Joint security and performance improvement in multilevel shared caches

Amin Sarihi1 | Ahmad Patooghy2 | Mahdi Amininasab3 | Mohammad Shokrolah Shirazi4 | Abdel-Hameed A. Badawy1

1Klipsch School of Electrical and Computer Engineering, New Mexico State University, Las Cruces, New Mexico, USA
2Department of Computer Science, University of Central Arkansas, Conway, Arkansas, USA
3Independent scholar
4R.B. Annis School of Engineering, University of Indianapolis, Indianapolis, Indiana, USA

Correspondence
Ahmad Patooghy, Department of Computer Science, University of Central Arkansas, 201 Donaghey Ave., Conway, AR 72035, United States. Email: apatooghy@uca.edu

Abstract
Multilevel cache architectures are widely used in modern heterogeneous systems for performance improvement. However, satisfying the performance and security requirements at the same time is a challenge for such systems. A simple and efficient timing attack on the shared portions of multilevel hierarchical caches and its corresponding countermeasure is proposed here. The proposed attack prolongs the execution time of the victim threads by inducing intentional race conditions in shared memory spaces. Then, a thread-mapping algorithm to detect such race conditions between a group of threads and resolve them as a countermeasure against the attack is proposed. The proposed countermeasure dynamically monitors races on cache blocks and distributes existing and new threads on processing cores to minimize cache contention. Upon detection of a high contention rate that might be either due to an attack or a natural race condition, two mechanisms, namely cache access-rate reduction and thread migration, will be used by the countermeasure algorithm to resolve the race situation. Evaluations on SPECCPU 2006 benchmark suite show that the proposed algorithm not only protects the system against the introduced attack but also boosts the overall system performance by an average of 46.35% and 55.92% for the worst and average cases, respectively.

1 | INTRODUCTION

According to Moore’s law, the number of transistors in an integrated circuit doubles almost every 2 years. However, this shrinkage has not led to the development of highly dense single-core processors since single-core processors have serious performance [1] and power challenges. Alternatively, recent advances in VLSI fabrication technologies have urged designers towards the design of multiprocessor system-on-chips with thousands of processing, memory and IO cores placed into a single chip [2]. Such chips are being increasingly used in desktop and server platforms as tens to hundreds of processing cores boost the overall performance while maintaining power consumption manageable [3].

Figure 1 depicts a typical heterogeneous system composed of multiple processing domains working in parallel. A domain is defined as a group of cores that shares some non-processing resources such as memory (cache), communicating channels to upper layer memories, floating-point processing units etc. within the same domain. The goal is to improve resource utilization as well as the chip’s overall performance [2, 4–6]. However, running processes/threads on the processing cores of a domain in parallel (also known as thread-level parallelism [7, 8]), may raise competitions for shared resources that in turn degrade the chip’s performance [9, 10]. Among all types of resource competitions that may happen in a domain, cache races have been shown [11] to have the most adverse effect on the overall chip’s performance. Cores will have to refer to upper levels of the memory hierarchy that incurs hundreds of cycles of delay. One of the major negative outcomes of cache races is when threads suffer extra cache misses due to ongoing races between co-running threads. In other words, when a thread causes the eviction of active data blocks belonging to other co-running threads [2].

Cache races can be dealt with or without having prior knowledge about the running threads and their interactions...
[12, 13]. Without such knowledge, we may need to have a hardware monitor to detect and resolve the race conditions happening between cores. In contrast, prior knowledge may be used to develop a detector to detect possible race conditions when running threads show rapid changes in memory access patterns. The detector could be implemented in hardware or defined as a particular thread that is a part of the operating system’s (OS’s) task scheduler. Investigations show that a task scheduler thread can considerably mitigate the shared resources problem [5] and improve the performance in some of the threads up to 50%. A software-based approach is also beneficial as it does not impose any hardware overhead. It applies to broader products (even fabricated chips), and it can be tuned based on the characteristics and features of the target application [4, 14]. Devising a software-based solution for race detection oblivious to security measures is not a sustainable solution. Overall, system designers need to consider the following three security goals while designing a system: confidentiality, integrity and availability [15]. In the case of a cache race, availability becomes one of the paramount security priorities. By targeting availability, we make sure that system resources are available during their operation. An adversary might run programs or threads aiming to slow down the race detector thread or raise intentional races to impair availability. Attacks make the situation harder for normal threads, for example, forcing threads to miss their execution deadlines and terminating them by intentionally increasing their cache miss rates.

This study addresses both the security and performance of modern processors equipped with shared caches based on our previous work [16] by introducing a security/performance-aware thread mapping. The essential feature of the proposed method is its simplicity that makes its overheads very low. The proposed algorithm maps threads onto cores of a domain such that race condition among them is minimized. We do this by monitoring the instruction per cycle (IPC) metric of the threads running in each domain. This study’s most important contributions are the following:

- Introducing a novel attack on shared caches that degrade the system’s performance up to 5 times.
- Proposing a security/performance-aware thread mapping.
- An analytical discussion to facilitate predicting the impacts of the proposed cache attack on the performance of victim threads.

Our experimental results and analyses confirm that the proposed mapping resists against the proposed attack.

The rest of this article is arranged as follows: In Section 2, related studies in the literature are reviewed and discussed. Section 3 explains the proposed attack model, and Section 4 elaborates the proposed thread mapping algorithm and the two mechanisms when it detects a race condition. Section 5 clarifies the details of the evaluation setup and results. The security analysis of the proposed system is presented in Section 6. Finally, Section 7 offers concluding remarks.

2 | MULTILAYER CACHE DESIGN CHALLENGES AND EXISTING SOLUTIONS

Multilevel hierarchical cache architectures have recently been used in the design and fabrication of heterogeneous multicore chips. Also, the local memory can be devoted to each core and higher levels of memory are shared among a group or all existing cores. The cores utilize different methods to access the shared cache and keep the data consistent on different portions of the multilevel memory. The main objective is to achieve a higher performance level for the whole system while keeping the data/threads secure. In this section, we review the design...
challenges of multilevel shared caches and the methods proposed in the literature to address them.

2.1 Cache performance problem

Cache contentions impose extra memory accesses to bring those active blocks of data which have recently got evicted from the cache. Much research has focused on this topic, and various methods have been devised for reducing race conditions such as [2, 5–7, 9, 10, 17–21]. However, many of these techniques remained at a theoretical stage, lacking practical aspects due to large operational overheads they impose on the system.

In [22], a hardware-assisted task partitioning and scheduling algorithm has been suggested based on the entropy maximization model. This work performs a range of complicated entropy evaluations to identify which thread(s) is (are) the source of race condition. Then, the algorithm maps them on various domains. Wen et al. [23] have suggested using a distributed hardware architecture over multicore systems to identify and remove races. For the suggested hardware architecture, a ring interconnection network is required to connect all processing cores to retrieve the required access patterns.

Authors in [24] have proposed the use of metadata in cache architecture to detect the race condition easier and faster by the hardware. The high overhead and processing time of the metadata are only embedded in selective regions of the shared memory. It helped to achieve a comparatively sound race detection coverage, but the hardware overhead is substantial, and the architecture needs more than 10K bytes of buffer per core.

In [20, 21], authors have proposed a cache partitioning technique to exploit different policies in different parts of the cache. Then each part is allocated to a separate thread, thereby alleviating the race issue. This method needs too many changes in the cache controller hardware. Furthermore, the method displays some inefficiency since it statically allocates cache to some threads [20, 21], thus failing to exploit the full capacity of the cache, leading to performance degradation.

Researchers primarily used scheduling algorithms to reduce cache race [2, 5, 6, 8, 10, 18, 19] in software-based techniques. Fedorova et al. [5] introduced a metric called ‘Pain’ calculated for all pairs of threads ready for execution. If their pain were minimum, thread pairs would be assigned to one domain. Pain (A, B) indicates performance degradation of threads A and B when executed on the same domain, compared to the scenario when executed individually without any contention. This technique has a heavy operational overhead and some problems with synchronization in actual implementation. In [25], authors have suggested the idea of running various race-detecting threads alongside extra application threads. Each race-detecting thread is accountable for a specific range of shared memory addresses for race detection. This method can be scaled when the number of cores increases on the system since the number of detecting threads can be scaled. However, we do not detect the race conditions happening by the detecting threads.

Zhuravlev et al. [2] introduced a method based on separating domains where threads have a high cache miss rate. In this technique, devil threads, that is, threads with high cache miss rate that tend to occupy all cache blocks are distributed among the processor domains [2]. An ordered list of threads is generated based on miss rates and two threads with the highest and lowest miss rates are selected to be mapped on the same domain. The next two threads are selected and mapped on another domain, and the process is continued [2]. In [18], a threshold-based method is suggested in which ready-to-run threads are divided into two groups: (i) devils: making contentions and (ii) turtles: not making contention. By considering whether or not their miss rate is larger than a predefined threshold, the devils are spread across domains, and the turtles are allocated to remaining cores. In [16], authors have used the IPC as a metric to detect conflicting threads. After detecting such threads, at least one of them will be removed to another domain to resolve the race condition. As this might not be an endpoint for contention, the algorithm proposed in [16] repeatedly checks the IPCs of every domain of the system.

Despite the notable previous work introduced in this section, the cache performance problem remains an important issue. Any devised approach that targets performance must consider the timing constraints of the tasks and should have a low impact on the system overall performance. Our software-based solution is proposed with an awareness of the shortcomings of the previous work. Moreover, it does not impose hardware overhead since we have used a software-based mechanism.

2.2 Cache security problems and countermeasures

Unix OS has suffered from time-of-check-to-time-of-use race conditions in the last 30 years [26]. Some countermeasures have been introduced to overcome problems such as static and dynamic detectors [27–30] and probabilistic user-space defenses [31, 32]. Authors in [33] bypass the firmware flash protection introduced by Intel chipsets; however, the influence of the attack is mitigated by the added chipset flash protection features.

In cache attacks, the attacker exploits the difference between the reading and writing time in the cache and uses this inherent cache feature to compromise the security of a system and steal information. This information is leaked inadvertently through hidden channels such as execution time and power consumption. Removing cache is not a viable solution since it contributes to a significant exacerbation of performance [11]. Yarom et al. [34] proposed the FLUSH + RELOAD attack which is based on cache hits/misses that the attacker code experiences. In this attack, a piece of data is shared with the victim, then the attacker flushes a particular line of interest in the cache and waits until the victim finishes. Then the attacker starts reloading his data into the cache, and checks the access time each time to detect if it is a hit or a miss (hit time is about 1/10 of miss time). Every hit means the data has already been
loaded in the cache by the victim thread, and accordingly the attacker will understand the accessed addresses of the victim thread.

PRIME + PROBE attack is the inverse of the FLUSH + RELOAD attack. A cache set is filled with the controlled data, and the victim performs operations and evicts some lines of data from the cache. After the victim has executed, the attacker finds out which sets and lines of the cache have been replaced by comparing the cache access time. Hence, the attacker gets a sound grasp of the victim's activities and can guess which blocks of memory were used by the victim [35].

The EVICT + TIME attack was proposed in [36]. The purpose of this attack is to gain knowledge about the advanced encryption standard (AES) key by evicting an entire cache set. The attacker then runs the encryption program, and if the partially evicted AES lookup table was accessed, a miss would occur, and encryption would take longer to be completed. By observing the time, the attacker could guess which indices of the table had been accessed.

One of the most straightforward solutions to mitigate cache-based channels is flushing which affects the performance cost. Flushing L1 cache comes with a relatively lower cost than L2 and L3 cache since it is smaller in size. Moreover, the cache coherence controller needs to synchronize the data in all cache levels [37]. Cache colouring was first proposed for enhancing the processor's overall performance; however, it was proven an efficient way to preserve the system from cache timing channels [38]. Cache colouring divides memory into coloured memory pools, and by allocating memory from different pools to isolated security domains, the security measures will be enhanced [37]. This method has drastic impacts on the system's performance since it limits the available cache per thread and is not salable.

In [39], authors propose a random fill cache architecture as a solution for reuse-based attacks. Cache refill strategy is altered in such a way that the history of memory accesses cannot be inferred. When a miss occurs in the cache, the data is sent directly to the processor, and no lines in it are allocated. Instead, the cache is filled with a window of the missing memory line. Percival [40] suggests that systems that exploit hyperthreading are susceptible to any attacks from an adversary. The effect of disabling hyperthreading is notable on the system's throughput, but the trade-off between security and performance is acceptable.

Availability of system resources has not been directly addressed in multilevel cache architectures as a security goal in the previous work. Most of the works have targeted confidentiality and data stealing. Hence, we emphasize availability as the security focus of our work. The proposed mapping algorithm will address both security and performance simultaneously.

3 | PROPOSED ATTACK MODEL

In this section, we introduce the proposed cache attack and explain how it prolongs the execution time of a specific task/thread in a multilayer hierarchical cache architecture. The attack is based on the idea of inducing intentional cache contentions to increase the execution time of tasks/threads running on a specific core of a multicore system. The critical feature of the attack is its independence to access a private code/data space while still prolonging the execution time by 5× (for results, see Section 6). Figure 2 shows the conceptual model of the proposed attack. We see the following components in the attack:

1. Normal tasks running in a given processing core: these tasks are owned by the user(s). An attacking task will try to prolong their execution time in a fashion that they miss their deadlines.
2. Shared memory: it is a level-one cache memory in the system that is shared between a group of tasks/threads running on the same core/domain. This shared memory is the place where all core tasks have common benefits, but at the same time may be the source of benefit loss by having cache contention.

3. Attacking task: it is owned by the attacker trying to access memory in a way that will end with a race condition. Here, the attacking task allocates memory in run-time and searches for access traces which introduce the required performance loss in the attacking task. Since the attacking task is sharing the memory space with ordinary tasks, the highest performance loss in the attacking task most probably means the same thing in typical tasks. Details of how the attacking task implements a real attack are explained in the rest of this section.

Attacking task follows the procedure shown in Algorithm 1 to implement a real cache contention attack. It performs random access in a run-time allocated memory to occupy shared cache the most and subsequently, this increases the miss rates of normal tasks. The attacking task needs a time-measure to understand that a successful attack is made. For this goal, in the first part of the proposed attacking task it defines an integer array bigger than 1K and issues sequential addresses to initialize all cells of this array to a constant value. The required time for this operation (denoted as \( \Delta_t \)) will be captured and logged by the attacking task as a normal time for this operation, that is, a reference time. Then, the attacking task issues a random address trace to the array and captures the time again. If the captured time is at least 100% larger than \( \Delta_t \), that is, whenever the random access needs at least \( 2\times \) time for accessing the array, the attacking task counts this as a successful attack. Note that this is, in fact, a lower bound for cases with an accomplished attack, that is, we assume that up to 100% delay in the deadline of the victim thread is acceptable, which is far more than the reality in many embedded and real-time applications. It is worth mentioning that when the memory access operation in the attacking task is delayed, there should be a contention that will delay the normal tasks. Our evaluations show that this expectation is valid and creating intentional connections may significantly delay in normal tasks.

In each iteration, the attacking task allocates a portion of the memory and defines an array on that. Since the OS will determine the starting address of the array, this gives the attacking task an alternative option for the next trials if the current trial does not end up with a good result. In each attempt, the attacking task tries 10 times and generates 10 random address traces, and if at least one of them results in a delay more significant than 100%, that trace will be saved for the next couple of cycles as long as the delay is still higher than 100%. Otherwise, the attacking task releases the memory and allocates another array with the same size. The renewal process takes advantage of the OS involvement and will have a different start address that may end up with a successful attack. To address the dynamic behaviour of a multilayer shared cache, the attacking task calls the attack algorithm shown in Algorithm 1 every couple of cycles.

We have implemented the attack on a system with a Linux OS running different applications of the Mibench suite [41]. Then, we logged the execution time of the applications using the OProfile performance monitoring tool [42]. Results shown in Figure 3 confirm that the original execution times of the tasks have been prolonged in the presence of the proposed attack. In this experiment, we have implemented the attack with arrays of 500 and 5000 entry lengths, and we have observed that larger arrays have a more severe impact in terms of imposed delay. This is because longer arrays give the attacker task chances of farther jumps and a more unpredictable address pattern. Another observation is that the effects of the performed attacks on the benchmark applications are not always the same. For example, in quicksort (qsort) and hash (sha) tasks, attacks with 500 and 5000 entry lengths could prolong the execution time by \( 5\times \); however, the prolongation figures for 500 and 5000 attack arrays in the cyclic redundancy check (crc32) benchmark are \( 1.5\times \) and \( 2.5\times \) respectively. This is due to the nature of crc32 that issues fewer spread addresses than those of sha and qsort.

### 4 SECURITY/PERFORMANCE AWARE THREAD MAPPING

In this section, a thread mapping/remapping algorithm is proposed to jointly to address the performance and security of multilayer shared caches (a pure performance version of this algorithm is published in [16]). The proposed algorithm conducts dynamic thread mapping/remapping for each domain's cores based on the average number of IPC. Average IPC is the arithmetic mean of all running threads within that domain. We assumed that if there is a cache race condition in a domain, it will result in unintended cache misses, which will degrade the domain's performance. Domains with race conditions show

---

**ALGORITHM 1** The Proposed Attack Algorithm.

```plaintext
Function AttackAlgorithm():
    Allocate a memory of 1KB → A[];
    Start_timer() → t0;
    for i ← 1 to size of A[] do
        A[i] = ConstantInteger;
    end
    End_timer() → t1;
    Δt = t1 - t0;
    for j ← 1 to 10 do
        Start_timer() → t0;
        Tj ← random trace of addresses to A[];
        For all addresses m∈Tj, A[m]++;
        End_timer() → t1;
        if ((t1 - t0) ≥ 2Δt) then
            Save the Tj;
            Exit the For loop; /* attack successful*/
        end
    end
    if (attack has not been successful) then
        Release Array A[];
        Call AttackAlgorithm();
    end
End Function
```
IPCs lower than other domains that are not involved in this condition. We define $\text{IPC}_1, \text{IPC}_2, \ldots, \text{IPC}_m$ as the average IPC for domains $D_1$ to $D_m$ in a multicore system, respectively. Concerning average IPC for every domain, we define the following working status (see Figure 4):

- **State $S_1$:** a normal state in which all domains work with no cache race. This can be detected when the average IPC of all domains is greater than an expected threshold IPC, that is, $\text{IPC}_1, \text{IPC}_2, \ldots, \text{IPC}_m \geq \text{Th}_{\text{IPC}}$. Additionally, the difference between the maximum and minimum average IPCs is lower than the predefined threshold, that is, $\text{Max}(\text{IPC}) - \text{Min}(\text{IPC}) < \text{Th}_\delta$.

- **State $S_2$:** in this underutilized state, at least one domain has its average IPC lower than the expected IPC, that is, $\exists i$ such that $\text{IPC}_i < \text{Th}_{\text{IPC}}$. However, the difference between the maximum and minimum average IPCs is lower than the predefined threshold, that is, $\text{Max}(\text{IPC}) - \text{Min}(\text{IPC}) < \text{Th}_\delta$.

- **State $S_3$:** in this underutilized state, there is at least one domain that its average IPC is lower than the expected IPC, i.e., $\exists i$ such that $\text{IPC}_i < \text{Th}_{\text{IPC}}$ and $\text{Max}(\text{IPC}) - \text{Min}(\text{IPC}) < \text{Th}_\delta$. This might be due to a natural competition between threads or an intentional competition by an attacking thread.

The proposed algorithm aims to keep the average IPC of domains as high as possible. Therefore, the algorithm attempts to maintain the least possible difference between the maximum and minimum average IPCs. It begins operating in state $S_1$, as shown in Figure 4. In this state, threads are going to be assigned to cores based on the Naive spread algorithm [4].

In state $S_1$, the algorithm repeatedly checks average IPCs of all running domains and finds domains $D_i$ and $D_j$ with the minimum and maximum average IPCs, respectively. The average IPC of domain $D_i$ is checked, and if it is higher than or equal to the expected IPC, $\text{Th}_{\text{IPC}}$, the algorithm keeps staying in state $S_1$. Otherwise, the algorithm moves to either state $S_2$ or $S_3$ based on the difference between $\text{Max}(\text{IPC})$ and $\text{Min}(\text{IPC})$ values.

Algorithm 2 shows the pseudocode of the algorithm and how it works in the states mentioned above. The inputs are the threads and $P$ (a specified sampling period). The output is the mapped threads. Based on this algorithm, if the difference between the maximum and minimum IPCs is lower than the threshold, that is, $\text{Max}(\text{IPC}) - \text{Min}(\text{IPC}) < \text{Th}_\delta$, the proposed algorithm moves from state $S_1$ to state $S_2$. Being in state $S_2$ most probably means that the low IPC problem in domain $D_i$ does not stem from inappropriate thread mapping, but the...
majority of threads inside the domain have high miss rates. Therefore, a random number of threads in the low-IPC domain are temporarily frozen for 100K cycles.

Other threads could execute without serious races during this period and could overcome critical situations. The algorithm attempts to create a time gap between racing threads and potentially solve the cache race issue, which is the cause of low IPC in a domain. Threads are activated again after the freezing period is completed and proceed with their run. The algorithm does not perform any other race removal process during the freeze cycles and remains in state S₂ and returns to state S₁ after 100K cycles.

On the other hand, a big difference between \( D_i \) and \( D_j \) domain IPCs, that is, \( \text{Max}(IPC) - \text{Min}(IPC) \geq \text{Th} \), shows unsuitable thread mapping. This may happen due to either a natural race condition or activities of an attacking thread. In this case, the proposed algorithm decides to change the initial thread mapping, which was done based on Naive spread. Selected thread with the lowest IPC in domain \( D_i \) will be replaced by the thread with the largest IPC in domain \( D_j \). The algorithm also keeps track of migrated threads to check if they may/may not get involved in consecutive migrations. Three consecutive migrations for a given thread put the thread in the alert list, and after the fifth one, the thread will be terminated. After thread migration is completed, the algorithm returns to state S₁ and continues to check for all domains’ average IPCs.

Although the proposed algorithm is simple, it is effectively useful and improves the performance of the benchmarks well enough, which is shown in the next sections. This should be regarded because there is no need for hardware assistance for the algorithm.

5 | EVALUATION OF THE PROPOSED ALGORITHM

The proposed algorithm makes a multilayer cache system robust against the introduced attack as well as maintains its high performance. To investigate this further, the proposed thread mapping algorithm has been compared with the distributed intensity (DI) [2], threshold-based [2], N-cluster [4] and N-spread [4] thread mapping algorithms under different task sets obtained from the SPEC CPU 2006 benchmark suite. The last two algorithms are two algorithms that are suggested as reference algorithms in [4], that is, they are not contention-aware mapping algorithms.

Figure 1 shows the typical processor architecture for which these techniques are displayed. Naive cluster algorithm attempts to use a minimum number of domains to execute all threads. For example, if threads A and B are to be executed on the system shown in Figure 1, this algorithm maps threads A and B to cores 1 and 2 of domain 1, respectively, to have the highest number of idle domains. In contrast, the Naive spread algorithm attempts to involve as many domains as possible, that is it spreads tasks across different domains. In the previous example, this algorithm allocates threads A and B, respectively, to core 1 of domain 1 and core 1 of domain 2.

Simulations are conducted using the Akula simulator [4], a toolkit for experimenting and creating multicore thread placement algorithms that support multicore shared-resource contention-aware scheduling methods. Akula simulator inputs are features of the processor and the set of threads. The output is a file indicating the period of handling each thread in seconds in the processor and its performance gain/loss concerning that the thread is the only running thread on the system. Simulations are repeated for 4, 8, 16, 32 cores per chip in 2, 2, 4, 8 processing domains, respectively. In all simulation experiments, cache L2 is 8 Mb.

5.1 | Used thread sets in simulations

In simulations, different scenarios are taken into consideration in order to study the weaknesses and strengths of our algorithm extensively. The threads from the SPEC CPU 2006 benchmark suite are chosen for all scenarios. Threads can be subdivided into the following two categories in the SPEC CPU 2006 benchmark suite:

1. High-memory demand (HI-MD): threads with high memory usage, which may result in (1) high rates of a cache miss and (2) cache race condition when placed with another HI-MD thread on the same processing domain [2].
2. Low-memory demand (LO-MD): threads with low memory usage that have relatively lower cache misses. This scenario resembles the normal mode operation without a race condition.

Various combinations of HI-MD and LO-MD threads are used to form thread sets to cover both natural and attack situations. To investigate the impact of memory-demand intensity on the gained/lost performance, thread sets are defined to show differing levels of memory demands. This task is done by having less or more HI-MD versus LO-MD threads in the thread set. Four types of thread sets are created as follows:

1. Level 1: as the thread with the highest memory demands, this set consists of six HI-MD and two LO-MD threads.
2. Level 2: this set consists of five HI-MD and three LO-MD threads.
3. Level 3: this set consists of four HI-MD and four LO-MD threads.
4. Level 4: this set consists of three HI-MD and five LO-MD threads to serve as the set with the lowest memory demand.

For every simulation experiment, 10 different threads satisfying the levels mentioned above the memory demands are selected from the SPEC CPU 2006 benchmark suite. Simulations are done 10 times, and after getting results from the Akula simulator, the least and average performances are logged. Results of every batch of 10 runs are used to calculate the arithmetic mean.
5.2 Experimental results and analysis

The performance of DI, threshold-based, Naive-cluster, Naive-spread and proposed algorithms for the mentioned sets of threads is assessed in terms of the two following performance metrics [6]: the average performance degradation (APD) as well as the worst-case performance degradation. The performance degradation factor is the performance aggravation of a given thread running on a multicore shared-cache platform compared to the time when it is running on a single-core chip without sharing the cache. It is worth mentioning that degradation occurs due to cache race conditions. The performance degradation factor for the $i$th thread is calculated using Equation (1).

$$\text{Perf}_{\text{Deg}}^i = \frac{RT_{\text{Multi-core}}^i - RT_{\text{Single-core}}^i}{RT_{\text{Single-core}}^i}$$  \hspace{1cm} (1)$$

where $RT_{\text{Single-core}}^i$ is the response time of $i$th thread in the single-core processor, and $RT_{\text{Multi-core}}^i$ is the response time of the $i$th thread in the multicore processor. APD is calculated using Equation (2).

$$\text{APD} = \frac{n}{\sum_{i=1}^{n} \text{Perf}_{\text{Deg}}^i}$$  \hspace{1cm} (2)$$

where $n$ is the number of cores in the chip on which the threads are running. The worst-case performance degradation is the maximum performance degradation of the threads that are running on $n$ cores, which is calculated using Equation (3).

$$\text{Worst_Peref}_{\text{Deg}} = \text{MAX}_{i}^{n} \text{Perf}_{\text{Deg}}^i$$  \hspace{1cm} (3)$$

Improving the average degradation of performance benefits the whole system. It can also save power consumption as the system can be shut down early. Improving the worst-case degradation results is useful for the quality of service (QoS) because the contention-aware scheduler provides stable execution times to guarantee better QoS for applications. To evaluate the scalability of the proposed algorithm, all steps, as mentioned earlier, are performed for configurations introduced in Table 1. Table 2(a) and (b) depicts percentages, respectively, and the proposed algorithm improves worst-case and average-case performance degradation over the other four mapping algorithms under various memory demand levels. As can be seen in Table 2(a) and (b), the proposed algorithm is highly capable of reducing contention between threads and improving performance, particularly for worst-case performance degradation. In Table 2(a), the proposed mapping outperformed N_cluster by 78.2% on average while the figure for DI is below 10%. The trend of average performance improvement (which is degradation reduction) is almost the same in Table 2b, and the proposed algorithm displays a 31% enhancement over DI. We also see that for some rare cases, the proposed mapping is not the best solution, for example, in level 3 memory-demand, the DI shows the least performance loss and is better than ours by 28% and 7% for worst and average cases.

Figure 5(a) and (b) shows the worst-case performance degradation and APD of separate threads in separate processors for all five algorithms, respectively. This shows a significant improvement in the proposed algorithm, which is comparatively better than the DI algorithm. The number of thread migrations undertaken to complete a thread set is also used in the evaluations (shown in Figure 6). The proposed algorithm needs far fewer migrations than DI algorithms. When the number of cores increases in the DI algorithm, migration exponentially increases while in the proposed algorithm, this is not the case.

### Table 1 Number of cores and domains used in the simulations

| Configuration | Total number of cores | Number of domains | Number of cores per domain |
|---------------|-----------------------|-------------------|---------------------------|
| Configuration 1 | 4                     | 2                 | 2                         |
| Configuration 2 | 8                     | 2                 | 4                         |
| Configuration 3 | 16                    | 4                 | 4                         |
| Configuration 4 | 32                    | 8                 | 4                         |

### Table 2 Performance improvement of the proposed thread mapping as compared to N_cluster, N_spread, threshold-base and DI algorithms in the worst (a) and average (b) cases under four different levels of memory demand in the system

| (a) Algorithm | Level 1 | Level 2 | Level 3 | Level 4 | Average |
|---------------|---------|---------|---------|---------|---------|
| N_cluster     | 70%     | 82%     | 89%     | 72%     | 78.2%   |
| N_spread      | 51%     | 64%     | 79%     | 65%     | 64.7%   |
| Threshold_base| 40%     | 30%     | 30%     | 40%     | 35%     |
| Distributed_intensity | 33% | 15% | -28% | 10% | 7.5% |

| (b) Algorithm | Level 1 | Level 2 | Level 3 | Level 4 | Average |
|---------------|---------|---------|---------|---------|---------|
| N_cluster     | 78%     | 70%     | 83%     | 79%     | 77.5%   |
| N_spread      | 66%     | 51%     | 68%     | 62%     | 61.7%   |
| Threshold_base| 61%     | 69%     | 39%     | 45%     | 53.5%   |
| Distributed_intensity | 59% | 37% | -7% | 35% | 31% |

Abbreviation: DI, distributed intensity.

6 SECURITY ANALYSIS

In this section, we analytically formulate the impacts of the proposed cache attack on the performance of a victim thread. We suppose that the victim thread occupies $m$ blocks of the
instruction cache, and the total size of the data cache is \( n \) blocks. While the victim thread performs on its targeted data accesses (data cache reads and writes), the attacker thread behaves randomly to maximize contentions. The attacker thread continuously runs in an infinite loop trying to bring random blocks into the shared data cache. Although the victim thread makes targeted accesses to the data cache, the attacker thread tries to point to the same block by issuing random accesses. Figure 7 shows the race condition between the attacker thread and the victim thread in a shared data cache. We formulate the average time needed for running \( m \) blocks of instructions of the victim thread using what is given in Equation (4).

\[
T_{\text{vic-thread}} = m \times \bar{T}_{\text{block}}
\]

where \( m \) is the number of instruction blocks of the victim thread at the instruction cache ready for execution by the local core and \( \bar{T}_{\text{block}} \) is the average time needed to execute a block of instructions. One major differentiating factor here is the type of instructions. They may be either memory or non-memory instruction. So, we can write the \( \bar{T}_{\text{block}} \) as given in Equation (5).

\[
\bar{T}_{\text{block}} = Rate_{\text{non-mem-inst}} \times \bar{T}_{\text{Inst}} + Rate_{\text{mem-inst}} \times [\bar{T}_{\text{Inst}} + \bar{T}_{\text{mem-access}}]
\]

where \( Rate_{\text{non-mem-inst}} \) and \( Rate_{\text{mem-inst}} \) are the rate of non-memory instructions and memory instruction in each block, respectively. \( \bar{T}_{\text{Inst}} \) is the average execution time of each instruction and \( \bar{T}_{\text{mem-access}} \) is the average time for accessing memory. Equation (6) gives the details for calculating the average memory access time.

\[
\bar{T}_{\text{mem-access}} = Rate_{\text{read}} \times [\text{Prob}_{\text{read-hit}} \times \bar{T}_{\text{read-hit}} \\
+ (1 - \text{Prob}_{\text{read-hit}}) \times \bar{T}_{\text{read-miss}}] \\
+ Rate_{\text{write}} \times [\text{Prob}_{\text{write-hit}} \times \bar{T}_{\text{write-hit}} \\
+ (1 - \text{Prob}_{\text{write-hit}}) \times \bar{T}_{\text{write-miss}}]
\]

Since the goal of the attack is to prolong the execution time of the victim thread, it tries to reduce either of \( \text{Prob}_{\text{write-hit}} \) or \( \text{Prob}_{\text{read-hit}} \) to delay the victim thread. From the attacker's point of view, reducing either \( \text{Prob}_{\text{read-hit}} \) or \( \text{Prob}_{\text{write-hit}} \) is the same. So let us calculate one of them in the presence of the attacker thread. For simplicity, at the first attempt we assume that the victim thread accesses only one block of data cache per each block of instructions, later we release this simplifying assumption. So, the probability of accessing the \( i \)-th block of data cache by the victim is one, for a specific \( i = j \) and zero otherwise as in Equation (7).

\[
P[Vic(B_i)] = \begin{cases} 
1, & \text{if } i = j \\
0, & \text{otherwise}
\end{cases}
\]

Since the attacker accesses all blocks of data cache randomly and uniformly, the probability of accessing the same block of data cache that the victim has already accessed would be as follows:

\[
P[Att(B_i)] = \frac{1}{n}
\]

As we mentioned, the attacker thread would like to reduce the read/write hit probabilities, that is, \( \text{Prob}_{\text{write-hit}} \) or \( \text{Prob}_{\text{read-hit}} \). The question is: what would these two probabilities be under the attack? To answer this, we calculate the probability of same-time access for all blocks of data cache.
using Equation (9), which would result in intentional misses in the victim thread.

$$P_{\text{Miss-inj}}^{\text{Vic}} = \sum_{i=1}^{n} P(\text{Att}(B_i) | \text{Vic}(B_i)) = \sum_{i=1}^{n} \frac{P(\text{Att}(B_i) \cap \text{Vic}(B_i))}{P(\text{Vic}(B_i))}$$

where $P(\text{Att}(B_i))$ and $P(\text{Vic}(B_i))$ are the probabilities of accessing the $i$-th block of data cache by the attacker and the victim threads, respectively (can be calculated by Equations (8) and (7), respectively). Based on the following two facts: (1) these two probabilities are independent and (2) the summation is non-zero in only one specific $i = j$, Equation (9) can be simplified as $P_{\text{Miss-inj}}^{\text{Vic}} = \frac{1}{k}$. Every time the attacker thread succeeds in an intentional miss, it is, in fact, reducing the read/write hit probability of the victim thread. So the new read/write hit probability for the victim thread relates to this probability and the number of memory instructions in every instruction block of the attacker thread is as stated in Equation (10).

$$\text{Prob}^{\text{new}}_{\text{hit}} = \text{Prob}_{\text{hit}} - \frac{1}{n} \times \overline{\text{No}}_{\text{mem-Inst}}^{\text{Att}}$$

where $\overline{\text{No}}_{\text{mem-Inst}}^{\text{Att}}$ is the average number of memory instructions per each block of instructions of the attacker thread. Since the attacker thread accesses the memory (and through that data cache) as much as possible, its $\overline{\text{No}}_{\text{mem-Inst}}^{\text{Att}}$ is higher than a normal program. So the victim’s hit probability decreases significantly as the attacker accesses the shared data cache. Now, let us remove the assumption we made in the derivation of Equation (7). Let the victim thread access $k$ different blocks of data cache during the execution of one of its instruction blocks where $k \ll n$. If we assume a uniform discrete distribution for accessing these $k$ blocks, the probability of accessing the $i$-th block of data cache by the victim can be given by Equation (11).

$$P(\text{Vic}(B_i)) = \begin{cases} \frac{1}{k}, & \text{if } i \in \text{set of } k \text{ blocks} \\ 0, & \text{otherwise} \end{cases}$$

Accordingly, Equations (12) and (13) generalize the probability of intentional misses in the victim thread.

$$P_{\text{Miss-inj}}^{\text{Vic}} = \sum_{i=1}^{n} P(\text{Att}(B_i) | \text{Vic}(B_i)) = \frac{\sum_{i=1}^{n} P(\text{Att}(B_i) \cap \text{Vic}(B_i))}{P(\text{Vic}(B_i))} = \frac{\sum_{i=1}^{n} \frac{1}{k}}{\sum_{i=1}^{n} \frac{1}{k}} = \frac{k}{n}$$

$P(\text{Att}(B_i))$ and $P(\text{Vic}(B_i))$ are two independent probabilities. Hence, they are multiplied and we have the new hit probability for the victim thread as Equation (13).

$$\text{Prob}^{\text{new}}_{\text{hit}} = \text{Prob}_{\text{hit}} - \frac{k}{n} \times \overline{\text{No}}_{\text{mem-Inst}}^{\text{Att}}$$

In this case, there will be more chance for the attacker thread to inject intentional misses to the victim thread and therefore a higher chance of realizing a successful attack.
To investigate the average number of memory instructions in the attacker thread and compare it with normal programs, we have set up an experiment using Intel Pin tool [43]. We ran a couple of normal programs and counted the instructions per each basic block (a series of instructions with jump instructions only as the last instruction). Figure 8 depicts the number of basic blocks and the average number of instructions per each basic block in five different programs. Although the number of basic blocks is different for each program, the average number of instructions per each basic block is just under five. We know that almost 15% of the instructions need access to memory. This percentage for the attacker program is calculated to be 30% by the same measuring tool.

7 | CONCLUSION

This study shows that race conditions in shared caches may raise serious performance problems for multicore processors. A novel attack is introduced to prolong the execution time of a specific thread, which is done independent of accessing a private data space. The attack can delay the execution of the normal program by up to 5×. Furthermore, a software-based race-aware thread mapping algorithm is proposed to mitigate cache contentions. The algorithm checks system status periodically and detects racing threads on the same domain based on IPC parameter. Racing threads are dealt with two mechanisms of temporary freezing and thread migration. Average and worst-case performance degradation confirm that our proposed thread-mapping algorithm saves notable performance by reducing cache contentions. It needs a much lower number of thread migrations as compared to other algorithms. Our wide-range simulation results revealed that relying only on task migration for race reduction is not an efficient solution. Finally, a thorough security analysis of the proposed attack is delivered by formulating the impacts of the proposed cache attack on the performance of a victim thread.

As the continuation of this research, we will try to implement the whole (or parts of) race detection algorithm on hardware. This expedites our race detection much faster, and hence more to the point decision will be made for thread migration or freeze.

ORCID
Amin Sarihi  https://orcid.org/0000-0002-0134-8418
Ahmad Patooghy  https://orcid.org/0000-0003-2647-2797

REFERENCES
1. Bui, J., Xu, C., Gururmurthi, S.: Understanding performance issues on both single core and multi-core architecture, Computer Organization. pp. 1–8. (2007)
2. Zhuravlev, S., Blagodurov, S., Fedorova, A.: Addressing shared resource contention in multicore processors via scheduling. ACM SIGPLAN Not. 45, 129–142. (2010)
3. Basmadjian, R., de Meer, H.: Evaluating and modeling power consumption of multi-core processors. In: 2012 Third international conference on future systems: where energy, computing and communication meet (e-Energy), IEEE Madrid, pp. 1–10. (2012)
4. Zhuravlev, S., Blagodurov, S., Fedorova, A.: Akula: a toolset for experimenting and developing thread placement algorithms on multicore systems. In: Proceedings of the 19th international conference on parallel architectures and compilation techniques, Vienna, pp. 249–260. ACM (2010)
5. Fedorova, A., Blagodurov, S., Zhuravlev, S.: Managing contention for shared resources on multicore processors. Commun. ACM. 53(2), 49–57 (2010)
6. Fedorova, A., et al.: Performance of multithreaded chip multiprocessors and implications for operating system design. In: Harvard computer science group technical report TR-09-05 (2005)
7. Xiang, X., et al.: Cache conscious task regrouping on multicore processors. In: Proceedings of the 2012 12th IEEE/ACM international symposium on cluster, cloud and grid computing (ccgrid 2012), IEEE Computer Society, Melbourne, pp. 603–611. (2012)
8. Lee, J., Shin, I.: Demand-based schedulability analysis for real-time multicore scheduling. J. Syst. Softw. 89, 99–108 (2014)
9. Samih, A., Krishna, A., Solihin, Y.: Understanding the limits of capacity sharing in CMP private caches. CMP-MSI Austin (2009)
10. Siddha, S., Pallipadi, V., Mallick, A.: Process scheduling challenges in the era of multi-core processors. Intel Technol. J. 11(4) 361–369 (2007)
11. He, Z., Lee, R.B.: How secure is your cache against side-channel attacks? In: Proceedings of the 50th annual IEEE/ACM international symposium on microarchitecture, Cambridge, pp. 341–353. ACM (2017)
12. Cai, J., Shrivastava, A.: Software coherence management on non-coherent cache multi-cores. In: VLSI design and 2016 15th international conference on embedded systems (VLSID), 2016 29th international conference on, IEEE, Kolkata, pp. 397–402. (2016)

13. Vantresse, D., Lipasti, M.H., Binkert, N.: Atomic coherence: leveraging nanophotonics to build race-free cache coherence protocols. In: 2011 IEEE 17th international symposium on high performance computer architecture (HPCA), IEEE, San Antonio, pp. 132–143. (2011)

14. Greathouse, J.L., et al.: Demand-driven software race detection using hardware performance counters. In: Proceedings of the 38th annual international symposium on computer architecture, San Jose, pp. 165–176 (2011)

15. Tehemlyk, A., et al.: Towards understanding uncertainty in cloud computing with risks of confidentiality, integrity, and availability. J. Comput. Sci. 36, 100581 (2019)

16. Sahneh, P.S., et al.: Racer: a thread mapping algorithm for race reduction in multi-level shared caches. In: 2019 27th Euromicro international conference on parallel, distributed and network-based processing (PDP), IEEE, Pavia, pp. 228–232. (2019)

17. Maheshwari, U., Devi, C., Anderson, J.: Improved conditions for bounded tardiness under EPDF fair multiprocessor scheduling. In: 18th International. IEEE proceedings of the parallel and distributed processing symposium, 2004, Santa Fe, p. 119. (2004)

18. Blagodurov, S., Zhuravlev, S., Fedorova, A.: Contention-aware scheduling on multicore systems. ACM Trans. Comput. Syst. 28(4), 8 (2010)

19. Anderson, G., Marwala, T., Newamondo, F.V.: Multicore scheduling based on learning from optimization models. Int. J. Innov. Comput. Inf. Control. 9(4), 1511–1522 (2013)

20. Qureshi, M.K., Patt, Y.N.: Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. In: MICRO-39, 39th Annual IEEE/ACM international symposium on microarchitecture, 2006, IEEE, Orlando, pp. 423–432. (2006)

21. Zhang, X., Dwarkadas, S., Shen, K.: Towards practical page coloring-based multicore cache management. In: Proceedings of the 4th ACM European conference on computer systems, Nuremberg, pp. 89–102. ACM (2009)

22. Li, S., et al.: Race-condition-aware and hardware-oriented task partitioning and scheduling using entropy maximization. IEEE Trans. Parallel Distrib. Syst. 29(7), 1589–1604 (2018)

23. Wen, C.N., et al.: Nuda: a non-uniform debugging architecture and nonintrusive race detection for many-core systems. IEEE Trans. Comput. 61(2), 199–212 (2012)

24. Huang, R., et al.: Low-overhead and high coverage run-time race detection through selective meta-data management. In: 2014 IEEE 20th international symposium on high performance computer architecture (HPCA), IEEE, Orlando, pp. 96–107. (2014)

25. Song, Y.W., Lee, Y.H.: A parallel fasttrack data race detector on multicore systems. In: 2017 IEEE internationalparallel and distributed processing symposium (IPDPS), IEEE, Orlando, pp. 387–396. (2017)

26. Cai, X., Gui, Y., Johnson, R.: Exploiting unix file-system races via algorithmic complexity attacks. In: 2009 30th IEEE symposium on security and privacy, IEEE, Oakland, pp. 27–41. (2009)

27. Michael, M., Bishop, M.: Checking for race conditions in file accesses. Comput. Syst. 2(2), 131–152 (1996)

28. Tyryllevich, E., Yee, B.: Dynamic detection and prevention of race conditions in file accesses. University of California, San Diego (2003)

29. Uppuluri, P., Joshi, U., Ray, A.: Preventing race condition attacks on file-systems. In: Proceedings of the 2005 ACM symposium on applied computing, Santa Fe, pp. 346–353 (2005)

30. Viega, J., Bloch, J.T., Kohno, Y., McGraw, G.: Is4k: a static vulnerability scanner for c and c++ code. In: Proceedings 16th annual computer security applications conference (ACSAC’00), IEEE, New Orleans, pp. 257–267. (2000)

31. Dean, D., Hu, A.J.: Fixing races for fun and profit: how to use access (2). In: USENIX security symposium, San Diego, pp. 195–206 (2004)

32. Tsafir, D., et al.: Portably solving file races with hardness amplification. ACM Trans. Storage. 4, 1–30 (2008)

33. Kalistenberg, C., Wojtczuk, R.: Speed racer: exploiting an Intel flash protection race condition. Bromium Labs (January 6 2015) (2015). https://bromiumlabs files wordpress.com/2015/01/speed_racer_whitepaper.pdf (data obrashcheniya: 0701 2016)

34. Yaron, Y., Farkkan, K.: Flush + reload: a high resolution, low noise, 13 cache side-channel attack. In: USENIX security symposium, San Diego, vol. 1, pp. 22–25 (2014)

35. Crane, S., et al.: Thwarting cache side-channel attacks through dynamic software diversity. NDSS. pp. 8–11. San Diego (2015)

36. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on AES, and countermeasures. J. Cryptol. 23(1), 37–71 (2010)

37. Ge, Q., et al.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. J. Cryptogr. Eng. 8(1), 1–27 (2018)

38. Cocks, D., et al.: The last mile: an empirical study of timing channels on sgx. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, Scottsdale, pp. 570–581. ACM (2014)

39. Liu, F., Lee, R.B.: Random fill cache architecture. In: Proceedings of the 47th annual IEEE/ACM international symposium on microarchitecture, IEEE Computer Society, Cambridge, pp. 203–215. (2014)

40. Percival, C.: Cache missing for fun and profit. BSCanCanada (2005)

41. Guthaus, M.R., et al.: Mibench: a free, commercially representative embedded benchmark suite. In: Proceedings of the workload characterization, 2001. WWC-4. 2001 IEEE international workshop. WWC 01, IEEE Computer Society, USA, pp. 3–14. (2001)

42. Chapter 38: oprofile. https://web.mit.edu/rhel/doc/5/RHEL-5-manual/Deployment_Guide-en-US/ch-oprofile.html

43. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Not. 40(6), 190–200 (2005)

How to cite this article: Sarihi A, Patroogyh A, Amininasab M, Shokrolah Shirazi M, A. Badawy AH. Joint security and performance improvement in multilevel shared caches. IET Inf. Secur. 2021;15:297–308. https://doi.org/10.1049/ise2.12023