iNPG: Accelerating Critical Section Access with In-Network Packet Generation for NoC Based Many-Cores

Yuan Yao
KTH Royal Institute of Technology
Stockholm, Sweden
yuanyao@kth.se

Zhonghai Lu*
KTH Royal Institute of Technology
Stockholm, Sweden
zhonghai@kth.se

ABSTRACT

As recently studied, serialized competition overhead for entering critical section is more dominant than critical section execution itself in limiting performance of multi-threaded shared variable applications on NoC-based many-cores.

We illustrate that the invalidation-acknowledgement delay for cache coherency between the home node storing the critical section lock and the cores running competing threads is the leading factor to high competition overhead in lock spinning, which is realized in various spin-lock primitives (such as the ticket lock, ABQL, MCS lock, *etc.*) and the spinning phase of queue spin-lock (QSL) in advanced operating systems. To reduce such high lock coherence overhead, we propose in-network packet generation (iNPG) to turn passive "normal" NoC routers which only transmit packets into active "big" ones that can generate packets. Instead of performing all coherence maintenance at the home node, big routers which are deployed nearer to competing threads can generate packets to perform early invalidation-acknowledgement for failing threads before their requests reach the home node, shortening the protocol round-trip delay and thus significantly reducing competition overhead in various locking primitives.

We evaluate iNPG in Gem5 using PARSEC and SPEC OMP2012 programs with five different locking primitives. Compared to a state-of-the-art technique accelerating critical section access, experimental results show that iNPG can effectively reduce lock coherence overhead, expediting critical section access by 1.35× on average and 2.03× at maximum and consequently improving the program Region-of-Interest (ROI) runtime by 7.8% on average and 14.7% at maximum.

1. INTRODUCTION

As the core count grows quickly, there is a greater potential to speed up the execution of parallel and concurrent programs. While this trend helps to linearly expedite the concurrent execution part, the ultimate application speedup is however limited by the sequential execution part of programs, as reflected in the well-known Amdahl's law [20].

Figure 1 shows the typical execution of a multi-threaded program. After an initialization phase, N threads start parallel executions and encounter a synchronization point where each thread competes to access and execute a critical section (CS). In particular, serialized CS access incurs *competition*

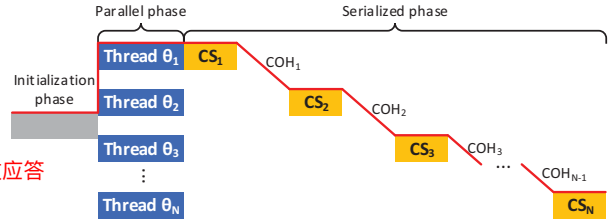


Figure 1: Parallel program execution.

overhead (COH) due to locking retries in spin-lock primitives [31, 2, 16, 14] and context switching & sleep overhead in queue spin-lock primitive (See details in Section 2). To expedite parallel application execution, most previous works emphasize on accelerating the CS parts shown in Figure 1 by exploiting various mechanisms such as running the CS code on fat cores in asymmetric chip multi-processors (CMPs) [36, 21, 27, 38, 37], or predicting and prioritizing threads that are executing critical sections [9, 39, 24]. Recent research in [40] demonstrates that COH may indeed exceed the time for CS execution in network-on-chip (NoC) based many-cores, and become the dominating factor limiting the performance of parallel applications. By opportunistically avoiding the OS-level context switching & sleep overhead, the proposed OCOR (Opportunistic Competition Overhead Reduction) technique in [40] shows great COH reduction (over 30% on average for PARSEC [3] and SPEC OMP2012 [28] benchmarks) and application acceleration up to 24.5%. However, this work uses only queue spin-lock for experiments and does not investigate the effect of cache coherence in influencing COH in cache-coherent many-cores.

In the paper, we show that the lock coherence overhead (LCO) in lock spinning for both spin-lock primitives and the spinning phase of queue spin-lock primitive is a major source of COH. This is because, in lock spinning, each competing thread will generate an exclusive access request to the home node where the lock variable is stored. However, only one of these exclusive accesses will succeed, and the home node will then invalidate the CS lock copies in all the other threads' L1 caches. Only when the winning thread, which is the new owner of the lock variable, receives all acknowledgements from the losing threads, it can proceed to execute the following critical section. To reduce such high overhead coherence round-trip latency, we propose an *in-network packet generation* (iNPG) technique that can perform early cache invalidation to locking fail-

*Zhonghai Lu is the corresponding author.

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00012

ure threads, thereby reducing LCO and accelerating execution of multi-threaded applications. iNPG is enabled by active "big" routers, which can not only transmit packets like normal routers but also generate packets in the network as early invalidations to L1 caches of failing threads. The big routers then wait for the acknowledgements from the losing threads, and then forward the acknowledgements to the winning thread to send a valid lock copy to each of the losing thread. In this way, the home node is largely freed from the lock coherence maintenance burden, and the winning thread can move forward to the CS more quickly. Focusing on lock spinning, iNPG is an orthogonal scheme to OCOR suggested for queue spin-lock in [40]. As such, it can be combined seamlessly with OCOR to achieve the highest COH reduction with various spin-lock primitives and the queue spin-lock primitive, as shown in experimental results.

The rest of the paper is organized as follows. Section 2 gives background and motivation. Section 3 illustrates lock spinning in cache coherent architecture and describes our design principle. Section 4 gives design and synthesis details of big router. In Section 5 we report experiments and results. After discussing the related work in Section 6, we finally conclude in Section 7.

2. BACKGROUND AND MOTIVATION

2.1 Lock spinning schemes

In lock spinning, when a thread fails to lock a variable protecting a CS, it will keep re-trying (after a short spin interval) until succeed [33]. During the OS runtime quantum (usually in the order of hundreds of milliseconds) of the spinning thread, no other threads are allowed to use the core, leading to a waste of core processing time and energy. However, by persistently polling on a lock variable, lock spinning can be quickly successful once the lock is released. Modern OSes provide different lock spinning mechanisms, including spin-locks and queue spin-lock, to support critical section synchronization for concurrent programs.

1) Test-and-set lock (TAS). The test-and-set lock is a spin-lock that employs a polling loop to access a global boolean flag that indicates whether the critical section is held. Each core repeatedly executes an atomic `test_and_set` instruction in an attempt to change the flag from false to true, thereby acquiring the lock. The shortcoming of the test-and-set lock is the contention for the lock, as each waiting core continually spins the single shared lock until success.

2) The ticket lock (TTL). As proposed in [31], the ticket lock is a spin-lock that consists of two counters, one request counter containing the number of cores to acquire the lock, and one release counter recording the number of cores that have released the lock. A core competes for the lock by fetching a *ticket* from the request counter, and waiting until its ticket is equal to the release counter. A core releases the lock by incrementing the release counter.

3) Array-based queuing lock (ABQL). By having all competing cores spinning on different lock variables, array-based queuing lock is a spin-lock ensuring that on a lock release only one core attempts to acquire the lock. In this way, ABQL significantly decreases the number of lock contentions when a lock is released. ABQL has two versions, which are corespondingly proposed by Anderson [2] and Graunke and Thakkar [16], in which the Anderson's version further protects the ABQL with an atomic `test_and_set`.

4) Mellor-Crummey and Scott (MCS) lock. As inspired by [14], Mellor-Crummey and Scott [26] propose a per-core structured spin-lock, which is able to eliminate much of the cache-line bouncing experienced by spin-locks. In MCS lock, when a core attempts to secure a global lock variable, it will create an `mcs_spin_lock` structure of its own. Using an atomic exchange operation (`compare_and_swap`), it stores the address of its own `mcs_spin_lock` structure into the global lock's "next pointer" field. The atomic exchange operation will then return the previous value of the next pointer. If that pointer was null, the acquiring core is successful in acquiring the lock. Otherwise, the core is put into a queue waiting for its predecessor to release the lock.

5) Queue spin-lock (QSL). QSL is a variant of spin-lock which is adopted in most modern OSes such as Linux 4.2[1] and Unix BSD 4.4[2]. It starts with a *spin-lock* phase to secure a CS, and if not successful after a certain times of retry (by default 128 times in Linux 4.2), it yields to enter a *sleep* phase after context switching (thus releasing the core for processing other tasks or in power-saving mode) and the thread's locking request is placed in a request queue. The thread continues sleeping until being woken up to secure the lock when the CS is unlocked by the holding thread. The spin-lock phase of a queue spin-lock can be one of the four spin-lock alternatives (TAS, TTL, ABQL, or MCS) discussed above. Because of its advantages in reducing synchronization traffic and OS overhead, we select QSL in the paper with the spin-lock implemented as MCS lock.

2.2 Motivation

To show the criticality of lock coherence overhead (LCO) in different locking primitives, we experimented on a 64-core architecture in Gem5 (See experimental setup in Section 5), with each program running alone in 64 concurrent threads. Figures 2 reports the percentage of LCO in application running time under TAS, TTL, ABQL, MCS and QSL in three applications: *kdtree* from SPEC OMP2012 and *facesim*, *fliudanimate* from PARSEC.

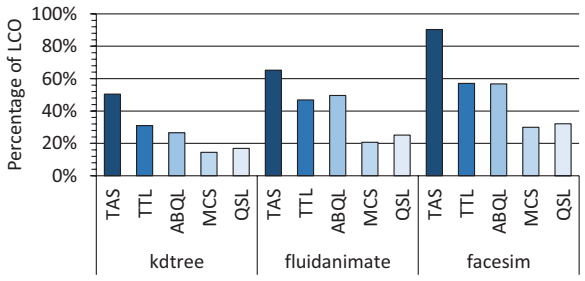


Figure 2: Percentage of LCO in application running time.

From Figure 2 we can observe that 1) across different benchmarks, the percentage of LCO in application running time differs. 2) TAS lock has the most percentage of LCO,

[1]https://kernelnewbies.org/Linux_4.2
[2]http://www.unixdownload.net/

followed by TTL and ABQL locks; LCO percentage in MCS and QSL locks are relatively smaller among all locking primitives. As shown in the figure, in benchmark *ketree*, TAS spends nearly 50% of application running time synchronizing critical section lock among competing threads. With TTL and ABQL, the percentage of LCO is reduced to 31% and 27%, correspondingly. In MCS and QSL, because locking contention among threads is further minimized, the percentage of LCO reaches 14% and 17% application running time, correspondingly. The same phenomenon has been consistently observed in *fluidanimate* and *facesim*, where LCO occupies 65% and 90% application running time under TAS, 47% and 57% under TTL, 50% and 56% under ABQL, 20% and 30% under MCS, and 25% and 32% under QSL, respectively. From the experiment we can conclude that LCO indeed lies straightly on application execution critical path, causing heavy overhead for lock spinning.

3. IN-NETWORK PACKET GENERATION

3.1 Target many-core architecture

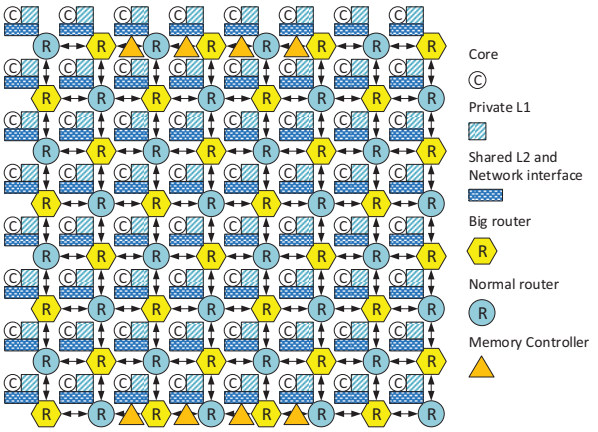


Figure 3: A 64-core 8×8 many-core architecture.

Figure 3 shows a typical architecture for a 64-core 8×8 many-core, where the NoC (routers and network interfaces) communicates messages among processor nodes (a core with its private L1 cache), secondary on-chip shared L2 cache banks, and on-chip memory controllers. Routers are organized in a popular mesh topology on which the XY dimensional routing algorithm is implemented to achieve simplicity and deadlock-free. Eight memory controllers are symmetrically connected to the middle nodes on the top and bottom rows. Due to its layout and packaging conveniences, this pattern has been used in academic and industrial many-core designs such as MIT SCORPIO 36-core processor [7] and latest Intel Knights Landing 36-tile processor [34]. To support cache coherency, a directory based MOESI cache coherence protocol is implemented. On the target many-core, once a core issues a memory request, the private L1 is first checked for the presence of the requested data. If this fails, the request is then forwarded to the local shared L2 or via the NoC to remote shared L2/DRAM. Figure 3 also depicts a big router deployment with 32 big routers interleaving with 32 normal routers.

3.2 Lock spinning under cache coherency

We look into how lock spinning works and how its associated LCO is caused on NoC based many-cores with cache coherence protocol.

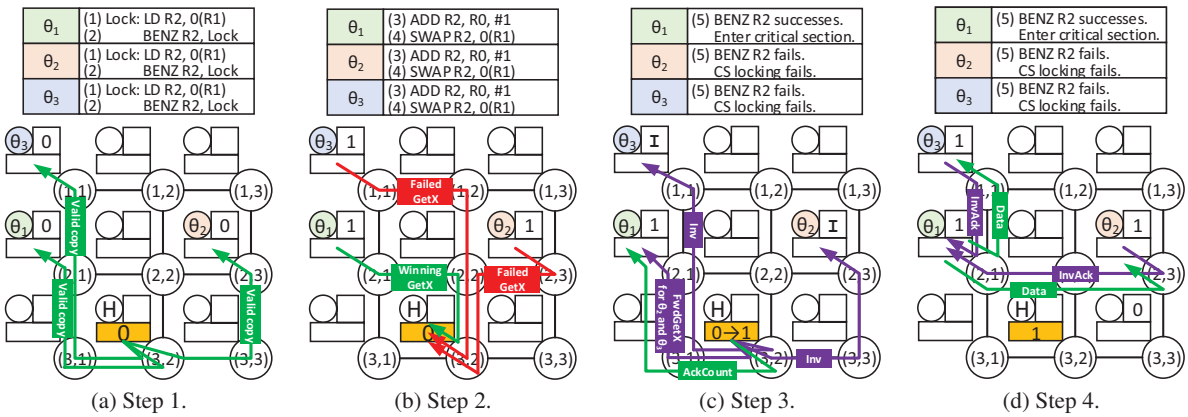
Algorithm 1 shows an example assembly code [19] for a simple test-and-set spin-lock implementation on many-cores. In the code, value 0 denotes a lock "available" and value 1 "occupied". In Line 1, a thread first spins to check on a local copy of the lock variable (whose memory address is stored in local register R1 and the lock itself is stored remotely at a home node) until it sees that the lock is available. Once a thread finds that the lock is freed (by loading 0 into register R2), it then passes Line 2. In Line 3, a thread first modifies the local lock copy from 0 to 1 to assert its ownership for the lock (but the lock variable at the home node still remains as 0). Then, in Line 4, threads compete with each other to secure the lock by sending an atomic swap request (`SWAP`) to the home node, which writes the locally modified lock (with value 1) to the one in the home node (with value 0). The winning `SWAP` request will then set the lock variable at the home node to 1 to assert its exclusive access right to the lock. Consequently, all loser `SWAP` requests will only see 1 in the lock variable which was set by the winner request. After the swap operation, the winning thread passes Line 5 to execute the critical section, with all loser threads spinning for the lock again from Line 1.

Figure 4 shows how Algorithm 1 is executed in three threads θ_1, θ_2 and θ_3 on an example 3×3 CMP based on the MOESI coherence protocol introduced in [35]. The lock variable in the home node is initialized to the "available" state (value 0).

Step 1: When θ_1, θ_2 and θ_3 execute Line 1 of Algorithm 1, because none of them has a copy of the lock in their local cache, they all encounter a cold cache read miss. Each thread then sends a data read request to the home node, which in-turn sends a valid copy of the lock (value 0) back to the corresponding L1 cache of each thread. At the same time, the home node marks θ_1, θ_2 and θ_3 as data sharers of the lock variable in its coherence directory.

Step 2: When the competing threads load 0 into their locally copied CS locks, they pass Line 2. In Line 3, each thread locally modifies the lock to the occupied state by changing its value to 1. Then, the three threads compete with each other by executing the swap operation (`SWAP` in Line 4). At the hardware level, this is done by each thread sending an atomic "read-for-modification" operation (`GetX` request in Gem5), which tries to get the exclusive access right to the lock in the home node. The winning thread will then execute the swap operation and write 1 into the lock variable.

Step 3: In Figure 4, assuming that `GetX` from θ_1 succeeds, and `GetX` requests from θ_2 and θ_3 fail. In this case, θ_1 wins the exclusive access right to the lock. The directory controller at the home node will perform three tasks. First, it notifies θ_1 that it now has the exclusive access right of the lock. Second, it searches its coherence directory and sends invalidations to the lock copies in θ_2's and θ_3's caches. θ_2 and θ_3, upon receiving the Invalidation, will then each send an Invalidation-Acknowledgement (`InvAck` in Gem5) to θ_1, which is now the owner of the lock variable. Third, it forwards the `GetX` requests from θ_2 and θ_3 to θ_1. Moreover, with the forwarded `GetX` requests, the directory controller at



(a) Step 1. (b) Step 2. (c) Step 3. (d) Step 4.

Figure 4: Spin-lock execution on an example 3×3 CMP under MOESI. In **Step 1**, the home node responds to read misses from θ_1, θ_2 and θ_3 and sends a valid lock copy to the local cache of each thread. In **Step 2**, each thread individually modifies the lock to the "occupied" status (value 1) and competes with each other to get the exclusive access right of the lock. In **Step 3**, the winning thread gets exclusive access to the lock, with all losing threads' lock copies invalidated. `GetX` requests from θ_2 and θ_3 are further forwarded by the directory controller of the home node to θ_1. In **Step 4**, θ_1 sends valid lock copies to θ_2 and θ_3 and then enters critical section after collecting all acknowledgements from θ_2 and θ_3.

Algorithm 1 Assembly code for lock spinning

```
1: Lock: LD R2, 0(R1)     ;Load of lock to R2
2:       BENZ R2, Lock    ;Lock unavailable, spin
3:       ADD R2, R0, #1   ;Change lock value to 1
4:       SWAP R2, 0(R1)   ;Swap
5:       BENZ R2, Lock    ;Loop to line 1 if seeing lock 1
```

the home node also forwards the total number of data sharers (`AckCount` in Gem5) of the lock variable to θ_1, so that θ_1 can monitor and count the receiving of `InvAck` messages.

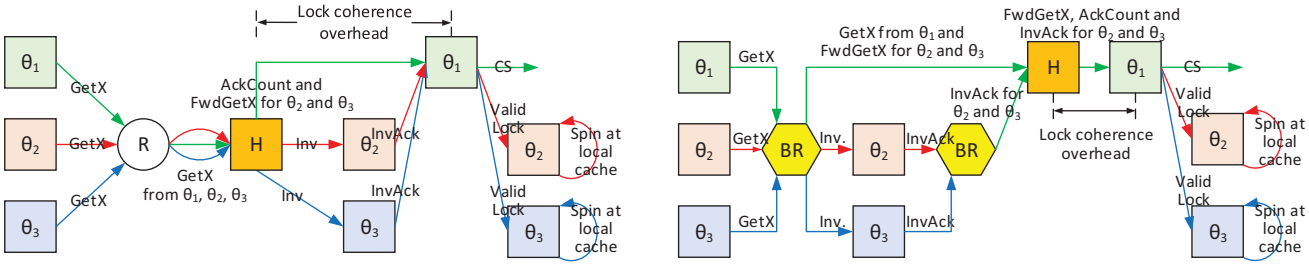
Step 4: After receiving the forwarded `GetX` requests and all of the `InvAck` messages from θ_2 and θ_3, θ_1 sends a valid copy of the lock to the L1 caches of θ_2 and θ_3 and then knows that there are no longer any invalid copy of the lock variable. Thus θ_1 can write to the lock without violating coherence. As a result, θ_1 will successfully execute the `SWAP` operation and executes the following critical section, while θ_2 and θ_3 will again loop back to Line 1. After θ_1 finishes its critical section, it releases the lock by changing the lock value from occupied (value 1) to available (value 0) in the home node.

3.3 Reduce lock coherence overhead with iNPG

Figure 5a depicts the transactional procedure for Step 2, 3 and 4 in Figure 4. When the three competing threads θ_1, θ_2 and θ_3 execute spin-lock, three atomic swaps (corresponding to three `GetX` requests in Gem5) are issued, which are routed through NoC routers (denoted R in Figure 5a) to the home node. The winning `GetX` will exclusively secure the lock. Directory controller in the home node then invalidates the lock copies in θ_2's and θ_3's L1 caches, and forwards the `GetX` requests from θ_2 and θ_3 to θ_1, which is now the owner of the lock. When θ_1 receives the forwarded `GetX` requests and all `InvAcks` from θ_2 and θ_3, it then sends valid lock copies to θ_2's and θ_3's L1 caches and moves forward to execute codes in the critical section. From figure 5a, we can observe that the coherence traffic between the home node and θ_1, θ_2, θ_3 lies on the critical path of program execution time. As shown in the motivational Figure 2, this *lock coherence overhead* (LCO) is a significant part of application running time because the home node is solely responsible for sending invalidations to all failing threads and θ_1 has to collect all the invalidation acknowledgements to move forward.

We aim to shorten the lock coherence latency, *i.e.*, LCO, so as to reduce COH [40] in lock spinning. To this end, we propose in-network packet generation (iNPG) enabled by "big" router (denoted BR in Figure 5b), a normal router enhanced with active packet generation functionality for maintaining cache coherency. This is possible because intermediate routers can be used to provide a temporary "barrier" for a lock variable. Once an intermediate router transfers a `GetX` request for lock variable ℓ (denoted `GetX[`ℓ`]`), a temporary "barrier" is set up for lock ℓ, which will stop delivering failing `GetX[`ℓ`]` requests that lose arbitration to the delivered `GetX[`ℓ`]`, or subsequent `GetX[`ℓ`]` requests that arrive later at the big router. Since the "barriers" are set up distributively at intermediate routers instead of aggregated at the home node, multiple early invalidations can be generated and sent by intermediate big routers to shorten LCO. More details are presented in Section 4.

The principle of iNPG can be illustrated in Figure 5b. When the losing or later arriving `GetX` operations reach a big router, their further transmissions are stopped. Instead, cache invalidations are immediately generated and sent by the big router to their issuing threads (θ_2 and θ_3 in Figure 5). The big router then forwards the `GetX` requests and the acknowledgements (`InvAcks`) from the failing threads to the home node, which are in turn forwarded by the home node to the winning thread (θ_1 in Figure 5). In this way, the invalidation and acknowledgement of the lock copy are done on the way to the home node instead of being done after the `GetX` requests of the losing threads reach the home node. Note that if the winning `GetX` request is determined at the home node (*e.g.*, two `GetX` requests arrive at the home node at the same time but from different inports), the home node is then responsible for sending invalidation to and waiting



(a) Conventional spin-lock transactional procedure.

(b) Big router enabled spin-lock transactional procedure.

Figure 5: Big router enabled iNPG aiming to shorten lock coherence overhead (LCO) in lock spinning.

for acknowledgement from the thread that loses the local arbitration at the home node, just as in the original operation.

4. BIG ROUTER DESIGN

4.1 Micro-architecture

The idea of iNPG for early cache coherence completion is realized by "big" router, which enhances a normal NoC router with packet-generation functionality. Figure 6 shows the micro-architecture of a big router, in which the packet generator is added to maintain a *locking barrier table* and a *packet generation logic* to generate cache coherence packets in the network. Our baseline router is a 2-stage pipelined speculative router [29] in which the first stage performs Route Computation (RC), Virtual Channel (VC) Allocation (VA), and Switch Allocation (SA) in parallel and the second stage is Switch Traversal (ST). In implementation, packet generation is realized in the same pipeline stage as ST.

In a big router, the VC allocator interacts with the packet generator as follows. (1) On detecting the first `GetX` lock request transferred at a big router, the VC allocator informs the packet generator to create a temporary "lock barrier" in the locking barrier table. (2) Once a lock barrier is created, subsequent or arbitration-failed `GetX` requests for the same lock will be stopped. As shown in Figure 6, for every stopped `GetX` request, an early invalidation (EI) entry is created under the corresponding lock barrier to track the status of executing early invalidation. When a `GetX` is stopped, an `Inv` packet will be generated, stored to a separate VC, and sent to the corresponding thread. Meanwhile, the VC allocator changes the failed GetX request to a FwdGetX request, which is forwarded to the home node. (3) Once receiving the `InvAck` packet for an early `Inv` packet that was sent out by the big router, the big router forwards the received `InvAck` to the home node, which further sends it to the lock owner.

Figure 6 sketches the locking barrier table. As shown in the figure, each lock barrier entry contains the memory address of the lock variable and its time-to-live (TTL) duration, which is by default set to 128 cycles. The purpose of TTL is to delete its corresponding lock barrier when it counts down to zero. The TTL counts down only if there is no EI entry and resets to the default value whenever an EI entry is created. Each EI entry contains the issuing core's ID of a stopped `GetX` request and consists of 4 phases: `Inv` generated (denoted *Inv*), `GetX` forwarded (denoted *GetXFwd*), `InvAck` for an early `Inv` received (denoted *InvAck*), and `InvAck` forwarded (denoted *AckFwd*). An EI entry is freed

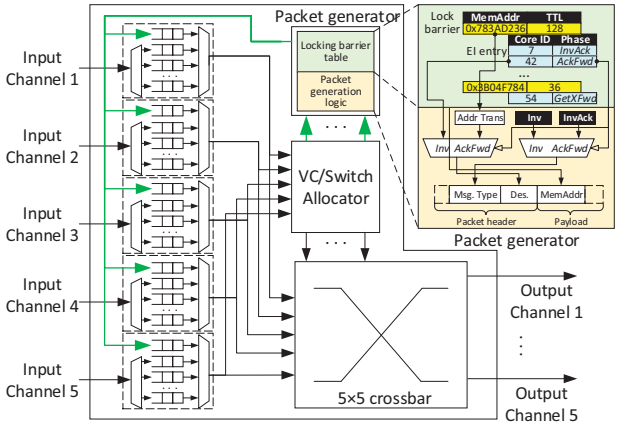


Figure 6: Big router architecture.

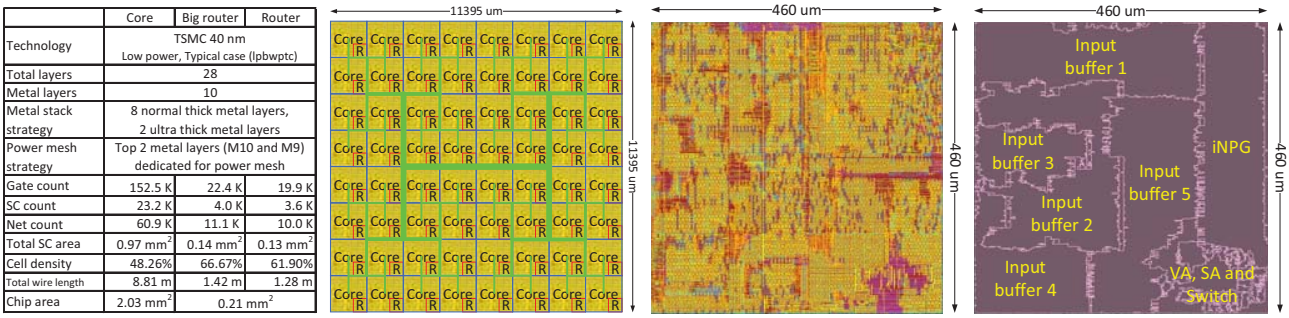
when all the four phases are finished. Further, when all EI entries of a lock barrier are released, its TTL starts countdown from the default value. Finally, a lock barrier entry is deleted after its TTL reaches 0.

Figure 6 also shows the format of generated packets. We omit the information fields that are used for flow control in all network packets (such as flit type, VC identifier, *etc.*), and focus on the additional information fields that are generated. As shown in the figure, the packet generator generates packet in the *Inv* phase (generates invalidation packet) and the *AckFwd* phase (changes the destination of the received `InvAck` message to the ID of the home node). When the locking barrier table is full, following `GetX` requests will pass through as in a normal router.

4.2 Synthesis and chip floorplan & layout

Methodology. We have made RTL implementations of the normal and big routers. We perform RTL synthesis in Synopsys Design Compiler and physical floorplanning & layout in Cadence SoC Encounter using TSMC's 40 nm low power library (typical case). During synthesis, the core/uncore voltage is set to 1.1V and frequency to 2.0GHz. During floorplanning, we use 1.7V as the chip input voltage, which drives all I/O pins of the chip and is scaled down to 1.1V for core/un-core voltage. We select OpenRISC 1200[3] (OR1200) open source CPU as the core model, whose configurations are adjusted according to our architectural setup in Table 1. Integrating 32 normal and 32 big routers to 64 OpenRISC cores, we build a 64-core chip following Figure 3.

[3]https://openrisc.io/implementations



	Core	Big router	Router
Technology	TSMC 40 nm Low power, Typical case (lpbwptc)		
Total layers	28		
Metal layers	10		
Metal stack strategy	8 normal thick metal layers, 2 ultra thick metal layers		
Power mesh strategy	Top 2 metal layers (M10 and M9) dedicated for power mesh		
Gate count	152.5 K	22.4 K	19.9 K
SC count	23.2 K	4.0 K	3.6 K
Net count	60.9 K	11.1 K	10.0 K
Total SC area	0.97 mm^2	0.14 mm^2	0.13 mm^2
Cell density	48.26%	66.67%	61.90%
Total wire length	8.81 m	1.42 m	1.28 m
Chip area	2.03 mm^2	0.21 mm^2	

(a) Module synthesis and layout (b) Chip floorplan with clock tree (c) Big/normal router dimension (d) Big router components

Figure 7: Synthesis and physical floorplan & layout results for one big/normal router and the whole 64-core chip.

Table 1: Simulation platform configurations.

Item	Amount	Description
Core	64 cores	Alpha 2.0 GHz out-of-order cores. 32-entry instruction queue, 64-entry load queue, 64-entry store queue, 128-entry reorder buffer.
L1-Cache	64 banks	Private, 32 KB per-core, 4-way set associative, 128 B block size, 2-cycle latency, split I/D caches, 32 MSHRs.
L2-Cache	64 banks	Chip-wide shared, 1 MB per-bank, 16-way set associative, 128 B block size, 6-cycle latency, 32 MSHRs.
Memory	8 ranks	4 GB DRAM, 512 MB per-rank, up to 16 outstanding requests for each processor, 8 memory controllers.
NoC	64 nodes	8×8 mesh network. Each node consists of 1 router, 1 network interface (NI), 1 core, 1 private L1 cache, and 1 shared L2 cache. X-Y dimensional order routing. Router is 2-stage pipelined, 6 VCs per port, 4 flits per VC, 4 virtual networks. 128-bit data path. Directory based MOESI. One cache block consists of 1 8-flit packet. One coherence control message consists of 1 single-flit packet.
OCOR [40]	-	128 retry times in the spinning phase. 9 priority levels, 8 higher levels for requests in the spinning phase; each priority level is mapped to 16 retry times; 1 lowest priority level for wakeup requests.
iNPG	-	Unless otherwise specified, 32 normal routers and 32 big routers, where one big router is deployed between every two normal routers. 16-entry locking barrier table in a big router.

Synthesis results. As shown in Figure 7a, a normal router consumes 19.9K (equivalent NAND) gates while a big router consumes 22.4K gates. A packet generator consumes 2.5K gates with the majority coming from the locking barrier table which by default contains 16 lock barriers and 16 EI entries. The added packet generator does not significantly increase the critical path inside a big router. Under 2.0GHz timing constraint, we observed 2.9% of all end-to-end paths violating the endpoint slack, which are all eliminated via placement optimization during floorplanning. A packet generator consumes 8.4 mW dynamic power, adding 9.9% overhead to a normal router. On the target architecture, one big tile (1 core plus 1 big router) consumes 716.1 mW dynamic power, with the core consuming 623.5 mW and the big router 92.6 mW. One normal tile (1 core plus 1 normal router) consumes 707.7 mW dynamic power, with the core power the same as in a big tile but the router power decreasing to 84.2 mW.

Floorplan & layout results. As summarized in Figure 7a, the number of floorplan layers is 28, which includes 10 metal routing layers, 11 via layers, 5 implant layers, 1 master-slice layer and 1 AP layer. In the 10 metal layers, the top 2 layers (layers M10 and M9) are dedicated for power mesh, which are featured with ultra thick wires. As shown in Figure 7b, on the target architecture, each tile (both big tile and normal tile) consumes equal chip area, with the distance between neighbour tiles being 1.8 um to accommodate the 256-bit bi-directional wires between routers (128-bit per direction with 1-bit wire width 0.007 um). Further, to implement both big and normal routers with practical physical layout, we manipulate the standard cell (SC) density to accommodate both big and normal router within the same dimension, which is 460 um by 460 um (0.21 mm^2) as shown in Figure 7c. The cell density (before filler insertion) is 66.67% in a big router while 61.90% in normal router, as reported in Figure 7a.

5. EXPERIMENTS

5.1 Methodology

We evaluate **iNPG** with timing-detailed full-system simulations using both PARSEC (10 programs [4][5] with all large input sets) [3] and SPEC OMP2012 (all 14 programs) [28] benchmarks. For data validity, results are obtained from the multi-threaded execution phase called Region-of-Interest (ROI). We implement and integrate our design in Gem5 [4], in which the embedded network GARNET [1] is enhanced according to our technique. The key configurations of the simulation platform are listed in Table 1. Each application runs on 64 cores with one core running one thread. The operating system is Linux 4.2. Unless otherwise specified, the synchronization primitive is the default queue spin-lock in Linux 4.2 with the spin-lock phase implemented as MCS lock. We set up four comparative cases as follows.

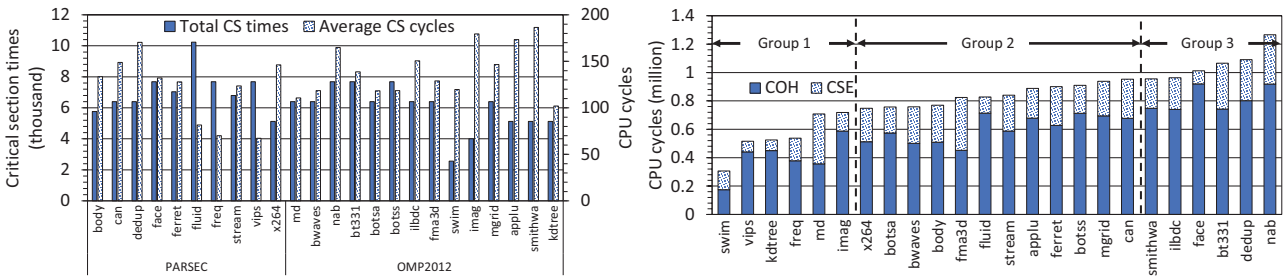
Case 1 is *Original*, the baseline architecture that Gem5 simulates according to the configurations in Table 1.

Case 2 is *OCOR* (Opportunistic Competition Overhead Reduction), the technique proposed for queue spin-lock in [40], which maximizes the chance that a thread gets CS in low-overhead lock-spinning phase and minimizes the chance that a thread gets CS during the high-overhead sleep phase.

OCOR is realized by software and hardware co-design. In software, *OCOR* monitors the *remaining times of retry* (RTR) in a thread's spinning phase, which reflects in how long the thread must enter the high-overhead sleep phase.

[4]We excluded *blackscholes* and *swaptions*, as the former contains merely barrier synchronization and the latter only one CS.

[5]For legibility, we use short names to denote applications with long names, in which *body* is short for *bodytrack*, *can* for *canneal*, *face* for *facesim*, *fluid* for *fluidanimate*, *freq* for *freqmine*, and *stream* for *streamcluster*.



(a) Total CS times and the average CPU cycles of each CS.

(b) Total CS CPU cycles breakdown.

Figure 8: Total CS times, average CPU cycle of each CS, and total CS execution time breakdown.

In hardware, *OCOR* integrates the RTR information into the packets of `SWAP` requests, and let the NoC routers prioritize `SWAP` request packets according to the RTR information. The principle is that the smaller RTR a `SWAP` request packet carries, the higher priority it gets and thus quicker delivery. However, when a thread is already in the sleep mode, it should access CS at a later time, since waking the thread up will introduce considerable overhead. It is thus preferable to allow threads in the low-overhead spinning phase to win. Besides, program progress information is embedded in request packets to avoid starvation for low-priority requests. Following [40], *OCOR* configuration is shown in Table 1.

Case 3 is our *iNPG*. In *iNPG*, we follow the NoC architecture introduced in Figure 3, in which a big router interleaves with a normal router.

Case 4 is *iNPG+OCOR*, which combines *iNPG* with *OCOR*. In this case, all routers support OCOR.

5.2 Experimental results

5.2.1 Benchmark CS characteristics

Figure 8 reports the CS characteristics of the 10 PARSEC and the 14 OMP2012 programs on the target 64-core many-core. Figure 8a gives the total CS access times and average CPU cycles per CS for each program. Since tasks in each program vary, both metrics differ from program to program. For example, in application *fluid* of PARSEC, critical sections (secured by `pthread_mutex_lock`) are used to synchronize indexes of liquid particles. Although each CS takes limited CPU cycles to finish (average 81.47 CPU cycles), the total number of critical sections is however high (10,240 times in total). In contrast, in application *imag* of OMP2012, critical sections (secured by `#pragma omp critical`) are used to atomically modify an image. Although the total number of critical sections is smaller (4,000 times in total), each CS performs relatively heavy tasks and thus takes more CPU cycles (average 179 CPU cycles) to finish.

Figure 8b breaks the total CS execution time (CS times $\times$ average cycles per CS) of each application into competition overhead (COH) and the CS execution time itself (CSE). It is evident that compared to CSE, COH contributes more significantly to application runtime. In the figure, we sort applications according to the total CS execution time (COH+CSE) in the ascending order and divide them into three groups, with Group 1 (6 programs) featuring lower, Group 2 (12 programs) medium, and Group 3 (6 programs) higher total CS execution time.

5.2.2 Application execution timing profile

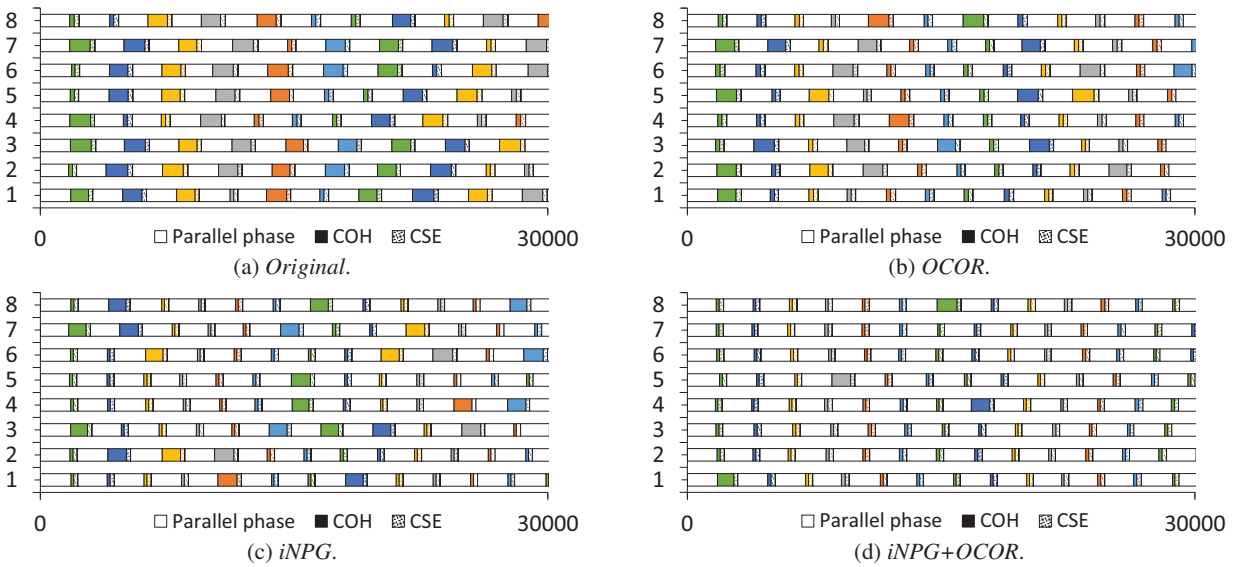
To look into the details and impact of CS entry and access, we profile the program execution timing in Figure 9 with program *freqmine* for the four comparative cases. For clarity, we show the execution results of 30,000 CPU cycles of the first 8 threads in *freqmine*, where we divide the program execution timing diagram into three phases. (1) Parallel phase, where threads perform concurrent computation tasks; (2) COH phase, where threads compete with each other to enter the next critical section; (3) CSE phase, where threads execute code in critical sections.

The *Original* application execution timing profile is shown in Figure 9a. As illustrated, 62.1% of the CPU cycles are spent in the parallel phase, with 28.3% in COH and 9.6% in CSE. Moreover, 78 critical sections are completed during the reported 30,000 CPU cycles. However, with *OCOR*, COH across different threads is significantly reduced. As shown in Figure 9b, 69.8% of the CPU cycles are spent in the parallel phase, with 19.8% in COH and 10.4% in CSE. Moreover, 92 critical sections are completed during the reported 30,000 CPU cycles, achieving 17.9% application progress improvement than *Original*. This is because more threads secure the critical section in the low-overhead spinning phase instead of the high overhead sleep phase.

Further, with *iNPG*, COH is also remarkably reduced. As shown in Figure 9c, 73.0% of the CPU cycles are spent in the parallel phase, with 17.0% in COH and 10.0% in CSE. Moreover, 96 critical sections are completed during the reported 30,000 CPU cycles, achieving 23.1% application progress improvement than *Original*. This is because invalidations to failing or later arriving `SWAP` requests are sent from big routers instead of from the home node. In this way, the losing threads get early invalidations so that the application avoids heavy coherence traffic latency. Finally, with *iNPG+OCOR* in which both CS grant order and the coherence traffic latency are optimized, COH is further reduced than with *iNPG* or *OCOR* alone. As drawn in Figure 9d, 80.1% of the CPU cycles are spent in the parallel phase, with 9.0% in COH and 10.9% in CSE. Moreover, 104 critical sections are completed during the reported 30,000 CPU cycles, achieving 33.3% progress improvement than *Original*.

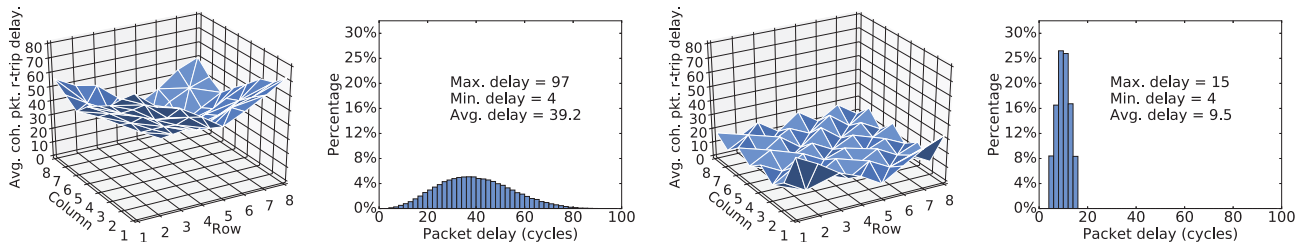
5.2.3 Analysis on LCO reduction

To reveal the effects of *iNPG* on reducing lock coherence overhead (LCO), Figure 10 compares the average and histogram of coherence packet round-trip delay for *Original* and



(a) *Original*. (b) *OCOR*. (c) *iNPG*. (d) *iNPG+OCOR*.

Figure 9: Comparison of the execution timing profiles of program *freqmine* with the four comparison mechanisms. Each thread execution is divided into three phases: parallel phase, COH phase, and CSE phase.



(a) Avg. coh. pkt. r-trip delay (b) Coh. pkt. r-trip delay hist. (c) Avg. coh. pkt. r-trip delay (d) Coh. pkt. r-trip delay hist.

Figure 10: Average coherence packet round-trip delay and coherence packet round-trip delay histogram comparisons for *Original* (Figure (a), (b)) and *iNPG* (Figure (c), (d)) with benchmark *freqmine*.

iNPG in benchmark *freqmine*. The results are obtained for a scenario where all 64 threads compete for the lock variable that is hosted at the shared L2 cache of core (5,6). The measurement starts at the time when all threads begin to compete for the critical section and ends at the time when the last thread gets its critical section.

Figure 10a shows the average coherence Invalidation-Acknowledgement (Inv-Ack) round-trip delay between the home node and all competing threads in *Original*. Without *iNPG*, the home node holding the critical section lock is solely responsible for all the coherence invalidations, with the winning thread responsible for collecting the acknowledgements from failing threads. Thus, depending on the distance between a core and the home node, cores near the home node enjoy low coherence overhead (since it can be invalidated at an early time), while cores that are farther away from the home node suffers from higher coherence overhead. Figure 10b further shows the corresponding coherence Inv-Ack round-trip delay histogram. The maximum coherence packet round-trip delay reaches 97 CPU cycles, exhibiting a "long tail" on the delay histogram, dominating the coherence completion time among the home node and competing threads.

Figure 10c shows the effects of *iNPG* in reducing the average coherence Inv-Ack round-trip delay. Applying *iNPG*, the coherence Inv-Ack round-trip delay has less dependence on the distance between the home node and the competing threads. This is because in *iNPG*, invalidation and the corresponding acknowledgement can be done with distributed big routers instead of aggregately at the home node. In this way, when a `GetX` request loses arbitration, the L1 cache in its issuing thread will be invalidated by the nearest big router instead of the home node. Figure 10d further shows the corresponding coherence Inv-Ack round-trip delay histogram. Compared to Figure 10b, the maximum coherence packet round-trip delay is reduced to 15 CPU cycles, where the "long tail" delay is eliminated. The average coherence packet round-trip delay is decreased from 39.2 to 9.5 cycles.

5.2.4 *CS and application finish time reduction*

Figure 11 shows the overall critical section (including both COH and CSE) expedition results of the four mechanisms. We normalize the results obtained from *Original* to 1 in each benchmark. We can observe that the critical section expedition results are proportional to the total CPU cycles that an application spends in critical sections as illustrated in Figure 8b. This is because the more CPU cycles that a program spends in critical sections, the more opportunity is opened for *iNPG* and *OCOR* to achieve higher competition overhead reduction. As shown in Figure 11, across Group 1 applications, critical sections are averagely expedited by 1.2× in

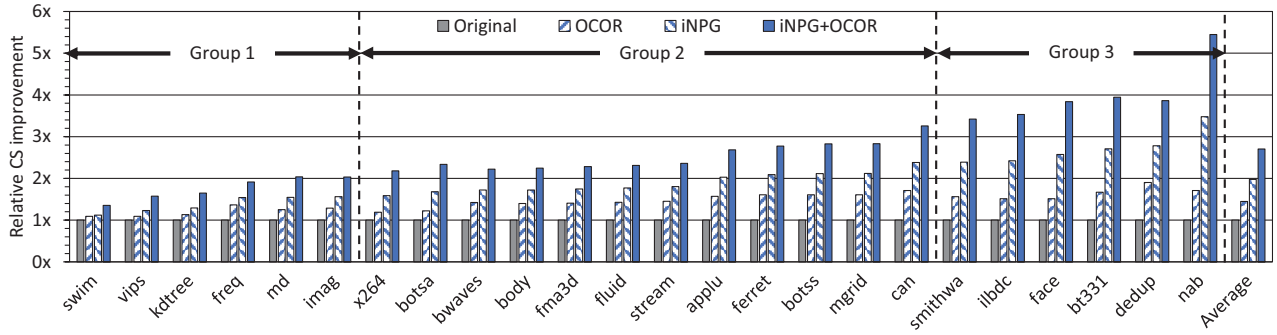


Figure 11: Critical section expedition results achieved by the four comparison mechanisms.

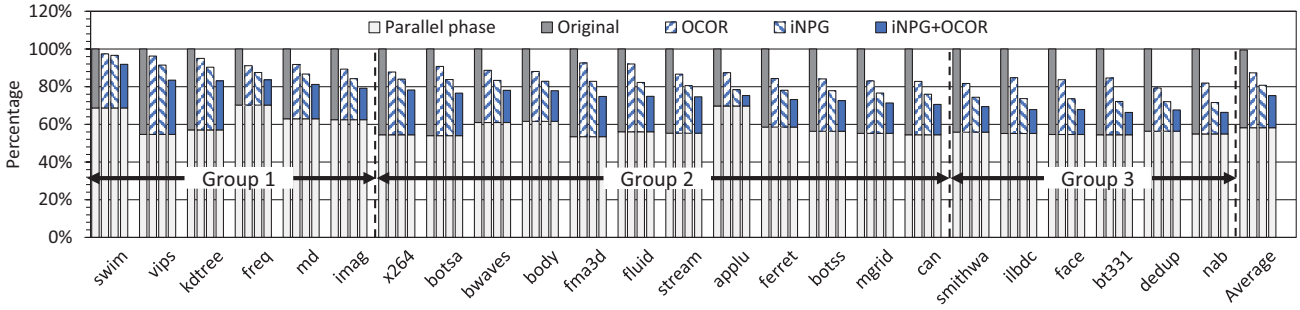


Figure 12: Application ROI finish time achieved by the four comparison mechanisms.

OCOR, 1.4× in *iNPG*, and 1.8× in *iNPG+OCOR*. Across Group 2 applications, average critical section expedition is further increased to 1.5× in *OCOR*, 1.9× in *iNPG*, and 2.5× in *iNPG+OCOR*. Across Group 3 applications, critical sections are further expedited by 1.6× in *OCOR*, 2.7× in *iNPG*, and 4.0× in *iNPG+OCOR*, respectively. Across all 24 programs, *OCOR* expedites critical sections averagely by 1.45× and maximumly by 1.90× (with program *dedup*). Further, *iNPG* averagely expedites critical section by 1.98× and maximumly by 3.48× (with program *nab*). In *iNPG+OCOR*, the average and maximum critical section completion is expedited to 2.71× and 5.45× (with program *nab*), respectively. Compare *iNPG* over *OCOR*, *iNPG* effectively expedites critical section access by 1.35× on average and 2.03× at maximum (with program *nab*).

Figure 12 shows the relative application ROI finish time of the four comparison mechanisms, in which we normalize the results obtained from *Original* to 100%. We can observe that all the four techniques have negligible effects on application parallel execution phase, where each thread executes parallel computation tasks and encounters no critical section. The ROI finish time reduction achieved by *OCOR*, *iNPG* and *iNPG+OCOR* correlates to the CS characteristics of each application. In Group 1 applications, compared with *Original*, the average ROI finish time is reduced by 6.5% with *OCOR*, 10.6% with *iNPG*, and 16.3% with *iNPG+OCOR*. Across Group 2 applications, the average ROI finish time is reduced by 12.6% with *OCOR*, 19.5% with *iNPG*, and 25.2% with *iNPG+OCOR*. In Group 3, the average ROI finish time is reduced by 17.3% in *OCOR*, 27.1% in *iNPG*, and 32.4% in *iNPG+OCOR*. Across all 24 programs, *OCOR* reduces the average ROI finish time to 87.7% by 12.3% compared to *Original*. *iNPG* decreases the average ROI finish time to 80.1% by 19.9%. Finally, *iNPG+OCOR* reduces the average ROI finish time to 75.3% by 24.7%, exhibiting the highest ROI finish time reduction. Compare *iNPG* over *OCOR*, *iNPG* improves the ROI finish time by 7.8% on average and 14.7% at maximum (with program *bt331*).

We can observe from Figure 11 and Figure 12 that the overall benefit of *iNPG+OCOR* is not an accumulation of benefits from stand-alone *iNPG* and *OCOR*. This is because in *iNPG*, LCO in the spinning phase is significantly reduced, thus more threads can gain CS access without entering into the sleep phase. Since *OCOR* also aims to have more threads enter CS in their spinning phase, when combined with *iNPG*, the opportunity for *iNPG+OCOR* is reduced.

5.2.5 *iNPG's effectiveness with other locking primitives*

We now explain and show the effectiveness of *iNPG* in reducing application ROI finish time with other locking primitives, including test-and-set (TAS) lock, the ticket lock (TTL) [31], array-based queuing lock (ABQL) [2, 16] and the MCS lock [14].

In TAS, each competing thread keeps spinning on the local CS lock variable until it successfully enters the critical section. Thus, TAS generates extensive read-modify-write operations because every time the lock is freed, each thread generates an exclusive access request among which only one will succeed, with all failed ones invalidated by the coherence protocol. The number of read-modify-write operation is significantly reduced in TTL and ABQL. In both primitives, competing threads are served in the FIFO order. When a critical section is released, only one thread can issue a read-modify-write operation to enter the next critical section. The thread then takes the CS lock, and invalidates all CS locks in other threads' caches. In the MCS lock, the lock contention traffic is further reduced. Instead of polling on one unique

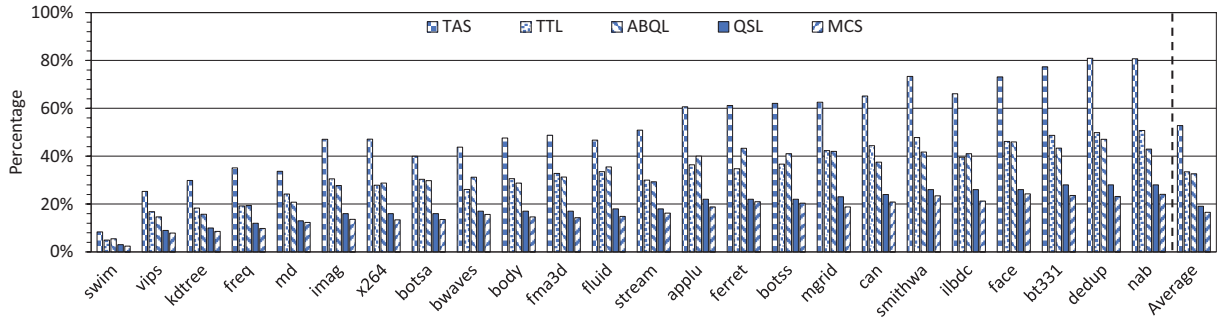


Figure 13: Application ROI finish time improvements with different locking primitives.

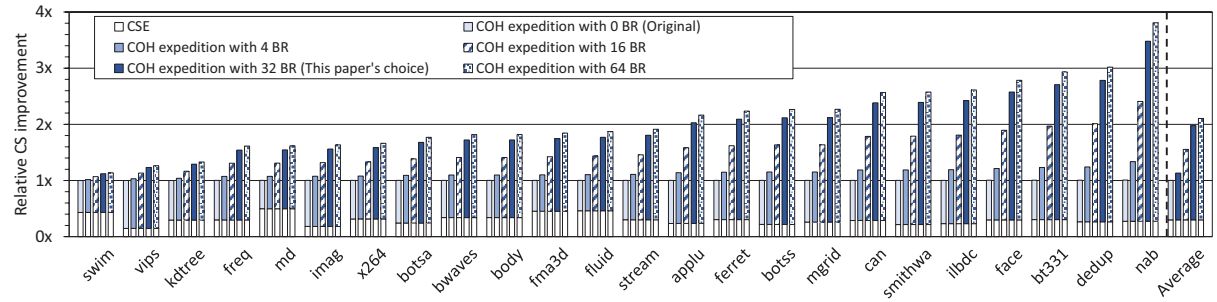


Figure 14: Critical section expedition results with different big router deployments.

lock variable, in the MCS lock, different threads poll on its previous thread to access the next critical section. That is, each thread (except for the first one) does not poll on a CS lock, but checks to see if its previous thread has finished its critical section execution. When a thread finishes the critical section, it directly notifies its successor to enter the next critical section.

Our experimental results are in accordance with the above analysis. Figure 13 compares the ROI finish time reduction achieved by *iNPG* with TAS, TTL, ABQL, QSL and the MCS lock. From the figure we can observe that 1) with different locking primitives, *iNPG* consistently reduces application ROI finish time across all benchmark programs. 2) The benefits of *iNPG* differ with different locking primitives. In average, after applying *iNPG*, ROI finish time is reduced by 52.8% in TAS, by 33.4% in TTL, by 32.6% in ABQL, by 19.9% in QSL and by 16.5% in the MCS lock. This shows that with TAS, TTL and ABQL, *iNPG* achieves more effective results than with QSL and MCS, which impose less lock competition traffic in the NoC.

5.2.6 *Sensitivity to big router deployment*

Since *iNPG* relies on big routers to function, we investigate its sensitivity to big router deployment by varying the number of big routers from 0 to 4, 16, 32 and 64. We distribute all big routers evenly on the chip, where 0 big router is the *Original* setup and 32 big routers represent the case illustrated in Figure 3. Figure 14 reports CS expedition results (including both CSE and COH) normalized to *Original*. We can make the following two observations. First, as expected, across different benchmarks, *iNPG* significantly expedites COH, with the CSE results remaining the same as in *Original*. Second, as the number of big routers increases, COH expedition is increased accordingly. However, the further COH expedition gains from 32 to 64 big routers are marginal. Therefore for the 64-core CMP, 32 big routers achieve a "sweet spot" between big router number and COH expedition, and are thus used as the default big router deployment for experiments in the paper.

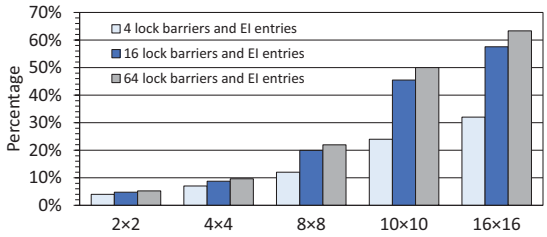


Figure 15: *iNPG*'s effectiveness with different NoC dimensions and locking barrier table sizes.

5.2.7 *Sensitivity to NoC dimension and number of entries in locking barrier table*

Figure 15 shows *iNPG*'s effectiveness on reducing average application ROI finish time across all benchmark programs with different NoC dimensions ranging from 2×2, 4×4, to 8×8 (default setup in the paper), 16×16 cores. From the figure we can observe that with the NoC dimension increasing, *iNPG* brings more effectiveness on reducing application ROI finish time. This is because as the number of cores scales, more threads are involved in competing for the same critical section, and thus more critical LCO becomes. For example, in the 2×2 NoC, *iNPG* averagely reduces application ROI finish time by 4.7%, which is increased to 19.9% in the 8×8 NoC and 57.5% in the 16×16 NoC.

Figure 15 also shows *iNPG*'s effectiveness with 4, 16 (default number in the paper) and 64 lock barriers and EI entries in the locking barrier table within a big router. From the figure we can observe that in 2×2 and 4×4 NoC, different size of locking barrier table brings marginal performance difference. However, in the 8×8, 10×10, and 16×16 NoC, 4 lock barriers and EI entries in the locking barrier table severely

restricts *iNPG*'s performance. This is because as the NoC dimension scales up, reducing the locking barrier table size directly limits the capability of big routers in sending early invalidations, thus reduces the benefit of *iNPG*. However, increasing the number of lock barriers and EI entries more than 16 does not proportionally increase *iNPG*'s performance. This is because each router in the NoC can only transfer a limited number of lock competing requests. Based on these observations, we choose 16 as the default number of lock barriers and EI entries in a locking barrier table within a big router.

6. RELATED WORK

Analysis and identification of critical sections and threads. Eyerman and Eeckhout analyzed Amdahl's law with the notion of critical section in [12]. They developed an analytical model to show that mutual excluded critical section execution imposes a fundamental limit on parallel programs' performance. In [5], Chen and Stenström proposed a critical lock analysis method for diagnosing critical section bottlenecks in multi-threaded applications. This method can reliably identify the critical sections that are critical for performance, and quantify the impact of such critical sections on the overall performance. In [11], based on the analysis of why multi-threaded workloads typically show sublinear speedup on multi-core chips, the concept of speedup stack was develop to quantify the impact of various scaling delimiters such as LLC and memory subsystem interference, lock spinning & yielding, workload imbalance, cache coherency, *etc.* Due to synchronization behavior, threads can be critical for program performance. In [8], the concept of thread criticality was developed and the thread criticality metric was defined to take into account both a thread's execution time and the number of concurrent waiting threads. Further, criticality stacks were designed to visually break down the total execution time into criticality-related components, facilitating detailed analysis of parallel imbalance.

Combining synchronization networks. Combining synchronization networks can be catogorized into 1) networks that can combine concurrent accesses to the same memory location [15, 13, 32, 30] and 2) networks in which atomic `fetch_and_add` operations from different cores are combined [14, 25].

In [10], Eisley *et al*. propose to implement cache coherence protocols within the network to allow in-transit optimizations of read and write operations. To reduce hot-spot contention for synchronization variables, Yew *et al*. [41] have proposed a data structure known as a software combining tree. Like hardware combining in a multi-stage interconnection network [15], a software combining tree combines multiple accesses to the same shared variable into a single access. Recently, Hendler *et al*. [17] present flat-combining, in which a combiner thread holding a global lock combines requests of all other competing threads. In [18], they apply the flat-combining mechanism to develop a synchronous queue algorithm. The algorithm first uses a single "combiner" thread to acquire a global lock and then serves other threads' CS requests.

To accelerate multi-threaded applications, both iNPG and combining synchronization networks exploit the opportunity that synchronization packets often pass through the same router. However, iNPG is fundamentally different from the concept of combining network such as [15, 13, 32, 30, 6]. In a combining network, the combining switches are able to recognize that two messages are directed to the same memory location and in such cases they can combine the two messages into a single one. However, iNPG seeks to invalidate shared cache copies of lock variables in an early stage of cache coherence protocol by generating new packets rather than combining the underlying synchronization operations. More importantly, combining network is invoked when two or more competing requests collide in-flight in the router: they both try to arbitrate in the same cycle with one win and the others fail. As investigated in [13], even in a large scale system such as a 512-node NYU's ultra-computer system, the occurrence of such scenario is still very low. However, in our iNPG, we do not require that two locking requests arbitrate at the same cycle at the same router. Instead, when the first locking request travels through a big router, a temporary "barrier" is then set up to stop subsequent locking requests.

In-network techniques for collective communication. In the context of providing efficient 1-to-M and M-to-1 communications in NoCs, in-network techniques for packet forking and aggregating were developed. In [23], Krishna *et al*. proposed Flow Across Network Over Uncongested Trees (FANOUT) and Flow AggregatioN In-Network (FANIN) to realize efficient 1-to-M forking and M-to-1 aggregation, respectively. On-chip routers were customized to support FANOUT and FANIN. At most routers along flow path, packets incur only single-cycle delays, thus approaching an ideal network with only wire delay/energy. By using clockless repetitive wires on the datapath, FANOUT and FANIN were leveraged to SMART-FANOUT and SMART-FANIN [22] to enable forking and reduction, respectively, across multiple routers in a network dimension in a single cycle, thus providing a scalable collective communication scheme for NoCs. The above in-network techniques aim to optimize collective communication via optimized routers, which perform efficient packet forking (like copy & paste) and merging but never create new packets. In contrast, our big routers generate new packets in the network, aiming to reduce COH.

7. CONCLUSION

We have presented an iNPG technique to reduce lock coherence overhead (LCO) in serialized critical section access so as to expedite multi-threaded applications running on NoC-based cache-coherent many-cores. Based on the observation that LCO lies straightly on the critical path dominating the overhead for lock spinning, iNPG intends to shorten LCO by turning long-range centralized coherence traffic into short-range distributed coherence traffic. iNPG is enabled by "big" router which is a conventional router enhanced with active cache-coherence packet generation capability. Extensive full-system simulation results in Gem5 running PARSEC and SPEC OMP2012 with five different locking primitives (test-and-set lock TAS, the ticket lock TTL, array-based queuing lock ABQL, Mellor-Crummey and Scott MCS lock, and the default queue spin-lock QSL in Linux 4.2) show that iNPG achieves significant improvements in COH and program runtime reduction over the state-of-the-art COH reduc-

competition overhead

tion technique OCOR [40]. Due to the nature of orthogonality, iNPG can be combined with OCOR to achieve the best improvements across the five locking primitives.

8. REFERENCES

[1] N. Agarwal, T. Krishna, L. S. Peh, and N. K. Jha, "GARNET: A Detailed On-Chip Network Model Inside A Full-System Simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 33–42.

[2] T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 1, no. 1, pp. 6–16, 1990.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 72–81.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[5] G. Chen and P. Stenström, "Critical Lock Analysis: Diagnosing Critical Section Bottlenecks in Multithreaded Applications," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 71:1—-71:11.

[6] D. N. Jayasimha, "Distributed Synchronizers," in *International Conference on Parallel Processing (ICPP)*, 1988, pp. 23—-27.

[7] B. K. Daya, C.-H. O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh, "SCORPIO: A 36-Core Research Chip Demonstrating Snoopy Coherence on a Scalable Mesh NoC With In-Network Ordering," in *International Symposium on Computer Architecture (ISCA)*, 2014, pp. 25–36.

[8] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior," in *International Symposium on Computer Architecture (ISCA)*, 2013, pp. 511–522.

[9] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel Application Memory Scheduling," in *International Symposium on Microarchitecture (MICRO)*, 2011, pp. 362–373.

[10] N. Eisley, L. S. Peh, and L. Shang, "In-Network Cache Coherence," in *International Symposium on Microarchitecture (MICRO)*, 2006, pp. 321–332.

[11] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup Stacks: Identifying Scaling Bottlenecks in Multi-Threaded Applications," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012, pp. 145–155.

[12] S. Eyerman and L. Eeckhout, "Modeling Critical Sections in Amdahl's Law and Its Implications for Multicore Design," in *International Symposium on Computer Architecture (ISCA)*, 2010, pp. 362–370.

[13] G. F. Pfister, W. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. S. Melton, A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," in *International Conference on Parallel Processing (ICPP)*, 1985, pp. 764–771.

[14] J. R. J. Goodman, M. M. K. Vernon, and P. P. J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1989, pp. 64–75.

[15] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," in *International Symposium on Computer Architecture (ISCA)*, 1998, pp. 239–254.

[16] G. Graunke and S. Thakkar, "Synchronization Algorithms for Shared-Memory Multiprocessors," *Computer*, vol. 23, no. 6, pp. 60–69, 1990.

[17] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat Combining and the Synchronization-Parallelism Tradeoff," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010, pp. 355–364.

[18] ——, "Scalable Flat-Combining based Synchronous Queues," in *Lecture Notes in Computer Science*, 2010, pp. 79–93.

[19] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, Fifth Edition*. Morgan Kaufmann Publishers Inc., 2011.

[20] M. D. Hill and M. R. Marty, "Amdahl 's Law in the Multicore Era," *Computer*, vol. 41, no. July, pp. 33–38, 2008.

[21] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs," in *International Symposium on Computer Architecture (ISCA)*, 2013, pp. 154–165.

[22] T. Krishna and L.-S. Peh, "Single-Cycle Collective Communication Over A Shared Network Fabric," in *International Symposium on Networks on Chip (NoCS)*, 2014, pp. 1–8.

[23] T. Krishna, L.-S. Peh, B. M. Beckmann, and S. K. Reinhardt, "Towards the Ideal On-chip Fabric for 1-to-many and Many-to-1 Communication," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 71–82.

[24] N. B. Lakshminarayana, J. Lee, and H. Kim, "Age-Based Scheduling for Asymmetric Multiprocessors," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2009, pp. 25:1–25:12.

[25] J. Lee and U. Ramachandran, "Synchronization with Multiprocessor Caches," in *International Symposium on Computer Architecture (ISCA)*, 1990, pp. 27–37.

[26] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Transactions on Computer Systems (ToCS)*, vol. 9, no. 1, pp. 21–65, 1991.

[27] T. Y. Morad, A. Kolodny, and U. C. Weiser, "Scheduling Multiple Multithreaded Applications on Asymmetric and Symmetric Chip Multiprocessors," in *International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, 2010, pp. 65–72.

[28] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, P. Shelepugin, M. van Waveren, B. Whitney, and K. Kumaran, "SPEC OMP2012 - An Application Benchmark Suite for Parallel Systems Using OpenMP," in *International Conference on OpenMP in a Heterogeneous World (IWOMP)*, 2012, pp. 223–236.

[29] L.-S. Peh and W. J. Dally, "A Delay Model and Speculative Architecture for Pipelined Routers," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2001, pp. 255–266.

[30] G. F. Pfister and V. A. Norton, "Hot-Spot Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers (TC)*, vol. C-34, no. 10, pp. 943–948, 1985.

[31] D. Reed and R. Kanodia, "Synchronization with Eventcounts and Sequencers," *Communications of the ACM*, vol. 22, no. 2, pp. 115–123, 1979.

[32] R. D. Rettberg, W. R. Crowther, P. P. Carvey, and R. S. Tomlinson, "The Monarch Parallel Processor Hardware Design," *Computer*, vol. 23, no. 4, pp. 18–28, 1990.

[33] M. L. Scott, *Shared-Memory Synchronization*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[34] A. Sodani, R. Gramunt, and J. Corbal, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[35] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture. Morgan Kaufmann Publishers Inc., 2011.

[36] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 60–70.

[37] ——, "Accelerating Critical Section Execution with Multicore Architectures," *IEEE Micro*, vol. 30, no. 1, pp. 60–70, 2010.

[38] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-Directed Pipeline Parallelism," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 147–156.

[39] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling Heterogeneous Multi-Cores Through Performance Impact Estimation (PIE)," in *International Symposium on Computer Architecture (ISCA)*, 2012, pp. 213–224.

[40] Y. Yao and Z. Lu, "Opportunistic Competition Overhead Reduction for Expediting Critical Section in NoC Based CMPs," in *International Symposium on Computer Architecture (ISCA)*, 2016, pp. 279–290.

[41] P. C. Yew, N. F. Tzeng, and D. H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers (TC)*, vol. C-36, no. 4, pp. 388–395, 1987.