Check for updates

Received: 1 August 2022 | Revised: 23 October 2022 | Accepted: 31 December 2022
DOI: 10.1002/cpe.7622

WILEY

Efficient thread-to-core mapping alternatives for application-level redundant multithreading

Sanem Arslan[1] | **Osman Ünsal**[2]

[1]Computer Engineering Department, Marmara University, Istanbul, Turkey

[2]Computer Science - Computer Architecture For Parallel Paradigms, Barcelona Supercomputing Center, Barcelona, Spain

Correspondence
Sanem Arslan, Computer Engineering Department, Marmara University, Istanbul, Turkeye.
Email: sanem.arslan@marmara.edu.tr

Funding information
The Scientific and Technological Research Council of Turkey, Grant/Award Number: BIDEB 2219

Summary

Redundant multithreading (RMT) is an effective thread-level replication method to improve the reliability requirements of applications. Although it significantly improves the robustness of applications, it comes with additional performance overhead since the redundant threads might share the same core resources. In our previous study [Efficient selective replication of critical code regions for SDC mitigation leveraging redundant multithreading. *J Supercomput* 2021;77:14130-14160], we presented an efficient software-level RMT approach, where we execute the most critical code regions with three threads to correct errors. In this study, we focus on further improving the performance of our software-level RMT method by presenting a set of different thread-to-core mapping alternatives. We provide different static mapping methods, which require preliminary information about the applications, such as execution time, instruction-per-cycle (IPC), or cache usage patterns, and a set of dynamic mapping methods, which map threads to cores dynamically at runtime without requiring any additional information. The dynamic mapping methods decide which threads are mapped to which cores at each scheduling point based on the IPC, cache miss rate, or cache access values of each thread as well as each core. Experimental results show that the dynamic mapping method, which maps threads to cores based on IPC values, outperforms all other static and dynamic methods. It also outperforms our baseline model, where the operating system handles the thread-to-core mappings by 8%, 7%, and 20% based on average speedup, harmonic speedup, and mean slowdown metrics.

KEYWORDS
performance evaluation, redundant multithreading, reliability, thread-to-core mapping

1 | INTRODUCTION

Redundant Multithreading (RMT) is an efficient method that applies thread-level redundancy to provide reliable execution. The performance overhead of RMT may not be too significant if there is enough number of cores to execute all the redundant threads in parallel without sharing the cores with other applications. However, the performance overhead becomes significant if there are multiple redundant threads for each application in a multi-application workload running on a hardware with a limited number of cores. If the number of cores is less than the number of application threads (including the redundant ones), then a set of threads needs to share a common simultaneous multithreading (SMT) core. The mapping of threads to available cores affects the system performance significantly in such a case. Threads sharing the same core may compete for the available shared resources, such as issue queue slots, instruction buffers, cache memories, and interconnection between these resources.[2] On the other

 © 2023 John Wiley & Sons, Ltd.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7622 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

hand, we should not map multiple high-IPC (instruction per cycle) threads on the same core in order not to slow down an application unnecessarily. Therefore, thread-to-core mapping decision is an important determinant of system performance in RMT implementations.

In our previous study,[1] we implemented an application-level RMT library in which the application programmer can annotate critical code regions (i.e., functions) to execute redundantly. We showed its' efficiency by presenting the reliability, performance, and energy consumption results of different applications. The first implementation did not explore different thread-to-core mappings. In this study, our focus is on further improving the performance of our RMT library by utilizing efficient thread-to-core mapping techniques. To do that, we construct a representative workload composed of multiple applications and run this workload on a limited number of cores to emulate a more realistic scenario. Therefore, multiple threads (either original or redundant threads) can share the same SMT core, and the performance of the system might vary depending on how we map the threads to the available cores. The focus of this paper is to observe which features of applications such as IPC, cache miss, cache access values have impact on thread mapping decisions. Furthermore, to decrease the number of thread migrations, should mapping decisions be made statically based on a preliminary investigation or should they be changed at each scheduling point even though it might suffer from high thread migrations costs. We aim to answer these questions to improve the performance of our RMT library with a more realistic scenario.

To demonstrate the efficiency of our mapping techniques, we construct the following experimental setup: First, we select six HPC kernels from the PolyBench suite,[3] where each application is annotated to execute its *main algorithm* region redundantly. Then, following a triplex-redundancy RMT implementation, each application features one primary and two redundant threads to correct errors occurring during the execution of these regions. Thus, our problem is to map 18 application threads onto the available 9 cores efficiently.

We implement several static mapping techniques, that require preliminary information about the applications, such as execution time, instruction-per-cycle (IPC), cache miss rate, or cache access rate. Furthermore, we implement a set of dynamic mapping techniques, which make the thread-to-core mapping decisions at runtime dynamically based on the IPC, cache miss rate, and cache access values of cores and threads. Since the execution behavior of the threads may change over time, the mapping decision can be updated based on the last scheduling interval. Experiment results show that the dynamic mapping technique, which maps the threads based on the IPC values, outperforms all other static and dynamic techniques as well as the baseline, where we do not interfere with the thread-to-core mapping decisions of the original RMT by 8%, 7%, and 20% based on average speedup, harmonic speedup, and mean slowdown metrics.

2 BACKGROUND

A soft error is a kind of transient error, that might alter the stored data in memory due to a temporary change in a semiconductor device.[4] Continuous reduction in transistor sizes makes modern architectures more vulnerable to such types of errors. Alpha particles, high energy cosmic rays, and thermal neutrons are some of the main reasons behind this type of error.[5] The effects of soft errors may not be dominantly visible for a single PC that is used everyday in an office environment; however, the soft error rate of an HPC system is reported as 0.15 per day with 900 compute nodes in one year of observation.[6] Therefore, reliability is an important design challenge for especially HPC systems with growing number of processing units in addition to performance and power consumption.

Redundancy is one of the most straight-forward methods to provide dependable computing by utilizing additional cores/threads. Inputs of the redundant units are copied first; then the execution is handled either serially or in parallel for the redundant parts and finally the outputs of those units are compared to detect/correct faults occurring during the execution. Dual Modular Redundancy (DMR) provides two execution copies to detect errors, Triple Modular Redundancy (TMR) not only detects errors occurring in one of the redundant units; but also corrects them by executing three execution copies. In redundant multithreading (RMT), replication is handled in thread-level and the redundant threads are executed either on the same SMT core or different cores.

We previously proposed an efficient application-level redundant multithreading method, in which the application programmer can annotate critical code regions (i.e., functions) to execute redundantly.[1] Our model is more suitable for applications with varying criticality needs among the functions. Figure 1 shows the execution steps of our approach in three steps.[1] In step 1, critical code regions (i.e., functions) of an application are determined offline by injecting faults while executing a specific function. We defined the most critical functions as the functions having a high impact on the application output as well as a high execution time percentage. After that, the most critical function is annotated by the application programmer by invoking a library function (*activateRMT*) to execute it redundantly. The inputs of the function with their types should be passed to the *activateRMT* function. Then, our library function transparently creates three threads for the redundant execution, copies inputs for each of them, compares the outputs of them after the function execution, and the correct output is determined based on majority voting before commencing the application execution. Therefore, the application can continue to the execution without being affected by a fault while executing that specified function.

An example annotation is shown in Listing 1 for the *syrk* application from the PolyBench suite.[3] For this application, the *main algorithm* kernel is found more critical than the other parts and it is annotated to execute in redundant mode. The *kernel_syrk* function is annotated in that example so that our RMT library transparently handles input replication for all arguments (2 integers, 2 doubles, and 2 two-dimensional double arrays in the

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7622 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

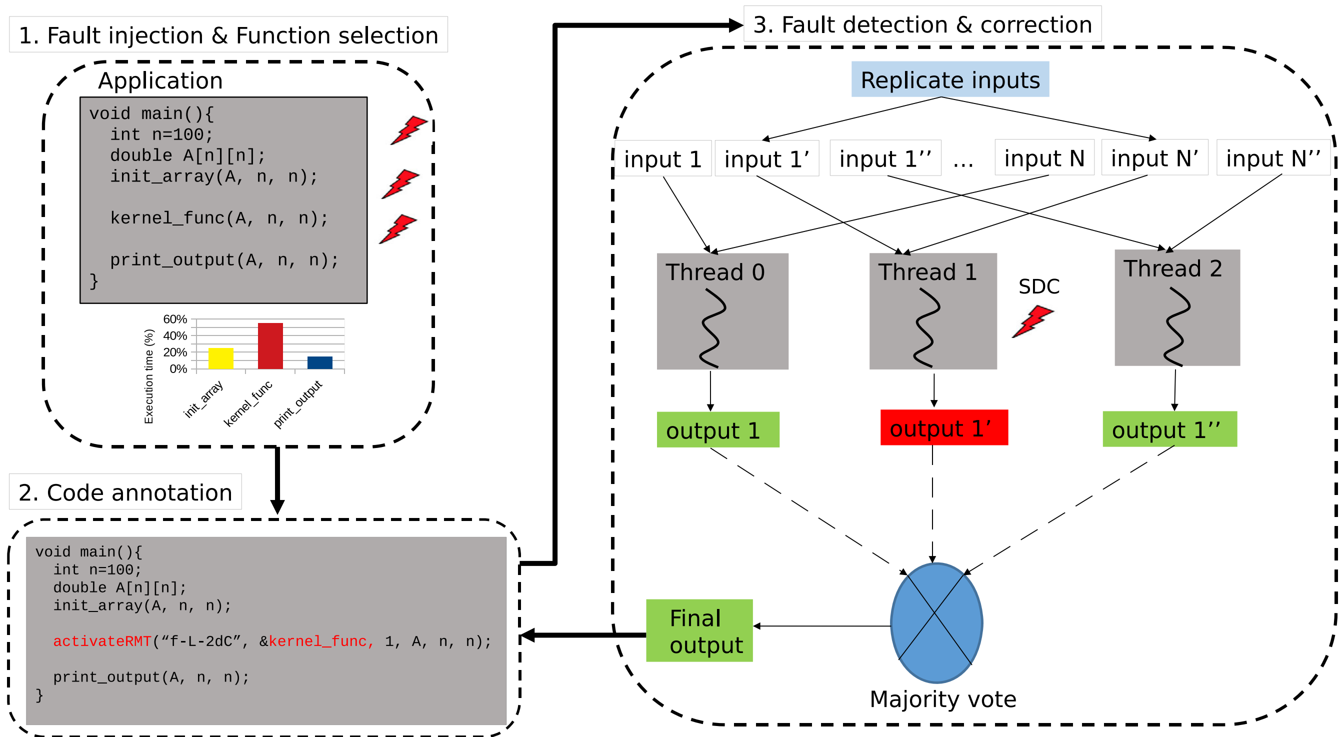

FIGURE 1 Execution flow of our application-level RMT approach[1]

given example), redundant thread execution, and output comparison (the array *B* will be compared at the end of the execution since it is marked with "C"). Further details about region selection for redundancy can be found in our study.[1]

```
/* Our RMT library should be included to invoke activateRMT function. */
#include "RMTlib.h"

void main(){
   ...
# ifdef ENABLE_RMT
    activateRMT("f-L-i-i-d-d-2dC-2d", &kernel_syrk, 6, n,  m, alpha, beta, B, n, n, A, n, m);
# else
    /* Run kernel. */
    kernel_syrk (n, m, alpha, beta, POLYBENCH_ARRAY(B), POLYBENCH_ARRAY(A));
# endif
    ...
}
```

Listing 1: Code annotation for the *syrk* application

3 | METHODOLOGY

3.1 | Use case

To observe the performance impacts of different mapping techniques, we construct a representative workload by selecting widely used HPC kernels from different categories in the PolyBench suite.[3] Specifically, we select *syrk* from *blas*, *doitgen* from *kernels*, *gramschmidt* from *solvers*,

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7622 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

fdtd-2d from *stencils*, *deriche* from *medley*, and *correlation* from *datamining*. The code regions that have a great impact on application output and have a high execution time percentage are annotated to be executed by redundant threads using our previous study.[1] The applications run with 3 redundant threads (to enable TMR) while they are executing the vulnerable code regions (i.e. *main_algorithm* region). In total, there might be at most 18 threads in the 6-application workload at a time. In this study, our problem is to map 18 threads to the available 9 cores efficiently.

It should be noted that the selected functions might spawn new threads and there is no problem to use our redundant execution model. We have already used some parallel applications in our previous work.[1] Our findings in terms of reliability, performance, and energy consumption can be found in that study. In this study, we used serial functions in a serial benchmark suite to manage thread-to-core mappings easily. Otherwise, more than 2 threads might need to share the same core for highly parallel workloads or we should increase the number of cores dedicated to the workload.

3.2 | Different thread-to-core mapping techniques

In this subsection, we present different thread-to-core mapping techniques in two groups, *static* and *dynamic mapping techniques*. In the former one, the applications are analyzed a priori and each thread is paired with another thread based on the applications' execution time, IPC, cache miss, and cache access values. On the other hand, thread-to-core mapping decisions are determined at runtime without requiring any additional knowledge about the applications by observing the IPC, cache miss, and cache access values of threads in the latter one.

3.2.1 | Static mapping techniques

We develop various static mapping techniques such as, *RMT_extime*, *RMT_IPC*, *RMT_cache_miss*, and *RMT_cache_access* all of which require a priori information about the applications. Additionally, We generate six different static mapping techniques in *RMT_static* by using six applications in which each application thread is paired with a different application thread on the same core. All of these static mapping methods map threads to the cores at the beginning of the execution and do not change the mappings throughout the execution. Therefore, thread migration costs are low in these techniques compared to the dynamic ones. We describe each of static mapping techniques below.

RMT_extime

In this technique, each thread of the application with the highest execution time is paired with a thread of the application with the lowest execution time. To do that, the individual execution time of each application is collected while it is running alone. We try to keep each core equally busy with this approach. The individual execution time of each application is shown in Table 1, where we show each paired application with the same color. Specifically, we pair each thread of the *syrk* with each thread of *correlation*, each thread of the *deriche* with each thread of *fdtd-2d*, and each thread of the *gramschmidt* with each thread of *doitgen* on the same core in a 9-core system, as shown in Table 2.

RMT_IPC

In this approach, we pair each thread of the application with the highest IPC with each thread of the application with the lowest IPC. To do that, we need the individual IPC of each application as shown in Table 3. We pair the threads of *syrk* with the threads of *deriche*, the threads of *doitgen* with the threads of *gramschmidt*, and the threads of *fdtd-2d* with the threads of *correlation* on the same core. Therefore, we try to keep the average IPC, as well as the activity, of each core similar in this approach.

TABLE 1 Execution time of the applications

Application name	Ex. time (s)
syrk	48.533
doitgen	29.653
gramschmidt	96.886
fdtd-2d	109.792
deriche	12.377
correlation	76.881

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7622 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 2 Thread-to-core mapping of application threads with *RMT_extime* strategy

Thread 1	Thread 0	
syrk (T1)	correlation (T1)	**Core 0**
syrk (T2)	correlation (T2)	**Core 1**
syrk (T3)	correlation (T3)	**Core 2**
deriche (T1)	fdtd-2d (T1)	**Core 3**
deriche (T2)	fdtd-2d (T2)	**Core 4**
deriche (T3)	fdtd-2d (T3)	**Core 5**
gramschmidt (T1)	doitgen (T1)	**Core 6**
gramschmidt (T2)	doitgen (T2)	**Core 7**
gramschmidt (T3)	doitgen (T3)	**Core 8**

Note: Each (Tx) value near the application name shows the thread number of the corresponding application in the RMT mode.

TABLE 3 IPC values of the applications

Application name	IPC
syrk	1.987
doitgen	2.285
gramschmidt	2.054
fdtd-2d	2.973
deriche	2.352
correlation	1.788

TABLE 4 Cache miss rates of the applications

Application name	Cache miss rate
syrk	36.31%
doitgen	61.08%
gramschmidt	79.44%
fdtd-2d	88.79%
deriche	36.01%
correlation	71.33%

RMT_cache_miss

In this mapping strategy, each thread of the application with the highest cache miss rate is paired with each thread of the application with the lowest cache miss rate. Some applications can benefit from cache usage with a high cache hit ratio. This type of application can be paired with the applications that cannot benefit from cache memories very well. The cache miss rate of each application is shown in Table 4. Therefore, we pair *syrk-gramschmidt*, *doitgen-correlation*, and *fdtd-2d-deriche* application threads on the same core in this mapping strategy.

RMT_cache_access

Similar to the previous strategy, each thread of the application with the highest number of cache accesses is paired with each thread of the application with the lowest number of cache accesses. In this strategy, we focus on the number of cache accesses of each application in the workload rather than cache miss rates to balance the memory access traffic on each core. The relative cache access rates of the applications over the total cache access of the workload are shown in Table 5. Therefore, we pair *syrk-gramschmidt*, *doitgen-fdtd-2d*, and *deriche-correlation* application threads on the same core in this mapping strategy.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7622 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 5 Relative cache access rates of the applications over the total cache access of the workload

Application name	Relative cache access rate
syrk	11.21%
doitgen	0.31%
gramschmidt	11.00%
fdtd-2d	58.15%
deriche	2.94%
correlation	16.62%

TABLE 6 Static mapping techniques

Mapping strategy	Cores 0, 1, 2	Cores 3, 4, 5	Cores 6, 7, 8
S1	*syrk - doitgen*	*gramschmidt - fdtd-2d*	*deriche – correlation*
S2	*syrk - gramschmidt*	*fdtd-2d - deriche*	*correlation - doitgen*
S3	*syrk - fdtd-2d*	*deriche - doitgen*	*correlation - gramschmidt*
S4	*syrk - deriche*	*doitgen - gramschmidt*	*correlation –fdtd-2d*
S5	*syrk - correlation*	*doitgen - fdtd-2d*	*gramschmidt – deriche*

RMT_static

In this technique, we construct six different static mapping techniques in which we pair each application with a different application in the workload. In the first five of them (*S1* through *S5*), each application is paired with every other application as shown in Table 6. It should be noted that we assume that all cores are similar, so the paired application threads may execute on any of the cores. Some of these static mapping techniques are similar with the previous mapping techniques such that *S2* is similar with *RMT_cache_miss* and *S4* is similar with *RMT_IPC*. For the sixth technique, named **S6** (not shown in the table), we pair two threads of the same application to the same core to increase the cache hit ratio of the application. Since there is one remaining thread belonging to each application, we pair them based on *RMT_extime* strategy explained before. With these static mapping techniques, we try to observe the performance gain relative to the original RMT, where we do not interfere with thread-to-core mapping.

3.2.2 | Dynamic mapping techniques

In this part, we develop three dynamic mapping techniques, *RMT_dynamicIPC*, *RMT_dynamic_cache_miss*, and *RMT_dynamic_cache_access* that do not require preliminary information about the applications, and compare them against the static mapping techniques. Thread execution behavior may change dynamically at runtime; therefore, checking the mapping strategy at each scheduling interval may improve the overall system performance. Although the dynamic mapping methods are more adaptive to the changing behavior of the application threads, they may suffer from high thread migration costs. As an example, the workload may have up to 1500 epochs, where we can change thread-to-core mapping decisions. We describe each of the dynamic mapping methods below.

RMT_dynamicIPC

In this technique, we calculate the IPC value of each core as well as each thread by monitoring *cpu cycles* and *instructions* at every epoch. Then, we sort the cores in descending order and the threads in ascending order based on their activities. We map the thread with the highest IPC value onto the core with the lowest IPC value at every epoch. Therefore, we try to keep the activity of each core balanced at each scheduling interval with this approach.

RMT_dynamic_cache_miss

In this method, we analyze the cache miss rates of each thread as well as each core (i.e., the average cache miss rates of threads executing on the same core) at every epoch and map the thread with the highest cache miss rate to the core with the lowest cache miss rate. Our goal is to match a thread that is benefiting from the cache effectively with another thread that is not using it, by observing the cache usage behavior dynamically at runtime.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7622 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

RMT_dynamic_cache_access

This technique is similar to the previous one except that we observe cache access values of each thread as well as each core rather than cache miss rates at every epoch. The thread with the highest cache access value is mapped to the core with the lowest cache access value on average at each epoch. Here, the mapping decisions are taken to balance the network traffic on each core at runtime.

4 | EXPERIMENTAL STUDY

4.1 | Experimental setup

We run our experiments in the MareNostrum4 supercomputer.[7] As hardware configuration, we use 9 cores on special SMT nodes of MareNostrum4 that supports 2 threads/core. Therefore, our problem is efficient mapping of 18 application threads to 18 cores (9 cores, each supports 2 threads) to gain performance. Each node of MareNostrum4 is equipped with 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores for each running at 2.10 GHz frequency for a total of 48 cores per node. There are private 32K of L1 data and instruction caches for each core, a unified 1024K of L2 cache, and a 33MB of L3 cache with 96 GB of main memory for each node.

As a baseline approach (*RMT*), thread-to-core mapping strategy is left to the operating system scheduler. To obtain IPC, cache access, and cache miss rate values in static mapping techniques, we run the application on the previously mentioned hardware configuration a priori using Linux *perf* tool,[8] which uses Performance Monitoring Unit (PMU) registers provided by the hardware. To dynamically observe the IPC, cache miss rate, and cache access values of each thread as well as each core (including the virtual cores), we use Linux *perf_event* interface in the workload source code. The scheduling interval of 100 ms is used for the dynamic methods which is arranged empirically. We utilized *pthread* library functions (*pthread_setaffinity_np* and *pthread_getaffinity_np*) for pinning application threads to the cores. We report the average results of 100 runs for each test case.

4.2 | Evaluation methodology

We use several metrics to evaluate the relative performance of the presented mapping techniques over the original version of the RMT. Specifically, average speedup, harmonic speedup, mean slowdown, and Jain's Fairness Index metrics are used to compare different mapping techniques by considering performance as well as fairness among the applications.

The average speedup of a workload using a mapping technique is defined as the average of per application speedups (in terms of execution time) with respect to the baseline, the original version of the RMT, where we do not interfere with thread-to-core mapping. The average speedup is calculated as follows:[9]

$$Average\ speedup = \frac{1}{n}\sum_{i=1}^{n}\frac{T_i(rmt_orig)}{T_i(rmt_mapping)}, \quad (1)$$

where n is the number of applications in the workload, $T_i(rmt_orig)$ is the execution time of the application i with the original version of RMT, and $T_i(rmt_mapping)$ is the execution time of the application i for the given mapping technique.

The average speedup is a widely used metric to measure performance; however, the result may be biased if there is an aggressive improvement in a single application. Therefore, a *harmonic speedup* metric,[10] which is derived from the *fair speedup* metric,[11] is also utilized. In this case, the relative speedup is defined as the harmonic mean of per application speedups (in terms of execution time) with respect to the original version of the RMT. The harmonic speedup is calculated as follows:[10]

$$Harmonic\ speedup = \frac{n}{\sum_{i=1}^{n}\frac{T_i(rmt_mapping)}{T_i(rmt_orig)}}, \quad (2)$$

Since the harmonic mean of a vector is maximized when the vector elements are equal, this metric considers fairness as well as system performance.

The third metric, mean slowdown, calculates the geometric mean of slowdowns of the applications. It is the lower the better metric that is calculated as follows:[12,13]

$$Mean\ slowdown = \sqrt[|n|]{\prod_{i\in n}\frac{T_i(rmt_mapping)}{T_i(alone))}} - 1, \quad (3)$$

where $T_i(alone))$ is the execution time of the application while it is running alone.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7622 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

On the other hand, there are specific metrics to measure fairness among the applications in a workload. The fourth metric, Jain's fairness index,[14] measures the fairness among n concurrently running applications using the relative slowdown of the applications as follows:[10]

$$Fairness = \frac{\left(\sum_{i=1}^{n} \frac{T_i(rmt_mapping)}{T_i(rmt_orig)}\right)^2}{n \times \sum_{i=1}^{n} \left(\frac{T_i(rmt_mapping)}{T_i(rmt_orig)}\right)^2}. \quad (4)$$

In a fair mapping technique, each application should be equally slowed down. The value of fairness ranges from 0 to 1, and the higher value is better.

4.3 | Experimental results

Figure 2 shows the performance results of a total of thirteen mapping techniques developed over the original version of the RMT based on various metrics. While average and harmonic speedups are the higher the better metrics, the mean slowdown is the lower the better. On the other hand, Jain's Fairness Index is the best when it is equal to 1. As it is seen, there is no static mapping technique that outperforms the original version of the RMT. It clearly shows that thread execution behavior changes over time and static mapping techniques cannot benefit from it. Furthermore, application threads are manually bound to cores, so the cost of migration may also affect the performance negatively if the initial core of the application thread is different than the migrated core. On the other hand, dynamic methods have better average and harmonic speedup values, where *RMT_dynamicIPC* has 8% and 7%, *RMT_dynamic_cache_miss* has 3% and 2%, and *RMT_dynamic_cache_access* has 4% and 3% better speedup values on the average relative to the original version. In terms of mean slowdown, *RMT_dynamicIPC* presents 20% better results than the original RMT compared to the standalone execution of the applications. *RMT_dynamic_cache_miss* and *RMT_dynamic_cache_access* show 7% and 10% better mean slowdown, respectively compared to the alone execution of the applications. In terms of Jain's Fairness index, the dynamic methods have similar values to the original RMT with very close values to 1 (0.996, 0.997, and 0.992, respectively). Although there is a higher number of migrations in the dynamic methods (since scheduling decisions are updated every 100 milliseconds) compared to the static methods, they outperform all static methods by matching the dynamic change of execution behavior for the application threads. Among dynamic methods, *RMT_dynamicIPC* outperforms others on all metrics, which shows that balancing the core activity at every epoch is better than balancing the network traffic for cache usage.

Figure 3 shows the normalized execution time of each application relative to the original RMT. There is a better scheduling technique, either static or dynamic, for each application that performs better than the baseline, the original RMT. However, there is no static mapping technique that performs better than the baseline for all applications. It is clear that any static mapping technique improves the performance of a specific application based on its needs. On the other hand, it decreases the performance of another application. As an example, *RMT_extime* improves the performance of the applications with higher execution times such as *gramschmidt* and *fdtd-2d* significantly. However, it decreases the performance of the applications with smaller execution times such as *syrk*, *deriche*, and *correlation*. Therefore, it cannot improve the performance of the workload in the overall evaluation. On the other hand, *RMT_dynamicIPC* performs better (or very close to the original RMT at worst) for all applications. Specifically, it has 9%, 17%, 6%, 8%, and 4% better performance for the *syrk*, *doitgen*, *gramschmidt*, *fdtd-2d*, and *correlation*. It should be noted that, the performance of the *deriche*

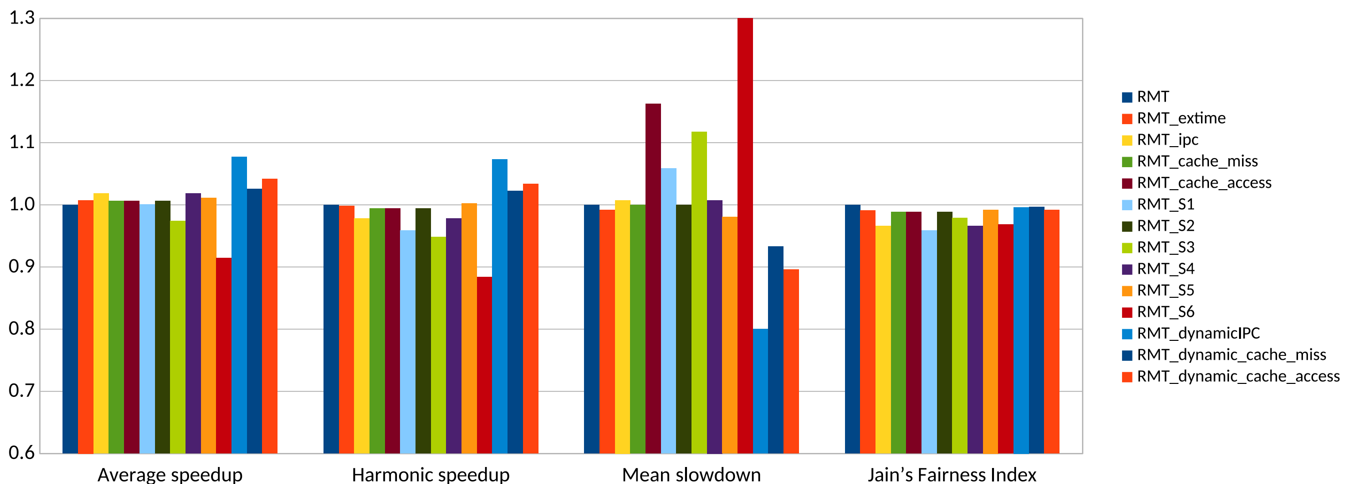

FIGURE 2 Performance results of the proposed mapping techniques over the original version of the RMT based on various metrics

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7622 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7622 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

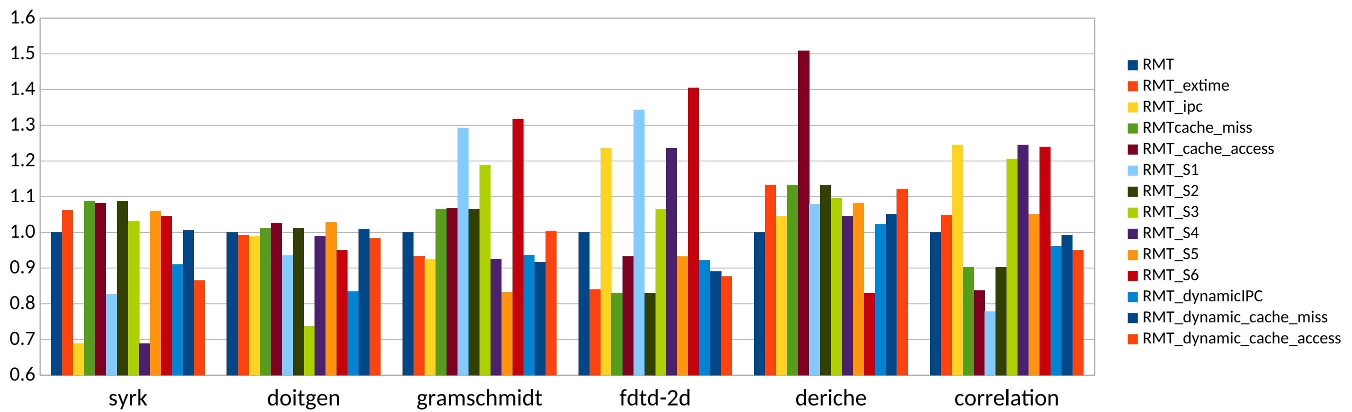

FIGURE 3 Normalized execution time results of applications relative to the original version of RMT

is slightly better for the original RMT with 2%. Other dynamic mapping methods, *RMT_dynamic_cache_miss* and *RMT_dynamic_cache_access*, show better (or very close to the original RMT) performance values for the majority of applications except *deriche*. Specifically, *RMT_dynamic_cache_miss* has very similar performance results for the *syrk*, *doitgen*, and *correlation*, it has 8% and 11% better results for the *gramschmidt* and *fdtd-2d*, and slightly worse performance (5%) for the *deriche* relative to the original RMT. On the other hand, *RMT_dynamic_cache_access* has very similar performance results for the *doitgen* and *gramschmidt*, it has 13%, 12%, and 5% better results for the *syrk*, *fdtd-2d*, and *correlation*, and 12% worse performance for the *deriche* relative to the original RMT. We conclude that when we consider the workload with various applications, *RMT_dynamicIPC* is the best technique with its promising results.

5 | RELATED WORK

There are several studies that improve the performance of RMT techniques by efficient scheduling of threads on manycore architectures.[12,15-19] The studies in the literature differentiate the priorities of redundant threads since there is information flow from leaders to trailers in hardware-level RMT techniques. Kalayappan and Sarangi[12] prioritized the leaders over the trailers and pinned them onto the dedicated cores. On the other hand, trailer threads can be migrated to the remaining cores at each scheduling point. Pouyan et al.[16] force the execution of the trailer threads on different cores than the leader threads by considering load balancing and system throughput. In a couple of studies, redundant task mapping approaches are applied to heterogeneous cores that stem from process variation or hardware aging.[17-19] In another application-level RMT approach,[20] the trailer threads are clustered and mapped onto a single core while the primary threads are mapped to the remaining cores. Contrary to the previous studies, there is no information flow among the redundant threads in our approach. So the redundant threads of an application have equal activities compared to the primary thread in that sense. Additionally, we do not force the threads to execute on the same or different cores based on the dynamic mapping strategy.

There are also studies to improve the performance and fairness of applications by efficient thread-to-core mapping techniques for simultaneous multithreading (SMT) processors. Since it is proven that scheduling in a CMP with more than two cores is an NP-complete problem,[21] the studies more focus on heuristics. These studies make the scheduling decisions based on L1-cache access characteristics of threads,[13] L2 cache access characteristics of threads,[22] the number of ready in-flight instructions,[23] and IPC of threads.[24] The aim of them to schedule threads to cores by capturing different execution phases. Our study also has similar goals with these studies but our primary concern is reliability by proposing redundant thread execution at the application level. Therefore, there are multiple threads with similar activities in our approach.

A set of studies focus on workload characterization and phase-detection using homogeneous and heterogeneous CMPs in literature.[25-28] Eeckhout et al.[25] found that architecture-independent characteristics, such as instruction-mix, register-dependencies, working-set size, and branch predictability can be used to find the similarity among single-threaded applications. Sawalha et al.[26] proposed to schedule threads not only observing phase-change of an application running on heterogeneous CMPs also re-use the information for later phases to reduce the sampling count. Becchi and Crowly[27] proposed using a dynamic thread assignment policy for heterogeneous cores based on IPC values. Ferrorón et al.[28] used the representative regions between barrier points for multithreaded HPC applications to predict cycles, instructions, and cache usage behaviors. Our study has similarities and differences with those studies. First, our study focuses on the thread-to-core mapping based on the RMT implementation of a given set of applications, not focusing on reducing the number of phase-changes or finding similarities among the applications. On the other hand, we found that using IPC values dynamically at runtime has optimal results similar to some of those studies.

6 | CONCLUSION

In this study, we analyze the performance impacts of various static and dynamic thread-to-core mapping strategies on our application-level RMT approach. Thread-to-core mapping strategies become important to improve workload performance in a system with a limited number of cores. The experimental results of our evaluations show that the dynamic mapping technique, *RMT_dynamicIPC*, outperforms the original RMT version, where we left the mapping decisions to the operating system scheduler, up to 17% with an average of 7% based on the harmonic speedup. It also improves the performance of the applications over all the static and the remaining dynamic mapping methods. Therefore, the integration of the dynamic mapping technique, *RMT_dynamicIPC*, with our application-level RMT approach improves the performance of a given workload significantly in a system with a limited number of cores.

ACKNOWLEDGMENTS

This work was completed while the first author, Sanem Arslan, was visiting researcher at Barcelona Supercomputing Center, Barcelona, Spain. Sanem Arslan had received financial support from the Scientific and Technological Research Council of Turkey (TUBITAK) under the program BIDEB 2219 during this work. The preliminary version of this work was presented at the seventh High Performance Computing Conference (BASARIM2022).[29]

CONFLICT OF INTEREST

The authors declare that there is no conflict of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Sanem Arslan https://orcid.org/0000-0003-3019-7070

REFERENCES

1. Arslan S, Unsal O. Efficient selective replication of critical code regions for SDC mitigation leveraging redundant multithreading. *J Supercomput*. 2021;77:14130-14160. doi:10.1007/s11227-021-03804-6
2. Dorai G, Yeung D. Transparent threads: Resource sharing in SMT processors for high single-thread performance. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*; 2002: 30-41.
3. Pouchet LN. PolyBench/C. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/; 2016.
4. Shivakumar P, Kistler M, Keckler S, Burger D, Alvisi L. Modeling the effect of technology trends on the soft error rate of combinational logic. *Proceedings International Conference on Dependable Systems and Networks*; 2002: 389-398.
5. Rehman S, Shafique M, Henkel J. *Reliable software for unreliable hardware: A cross layer perspective*. Springer; 2016.
6. Bautista-Gomez L, Zyulkyarov F, Unsal O, McIntosh-Smith S. Unprotected computing: A large-scale study of dram raw error rate on a supercomputer. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis IEEE*; 2016: 645–655.
7. MareNostrum 4. https://www.bsc.es/marenostrum/marenostrum/technical-information; 2017.
8. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page; 2015.
9. Nemirovsky M, Tullsen DM. *Multithreading Architecture*. 1st ed. Morgan & Claypool Publishers; 2013.
10. Wang J, Abu-Ghazaleh N, Ponomarev D. Controlled contention: Balancing contention and reservation in multicore application scheduling. *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*; 2015: 946-955.
11. Chang J, Sohi GS. Cooperative cache partitioning for chip multiprocessors. *Proceedings of the 21st Annual International Conference on Supercomputing ICS'07*; 2007: 242–252.
12. Kalayappan R, Sarangi SR. FluidCheck: A redundant threading-based approach for reliable execution in manycore processors. *ACM Trans Archit Code Optim*. 2015;12(4):1-26. doi:10.1145/2842620
13. Akturk I, Ozturk O. Adaptive Thread Scheduling in Chip Multiprocessors. *Int J Parallel Program*. 2019;47:1-31.
14. Jain R, Chiu D, Hawe W. *A quantitative measure of fairness and discrimination for resource allocation in shared systems. Technical report*. Digital Equipment Corporation, DEC-TR-301; 1984.
15. Chen K, der Brüggen vG, Chen J. Reliability optimization on multi-core systems with multi-tasking and redundant multi-threading. *IEEE Trans Comput*. 2018;67(4):484-497.
16. Pouyan F, Azarpeyvand A, Safari S, Fakhraie SM. Reliability aware throughput management of chip multi-processor architecture via thread migration. *J. Supercomput*. 2016;72(4):1363-1380.
17. Chen K, Chen J, Kriebel F, Rehman S, Shafique M, Henkel J. Task mapping for redundant multithreading in multi-cores with reliability and performance heterogeneity. *IEEE Trans Comput*. 2016;65(11):3441-3455.
18. Dong J, Zhang L, Han Y, Yan G, Li X. Variation-aware scheduling for chip multiprocessors with thread level redundancy. *Proceedings of 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*; 2009: 17-22.
19. Kriebel F, Rehman S, Shafique M, Henkel J. ageOpt-RMT: Compiler-driven variation-aware aging optimization for redundant multithreading. *Proceedings of 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*; 2016: 1-6.
20. Hukerikar S, Teranishi K, Diniz PC, Lucas RF. RedThreads: An interface for application-level fault detection/correction through adaptive redundant multithreading. *Int J Parallel Program*. 2018;46(2):225-251. doi:10.1007/s10766-017-0492-3

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7622 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

21. Jiang Y, Shen X, Jie C, Tripathi R. Analysis and approximation of optimal co-scheduling on chip multiprocessors. *Proceedings of 2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*; 2008: 220-229.
22. Settle A, Kihm J, Janiszewski A, Connors D. Architectural support for enhanced SMT job scheduling. *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques, 2004 (PACT 2004)*; 2004: 63-73.
23. El-Moursy A, Garg R, Albonesi DH, Dwarkadas S. Compatible phase co-scheduling on a CMP of multi-threaded processors. *Proceedings of the 20th IEEE International Parallel Distributed Processing Symposium*; 2006.
24. Parekh SS, Eggers S, Levy H, Jack L. *Thread-Sensitive Scheduling for SMT Processors. tech. rep.* University of Washington; 2000.
25. Eeckhout L, Sampson J, Calder B. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. *Proceedings of the IEEE Workload Characterization Symposium*, 2005: 2-12.
26. Sawalha L, Wolff S, Tull MP, Barnes RD. Phase-guided scheduling on single-ISA heterogeneous multicore processors. *Proceedings of the 2011 14th Euromicro Conference on Digital System Design*; 2011: 736-745.
27. Becchi M, Crowley P. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. *Proceedings of the 3rd Conference on Computing FrontiersCF'06*. Association for Computing Machinery; 2006:29-40.
28. Ferreron A, Jagtap RS, Bischoff S, Rusitoru R. Crossing the architectural barrier: Evaluating representative regions of parallel HPC applications. *Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*; 2017: 109-120.
29. Arslan S, Ünsal O. Efficient thread-to-core mapping for application-level redundant multithreading. *Proceedings of the 7th National High Performance Computing Conference (BAŞARIM 2022)*; 2022.

How to cite this article: Arslan S, Ünsal O. Efficient thread-to-core mapping alternatives for application-level redundant multithreading. *Concurrency Computat Pract Exper*. 2023;e7622. doi: 10.1002/cpe.7622

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7622 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License