.S.: Reducing coherence overhead of barrier synchronization in software DSMs. ICS '98. 1-18 (1998)
8. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. 9(1), 21-65 (1991)
9. Coteus, P., et al.: Packaging the BlueGene/L supercomputer. IBM Journal of Research and Development 49(2-3), 213-248 (2005)
10. Adams, D.: Cray T3D system architecture overview manual. ftp://ftp.cray.com/product-info/mpp/T3D_Architecture_Over/T3D.overview.html (1993)
11. Freudenthal, E., Peze, O.: Efficient synchronization algorithms using fetch-and-add on multiple bitfield integers. Ultracomputer Note 148 (1988)
12. Beckmann, C., Polychronopoulos, C.: Fast barrier synchronization hardware. ICS '90. 180-189 (1990)
13. Biswas, R.: NAS parallel benchmarks. http://www.nas.nasa.gov (2009)
14. Kumar, R., Zyuban, V., Tullsen, D.: Interconnections in multi-core architectures: Understanding mechanisms, overheads, and scaling. ISCA '05. (2005)
15. Althaus, E., Funke, S., Har-peled, S., Knemann, J.: Approximating k-hop minimum-spanning trees. Operations Research Letters 33, 120 (2005)
16. Kumar, A., et al.: Express virtual channels: Towards the ideal interconnection fabric. SIGARCH Comput. Archit. News 35(2), 150-161 (2007)
17. Binkert, N.L., et al.: The M5 simulator: Modeling networked systems. MICRO 26(4), 52-60 (2006)
18. Sampson, J., et al.: Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. MICRO 39. 235-246 (2006)
19. McMahon, F.: Livermore loops coded in C. http://www.netlib.org/benchmark/livermorec (1992)
20. E.M.B. Consortium: EEMBC. http://www.eembc.org (2009)
21. Zhu, W., et al.: Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures. ISCA '07. 35-45 (2007)
22. Villa, O., Palermo, G., Silvano, C.: Efficiency and scalability of barrier synchronization on NOC based many-core architectures. CASES '08 81-90 (2008)
23. Scott, S.L.: Synchronization and communication in the T3E multiprocessor. SIGOPS Oper. Syst. Rev. 30(5), 26-36 (1996)