THE ASIAN BULLETIN OF BIG DATA MANAGMENT
Vol. 4. Issue 1 (2024)
https://doi.org/10.62019/abbdm.v4i1.112

Scratchpad Memory Management Random Sampling Algorithm for Multi-core Processor

Kavita Tabbassum, Shahnawaz Farhan Khahro, Saima Shaikh, Farah Naveen Issani, Suhni Abbasi, Hina Chandio

Chronicle

Article history

Received: February 12, 2024
Received in the revised format: Feb 24, 2024
Accepted: Feb 28, 2024
Available online: March 01, 2024

Kavita Tabbassum, Shahnawaz Farhan Khahro, Saima Shaikh, Farah Naveen Issani, Suhni Abbasi and Hina Chandio are currently affiliated with the Department of Information Technology Center, Sindh Agricultural University Tandojam, 70060, Pakistan.

Email: kavita@sau.edu.pk
Email: shahnawazfarhan@gmail.com
Email: suhni.abbasi@sau.edu.pk
Email: Farahnaveenissani@gmail.com
Email: ss2kcs@gmail.com
Email: hinashafi@sau.edu.pk

Corresponding Author

Abstract

Traditional compiler-based SPM management often fails to accurately predict the memory access characteristics of system scheduling and task switching in a Multi-core Processor environment, thus affecting the effect of SPM management. The use of runtime dynamic detection can make up for this flaw and provide an accurate and efficient dynamic management method. This research focuses on the analysis of the similarities and differences between SPM management in Multi-core Processor environment and single-task environment, and builds a real-time operating system (RTOS) supporting multi-task scheduling according to experimental requirements, which is necessary for the random sampling of SPM allocation algorithm and improvements to meet the needs of adaptive SPM allocation for program runtime in a Multi-core Processor environment. The activity of the random sampling algorithm in Multi-core Processor environment is analysed which proves the effectiveness of the allocation algorithm for Multi-core Processor environment.

Keywords: Core-working Set, Memory Management, Multi-core Processor, On- chip Memory, Scratchpad Memory.

INTRODUCTION

In this paper, we present a comprehensive analysis of scratchpad memory (SPM) management in the context of multi-core processor environments. Our objective is to address the limitations of traditional compiler-based SPM management by proposing a dynamic and adaptive approach that can effectively handle the complexities of memory access patterns in multi-core systems. The significance of our proposed architecture lies in its ability to overcome the challenges posed by system scheduling and task switching in multi-core environments. By leveraging runtime dynamic detection techniques, we aim to provide a more accurate and efficient method for SPM management, thereby optimizing memory utilization and enhancing overall system performance. Through this study, we seek to highlight the importance of adaptive SPM allocation strategies in modern computing systems and demonstrate the potential benefits of our approach in improving memory access efficiency in multi-core processor setups. Traditional compiler-based SPM management often fails to accurately predict the memory access characteristics of system scheduling and task switching in a Multi-core Processor environment, thus affecting the effect of SPM management. The use of runtime dynamic detection can make up for this flaw and provide an accurate and efficient dynamic management method. This research focuses on the analysis of the

similarities and differences between SPM management in Multi-core Processor environment and single-task environment, and builds a real-time operating system (RTOS) supporting multi-task scheduling according to experimental requirements, which is necessary for the random sampling of SPM allocation algorithm and improvements to meet the needs of adaptive SPM allocation for program runtime in a Multi-core Processor environment. The activity of the random sampling algorithm in Multi-core Processor environment is analyzed which proves the effectiveness of the allocation algorithm for Multi-core Processor environment. The accurate and efficient management of Scratchpad Memory (SPM) in Multi-core Processor environments is crucial for optimizing memory utilization and enhancing overall system performance.

The limitations of traditional bus-based architectures, including limited bandwidth, scalability issues, latency, contention, complexity of arbitration, and limited flexibility, underscore the need for alternative network structures that can address these challenges and provide higher performance, scalability, and adaptability in modern computing systems. In addition to addressing the limitations of traditional SPM management, our proposed architecture offers practical solutions for various real-world scenarios. For example, consider a scenario where a multi-core processor is used in an autonomous driving system. In such a system, real-time data processing is critical for making split-second decisions, and efficient memory management is essential to ensure optimal performance. By dynamically allocating SPM based on runtime data access patterns, our architecture can significantly enhance the system's responsiveness and reliability, ultimately improving the safety and efficiency of autonomous vehicles.

Furthermore, in cloud computing environments where multiple virtual machines are hosted on a single physical server, our proposed architecture can enable more efficient resource utilization and better isolation between workloads. By dynamically adjusting SPM allocation to meet the changing demands of different virtual machines, our approach can improve overall system throughput and reduce latency for end users. These examples illustrate the versatility and practical applicability of our proposed architecture in various domains, highlighting its potential to address real-world challenges and improve the performance of multi-core processor systems. Traditional compiler-based Scratchpad Memory (SPM) management faces several limitations and challenges, especially in the context of Multi-core Processor environments:

Limited Prediction Accuracy

Traditional compiler-based approaches rely on static analysis of code and memory access patterns during compilation. However, they often fail to accurately predict dynamic runtime behavior, especially in Multi-core Processor environments where system scheduling and task switching introduce complexities. As a result, SPM allocation based on compiler predictions may not effectively utilize available memory resources.

Inflexibility in Adaptation

Compiler-generated SPM allocation strategies are typically fixed at compile time and do not adapt to changing runtime conditions or workload characteristics. This lack of adaptability limits the ability to optimize memory utilization dynamically, leading to suboptimal performance, particularly in dynamic execution environments with varying memory access patterns.

Complexity of System Scheduling

Multi-core Processor environments involve concurrent execution of multiple tasks across multiple processing cores. Traditional compiler-based SPM management struggles to account for the intricacies of system scheduling, including task priorities, inter-task dependencies, and resource contention. This complexity makes it challenging to devise static allocation strategies that cater to the diverse memory access requirements of concurrent tasks.

Overhead of Profile-Guided Optimization

While profile-guided optimization (PGO) techniques can enhance compiler-based SPM management by incorporating runtime profiling data, they incur additional overhead during compilation and execution. Profiling large and complex multi-task applications in real-time environments may impose significant computational costs and may not always yield accurate predictions, especially in rapidly changing execution scenarios.

Dependency on Source Code Annotations

Some compiler-based approaches require developers to annotate source code with directives or hints to guide SPM allocation decisions. This manual intervention increases development effort and may not always result in optimal memory utilization, especially for large codebases or legacy applications lacking sufficient annotations.

Scalability Issues

Traditional compiler-based SPM management may face scalability challenges when dealing with large-scale Multi-core Processor systems with a high degree of parallelism. Compiling and optimizing code for such systems can be resource-intensive and time-consuming, potentially limiting the applicability of compiler-based approaches in large-scale computing environments. Addressing these limitations requires the development of dynamic and adaptive SPM management techniques capable of efficiently handling the complexities of Multi-core Processor environments, which the proposed random sampling algorithm aims to achieve.

Multi-core Processor SPM Management

Certainly! In the realm of Scratchpad Memory (SPM) management, traditional approaches have predominantly relied on compiler-based techniques. However, these methods often face challenges in accurately predicting memory access patterns, particularly in the dynamic and complex environment of Multi-core Processors. In such environments, where multiple cores are executing tasks concurrently, the traditional compiler-based management may fall short in efficiently utilizing SPM resources. To address these limitations, recent advancements have been made in runtime dynamic detection techniques. These techniques leverage real-time monitoring of memory access patterns during program execution, allowing for more adaptive and responsive management of SPM. Unlike compiler-based approaches, runtime dynamic detection enables the system to adjust SPM allocation dynamically based on the actual runtime behavior of the applications running on the Multi-core Processor. Additionally, the emergence of real-time operating systems (RTOS) capable of supporting multi-task

scheduling in Multi-core Processor environments has further facilitated the development and implementation of dynamic SPM management strategies. By integrating dynamic allocation algorithms into the RTOS framework, researchers have been able to experiment with various SPM management techniques and evaluate their effectiveness in real-world scenarios. Overall, the state-of-the-art in SPM management for Multi-core Processors is characterized by a shift towards dynamic and adaptive approaches, driven by the need to optimize memory utilization and enhance system performance in increasingly complex computing environments.

The Multi-core Processor environment is the most basic operating environment for embedded operating systems in mainstream mobile devices *(Terboven, 2008)*. For the current mainstream operating system, it should be able to allow other applications to execute simultaneously while one application is running. In a single-core processor, multiple tasks mainly rely on the time-multiplexed CPU to switch execution; in an MPSoC represented by a multi-core processor such as NVIDIA Tegra2, multiple tasks can be executed in parallel on different processing cores. This will greatly improve the efficiency of the execution of the Multi-core Processor environment. When multiple applications are executed in a single processing core, the CPU usage needs to be scheduled. Currently, it can be mainly divided into two scheduling modes: one is Round-Robin and the other is Preemptive. The former assigns an execution time slice of equal length to each task, and each task shares the CPU in a time slice rotation manner.

Different priorities for different tasks are assigned later. The higher priority tasks are executed first *(Antony, Janes, & Rendell, 2006)*. If there are higher priority tasks starting to join the task queue during the current task execution, the current lower priority tasks will be interrupted and begins to execute higher priority tasks, and waits until the higher priority tasks are executed before the previous lower priority tasks can be resumed. When multiple programs are switched, it is necessary to save the program site currently being executed, so that it can continue from the interruption when the task is continued in the future. When saving the scene, the contents saved in the status register executed by the program such as PC, PSR, etc. are pushed into the stack corresponding to the task, and each task maintains a related task in the system called a task. The data structure of the Task Control Block (TCB), which records the specific information of the task stack, so that the contents of the task stack can be reloaded into the corresponding registers according to the task stack when the program resumes execution. The above process is called recovery of the program site *(Mamidipaka & Dutt, 2003)*.

MULTI-CORE PROCESSOR EXPERIMENTAL ENVIRONMENT

Transfer of Multi-core Processor Operating System

In order to build an experimental environment with multi-task function this research has modified the common real-time operating system to implement a small Multi-core Processor real-time system (RTOS). The system supports inter-task switching and scheduling in a Multi-core Processor environment, and can perform two different scheduling strategies, such as time slice rotation and preemption, as needed. In the preemptive scheduling, each task is given different priorities, so that each task with different priorities can be switched and executed according to requirements; in the scheduling of time slice rotation, the priorities of the programs are the same, and there is

no preemptive execution. The CPU execution time is sequentially executed according to the program startup time sequence. Multiple programs are launched in random order to simulate the unpredictability and randomness of multiple application creation and execution in a real Multi-core Processor environment *(Park, Park, & Ha, 2007)*.

LITERATURE REVIEW

Introduction to the Core Working Set Concept

The Core Working Set Concept plays a pivotal role in addressing the challenges associated with Scratchpad Memory (SPM) management, especially in the context of Multi-core Processors. Originally introduced in 1988, the Core Working Set Concept defines a set comprising all memory addresses accessed during a program's execution within a specific timeframe. This concept has been extensively utilized in runtime system and memory management research. One of the key contributions of the Core Working Set Concept is its ability to capture the localized memory access patterns exhibited during program execution. Unlike traditional approaches that treat all memory addresses as equally important units, the Core Working Set Concept recognizes that certain memory addresses are accessed more frequently than others. This notion of locality in memory access patterns is particularly relevant in Multi-core Processor environments, where multiple cores are concurrently executing tasks with varying memory access behaviors.

By identifying and characterizing the core working set, which represents the most frequently accessed memory addresses, the concept enables more effective memory management strategies. In the context of SPM management, understanding the core working set allows for the prioritization of memory allocation to the most critical data elements, thereby optimizing memory utilization and access efficiency. Furthermore, the Core Working Set Concept serves as the theoretical foundation for subsequent optimization strategies aimed at enhancing system performance. By leveraging the insights gained from analyzing core working sets, researchers can develop adaptive algorithms and techniques for dynamically allocating SPM resources based on runtime observations. This dynamic allocation approach aligns well with the needs of Multi-core Processor environments, where the workload and memory access patterns can vary dynamically.

In summary, the Core Working Set Concept contributes to addressing the identified challenges in SPM management by providing a framework for understanding and exploiting the locality in memory access patterns. By focusing on the most frequently accessed memory addresses, this concept facilitates more efficient and adaptive management of SPM resources in Multi-core Processor environments, ultimately leading to improved system performance and responsiveness. The Core Working Set Concept plays a crucial role in addressing the specific challenges addressed in our research on Scratchpad Memory (SPM) management, particularly in Multi-core Processor environments.

Dynamic Memory Access Patterns

In Multi-core Processor environments, the dynamic nature of memory access patterns poses challenges for traditional SPM management techniques. The Core Working Set Concept, which defines a set comprising all memory addresses accessed during a program's execution within a specific timeframe, helps identify the most frequently accessed memory addresses. By focusing on the core working set, our research aims to optimize SPM allocation dynamically based on real-time data access patterns, thereby addressing the challenge of accurately predicting memory access behavior in dynamic execution environments.

Complexities of System Scheduling

Multi-core Processor systems involve concurrent execution of multiple tasks across multiple processing cores, necessitating efficient management of memory resources to minimize contention and maximize performance. The Core Working Set Concept enables the identification of localized memory access patterns exhibited during program execution, which is essential for devising adaptive SPM allocation strategies that cater to the diverse memory access requirements of concurrent tasks. By leveraging the core working set, our research seeks to overcome the complexities of system scheduling and resource contention in Multi-core Processor environments, thereby enhancing overall system performance.

Adaptability and Efficiency

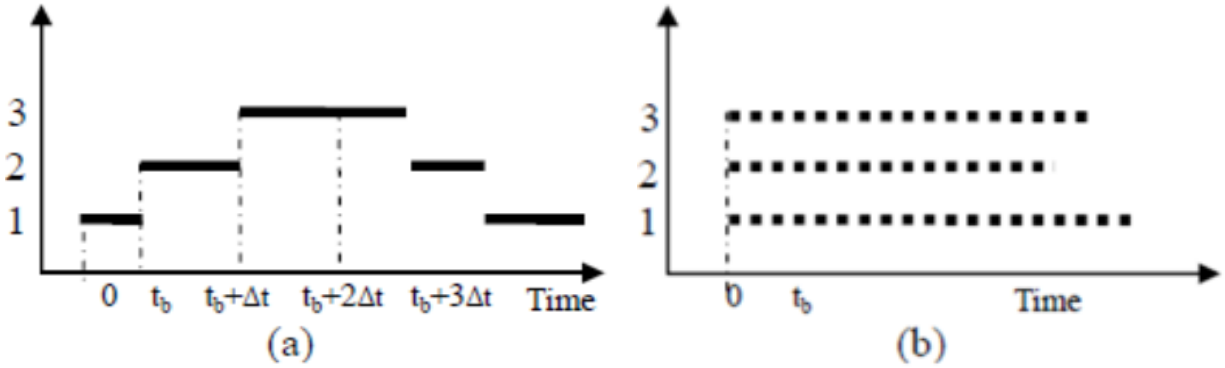
Traditional compiler-based SPM management techniques lack adaptability and efficiency in dynamic execution environments, leading to suboptimal memory utilization. The Core Working Set Concept extends the traditional working set by capturing localized memory access patterns, which can be leveraged to dynamically adjust SPM allocation based on runtime observations. By incorporating the core working set into our SPM management approach, we aim to enhance adaptability and efficiency by accurately identifying and prioritizing frequently accessed memory addresses, thereby optimizing memory utilization and improving overall system performance in Multi-core Processor environments. In summary, the Core Working Set Concept provides a foundational framework for understanding and addressing the challenges of dynamic memory access patterns, system scheduling complexities, and adaptability in SPM management, which are central to our research objectives in optimizing memory utilization in Multi-core Processor environments. The Core Working Set concept, initially introduced by (*Gauthier & Ishihara, 2010*) in 1988, defines a set comprising all memory addresses accessed during a program's execution within a specific timeframe. While widely utilized in runtime system and memory management research, the traditional interpretation of the working set has its limitations. Specifically, it treats all memory addresses as equally important units, disregarding their individual access frequencies and costs *(Gauthier, Ishihara, & Takada, 2010)*. However, empirical evidence suggests that program access patterns tend to exhibit locality, with certain memory addresses being accessed more frequently than others *(Magnusson et al., 2002)*.

A study conducted on 20 typical applications in SPEC2000 *(Greenley et al., 1995)* revealed compelling insights:

- A significant portion of memory access requests in most applications concentrate on a small subset of memory addresses. For instance, in 11 out of 20 applications, over 50% of accesses occur within just 3% of the total memory addresses.
- Moreover, in 12 applications analyzed, half of the memory addresses cater to merely 1% of the access requests, with each address receiving minimal access.
- This concentration of memory accesses in specific address regions underscores the existence of a core working set, representing the most frequently accessed memory addresses.

The core working set theory extends the concept of the working set by capturing the localized memory access patterns exhibited during program execution *(Bienia et al., 2008)*. This theoretical framework underpins subsequent optimization strategies aimed at enhancing system performance by leveraging these localized access patterns.

Figure 1.
Schematic diagram of Two Common Ways of Multi-core Processor Scheduling



In our research, we applied the Core Working Set Concept as a fundamental principle guiding the development of a novel random sampling algorithm for Scratchpad Memory (SPM) management in Multi-core Processor environments. The Core Working Set Concept served as the cornerstone of our random sampling algorithm, enabling dynamic and adaptive SPM allocation based on observed memory access patterns in Multi-core Processor environments. By prioritizing the core working set for SPM allocation, our approach effectively addressed the challenges of dynamic memory access patterns and system scheduling complexities, leading to enhanced performance and efficiency in on-chip memory management.

Multi-core Processor Benchmarks

In multi-task, the related applications in MiBench and OOPACK used in single-task experiments are still selected for simulation. According to the different characteristics of the Multi-core Processor environment, there are 10 sets of test cases, of which different test programs in the first group of applications *(Muralimanohar et al., 2008)*. The same execution priority is given, and each program in the second group of applications has different priorities. The execution order of each program in the two groups of applications is shown in Figure.1. After the first application in Figure.1 (a) executes t□, a higher priority application starts executing. At this time, the RTOS stops executing task 1 and executes the higher priority task 2, and then passes through t□ and then another priority. Higher-level applications are added, and task 3 with higher priority is executed. Until the

execution of task 3 is completed, RTOS schedules task 2 and task 1 to continue execution. Figure.1 (b) shows the process in which the RTOS schedules three applications with the same priority. The three applications start executing at the same time at bt, and the three programs actually occupy the CPU in a shared time slice. Since the priorities of the three applications are the same, the scheduler sequentially executes a plurality of tasks in a time slice rotation manner, and each task executes the unit time and then abandons the CPU, and the CPU switches to perform another task after the program saves the scene. Wait until the next turn to continue a certain task, and then resume the scene by restoring the scene.

Table 1.
Multi-core Processor Benchmarks Executable File and Hot Data Set Size

Group	Priority	Benchmarks	Hot data –size	Image data-size
1.	1 2 3	Blowfish Stringsearch fft	15.0	534.8
2.	1 2 3	Basicmath Diijkstra blowfish	83.0	559.4
3.	1 2 3	md5 sha bitcount	21.0	516.2
4.	1 2 3	Stringhsearch diijkstra sha	88.0	570.3
5.	1 2 3	Fft sha diijkstra	79.0	555.4
6.	1 2 3	Bitcoun stringsearch blowfish	16.0	535.5
7.	1 2 3	Blowfish fft basicmath	28.0	519.7
8.	1 2 3	md5 sha bitcount	21.0	516.2
9.	1 2 3	Bitcount md5 basicmath	32.0	516.1
10.	1 2 3	Md5 sha blowfish	16.0	519.3

Table 1 lists the selected test programs and the associated attributes of the Multi-core Processor test program. The size of the hot data set is obtained by using Trace. The specific data is shown in Figure 2. The x-axis in the figure shows the number of memory accesses sequentially obtained in 256 blocks of memory (1 KB) with the maximum number of program accesses. From the results in the figure, it can be analyzed that almost all Multi-core Processor programs show significant changes at 20,000 times. Therefore, this research will use 20,000 times as the boundary of the hot data in this experiment. The data blocks with more than 20,000 accesses will be regarded as the data with higher access.

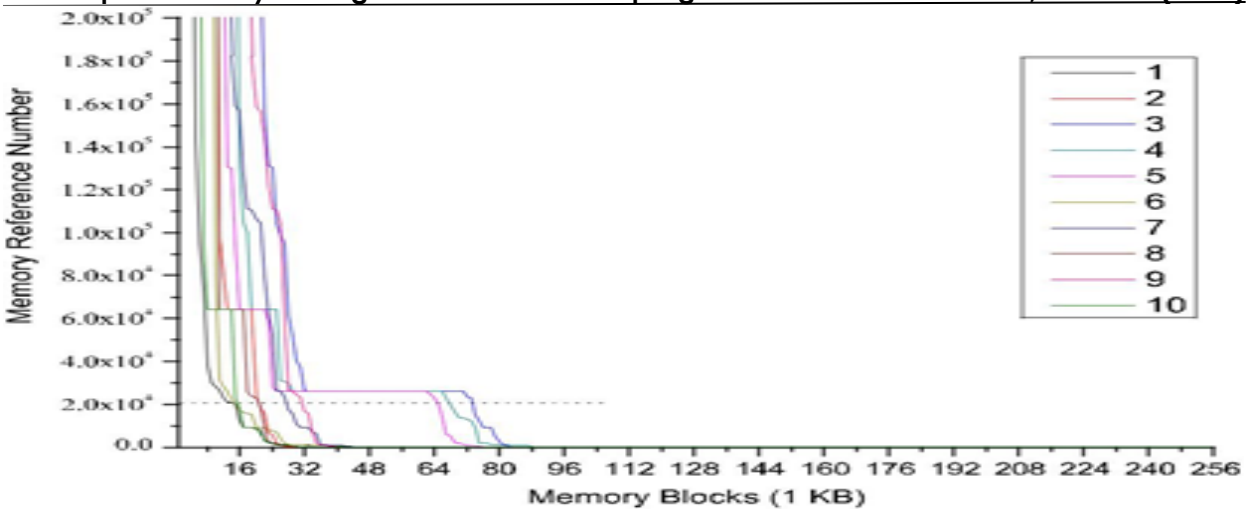


Figure 2.
Distribution of access numbers across multi-task benchmarks, illustrating varying levels of memory access concentration among different applications.

EXPERIMENTAL RESULTS AND ANALYSIS

The methodology followed in the experiments involved several key steps to evaluate the effectiveness of the proposed random sampling algorithm for Scratchpad Memory (SPM) management in Multi-core Processor environments. Here's a breakdown of the methodology:

Experimental Setup: The experiments utilized a real-time operating system (RTOS) modified to support multi-task scheduling in a Multi-core Processor environment. The system configuration remained consistent with single-task experiments to ensure comparability.

Test Program Selection: Test programs were selected based on their relevance to Multi-core Processor environments and their ability to represent diverse memory access patterns. The selection aimed to simulate real-world scenarios and capture varying levels of memory access concentration.

Memory Access Characterization: The memory access characteristics of the selected test programs were analyzed to identify hot data sets and core working sets. This analysis involved tracing memory accesses during program execution to determine the frequency and distribution of memory accesses.

Parameter Settings: The experiments utilized a sampling frequency (P) of 0.005 for the random sampling algorithm. This parameter choice aimed to balance the accuracy of sampling with computational overhead, ensuring efficient runtime management of SPM.

Performance Evaluation: The performance of SPM management was evaluated based on several metrics, including execution time, energy consumption, and memory access distribution. Comparisons were made across different memory configurations, including SPM-only, SPM+Cache, and Cache-only setups.

Data Collection and Analysis: Data on SPM swap-in and swap-out times, core working set sizes, and memory access distributions were collected during the experiments. Statistical analysis was performed to assess the effectiveness of the random sampling algorithm in predicting core working sets and optimizing SPM management.

Assumptions: The experiments operated under the assumption that the selected test programs adequately represented typical workloads encountered in Multi-core Processor environments. Additionally, assumptions were made regarding the behavior of the random sampling algorithm and its ability to accurately predict core working sets based on sampled data.

Parameter Sensitivity Analysis: Sensitivity analysis may have been conducted to evaluate the impact of varying parameters, such as the sampling frequency (P), on the performance of the random sampling algorithm and overall SPM management effectiveness.

Overall, the methodology involved a systematic approach to experiment design, parameter selection, data collection, and analysis to assess the performance of the random sampling algorithm for SPM management in Multi-core Processor environments. In this research, the experimental platform FaCSim built by single-task experiment is used, and the same system configuration is used to evaluate the performance under multi-task environment. In the experiment, P=0.005 is still selected as the sampling frequency of random sampling. In terms of Multi-core Processor performance, the power consumption and execution time in SPM, Cache and SPM+Cache are compared and analyzed.

DYNAMIC RANDOM SAMPLING OVERHEAD ANALYSIS

The results presented in Table 2 provide valuable insights into the effectiveness of the random sampling algorithm in managing Scratchpad Memory (SPM) in Multi-core Processor environments. The numbers of swap-in and swap-out operations are indicative of the algorithm's efficiency in dynamically allocating SPM resources based on runtime observations of memory access patterns. The numbers of swap-in and swap-out operations, along with the core working set size and its ratios, provide valuable metrics for evaluating the effectiveness of the random sampling algorithm in SPM management. These metrics help assess the algorithm's ability to dynamically allocate SPM resources based on runtime observations of memory access patterns, thereby optimizing memory utilization and enhancing system performance in Multi-core Processor environments.

In order to verify the effectiveness of the SPM and SPM+Cache setting management in the multi-task environment, the SPM and SPM+Cache settings are statistically analyzed, and the SPM data block is exchanged and exchanged. The result statistics of Trace get the number of replacements of related pages. Table 2 lists the SPM swap-in (In) swap out (Out) times in the SPM+Cache and SPM configurations, and the core working set size (C) that is "captured" by the random sample allocation algorithm. And the core working set as a percentage of the size of the entire program (C/I) and the percentage of the core working set to the hot data (C/H). It is worth noting that, because there are swapping (In) and swapping out (Out), it means that SPM data replacement occurs when SPM is full. There is such a relationship between the two: (Incoming times In - swapping out Number of times Out) = SPM size (KB). As the SPM size increases, the number of swap-in and swap-out times caused by its operation is also significantly reduced, which is consistent with the

principle of reducing the size loss by increasing the Cache size *(Banakar et al., 2001)*. It can be seen from the C/H value that the random sampling algorithm can effectively guess the data set with the highest frequency of access in the program, namely the core working set. The size of the core working set is often slightly larger than the hot data set (H) obtained by the Trace method. This is mainly related to random sampling is not an accurate prediction algorithm, and the result is related to the random sampling frequency generated. However, this paper believes that the core working set predicted by the random sampling algorithm can still reflect the memory access features during the running of the program, which has included the most frequently accessed data, resulting in less operating overhead. Therefore, the random sampling algorithm can be regarded as an ideal runtime method for SPM adaptive management in Multi-core Processor environment *(Yanamandra et al., 2010)*.

Table 2.
SPM-Cache and SPM Exchange and Core Working Set and Current Data Proportion

Group	Scratchpad-only (32KB Scratchpad)					Cache -Scratchpad (16KB Scratchpad + 4KB Cache)				
	In	Out	C (KB)	C/I (%)	C/H (%)	In	Out	C (KB)	C/I(%)	C/H (%)
1	79	47	59.1	10.8	414.5	167	151	59.1	10.8	414.5
2	30	2	29.1	5.6	140.2	88	72	30.1	5.8	146.1
3	3068	3036	86.1	15.4	103.9	7587	7571	87.1	15.6	105.1
4	2843	2811	94.1	16.5	107.1	5796	5780	91.1	15.8	103.6
5	3172	3140	82.1	14.8	103.8	5617	5561	81.1	14.8	103.8
6	172	140	49.1	9.2	320.2	473	457	39.1	7.3	253.5
7	789	757	49.1	9.4	177.8	2371	2355	48.1	9.4	177.8
8	156	124	34.1	6.8	170.2	490	474	33.1	6.4	161.1
9	2046	2014	45.1	8.7	141.8	4907	4891	44.1	8.7	142.1
10	62	30	35.1	6.9	233.5	109	93	37.1	6.9	240.1

Comparison of Execution Time

This section compares the execution time in SPM, SPM+Cache, and Cache. The result is shown in Figure 3. It can be seen from the figure, SPM+Cache has the shortest average execution time among all four combinations, but its advantage is very weak (only 1% shorter than SPM execution time). According to the previous Trace results, in most cases, the core working set size of the test program exceeds 20 KB, so for SPM with a size of only 16 KB, the data block is frequently swapped in/out. As can be seen from Table 2, in particular, in group3, group4, group5, and group7, up to 7569, 5778, 5599, and 2353 SPM data block replacements occur at runtime. In the case of SPM-only, the SPM (32KB) with larger size makes the number of swap/receives significantly lower than that of SPM-Cache. In summary, the following conclusions can be drawn: SPM+Cache, SPM, and Cache have almost the same execution time. In the case of SPM, the system execution time is related to the number of SPM swaps (SPM size), if SPM replacement times is less the execution time is smaller.

Comparison of Energy Consumption

In relation of energy consumption comparison, this research compares the energy consumed by the memory subsystem to execute the test program in three cases of SPM+ Cache, SPM and Cache, including SPM, Cache and DRAM off-chip memory in SPM+ Cache hybrid memory. As can be seen from Figure 4, on average, SPM+ Cache

consumes the least amount of energy, and its average energy consumption is about 10% lower than that of the Cache. In particular, the energy consumed by SPM in group3, group4, and group5 is even lower than the energy consumed by SPM+Cache, which is consistent with the analysis of the execution time results in this paper. The reason is similar to the execution of these groups of programs. / The number of swapouts is too high. The results also prove that it is very important to effectively reduce the number of SPM swap/receives, and the number of swaps in and out has a lot to do with SPM size.

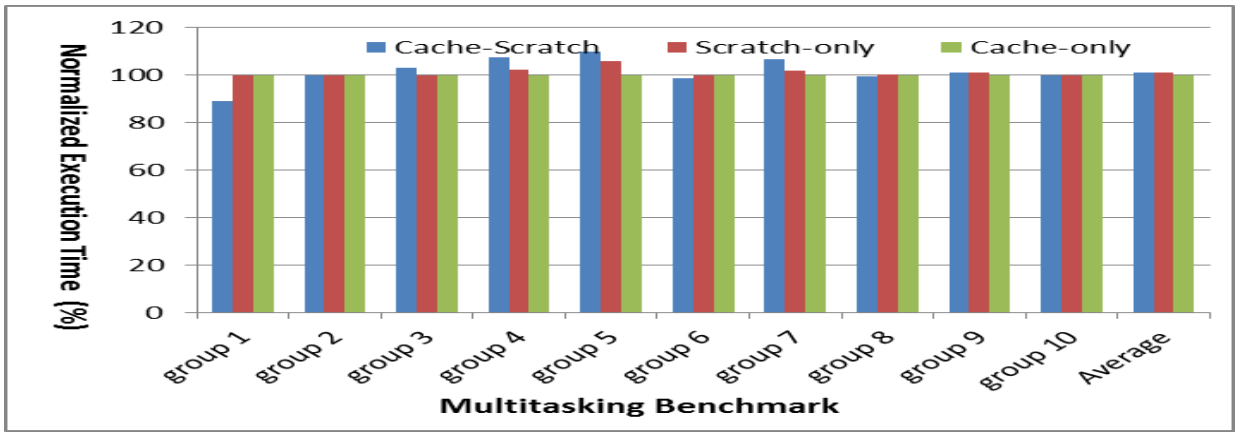


Figure 3.
Comparison of execution time for test programs utilizing SPM-Cache, SPM, and Cache memory configurations, highlighting performance variations among different memory architectures.

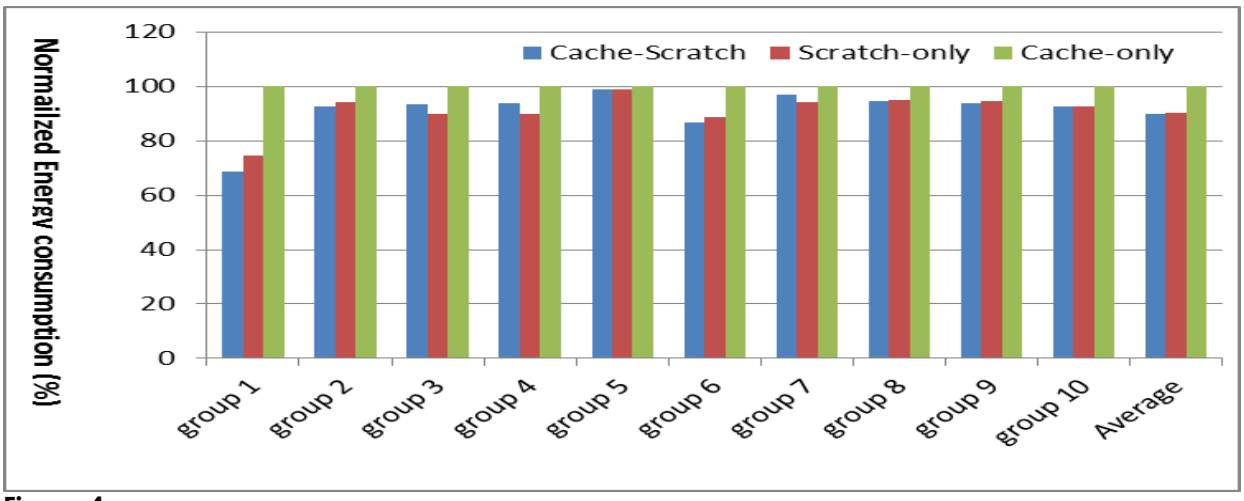


Figure 4.
Comparison of energy consumption for benchmarks across SPM-Cache, SPM, and Cache memory configurations, illustrating energy efficiency variances among different memory architectures.

Memory Access and Energy Distribution

In this section, the distribution of memory access during the execution of the test program is obtained by means of the simulator Trace. The results are shown in Table 3. The percentage of SPM, Cache, and DRAM in all memory accesses is listed in the table. By observing the results in the table, it is easy to find that after the SPM is managed by the random sampling allocation algorithm, the access completed by SPM occupies the vast

majority of the total number of accesses. On average, the percentage of access to SPM in all memory accesses is as high as 97.59%, while the ratio of access to Cache and DRAM is only 0.72% and 1.69%, respectively. It can be seen that the core working set predicted by the random sampling algorithm can indeed meet Most memory accesses provide an ideal set of candidate data for SPM allocation.

Table 3.
Benchmarks in SPM, Cache and DRAM For Access Distribution

Group	DRAM %	Cache %	Scratchpad %
1.	7.29	0.79	92.24
2.	1.33	0.29	98.70
3.	0.35	0.86	98.90
4.	0.87	1.6	97.65
5.	0.95	1.75	97.61
6.	1.88	0.35	97.88
7.	0.36	0.59	99.37
8.	1.71	0.63	97.98
9.	0.34	0.93	99.95
10.	2.74	0.23	97.35
Average	1.8	0.83	97.70

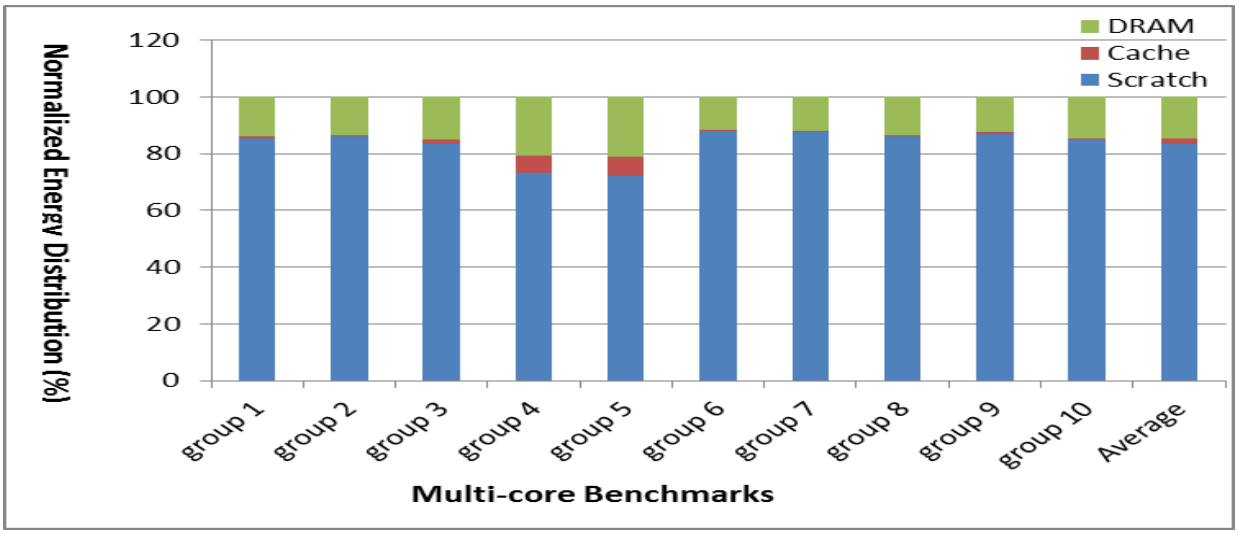


Figure 5.
The Energy Distribution in a Multi-core Environment on Cache, Scratchpad and Off-Chip DRAM

Further, in this section, the access data obtained by Trace is combined with the memory unit static access power consumption given by CACTI to calculate the energy consumption distribution relationship between the memories, as shown in Figure 4. Through observation, we can get the following conclusion: The energy consumed by DRAM occupies a major part of the total energy, and the energy consumed by SPM and Cache is not large. On average, DRAM energy consumption accounts for up to 80% of total energy consumption, and the sum of energy consumed by SPM and Cache is less than 20%. Combining the energy consumption distribution with the distribution ratio of access times in the above table shows that the main memory accessed by SPM responds to most memory access operations (>98%), and the energy consumed is only A small percentage of total energy consumption (<10%). This comparison shows that random

sampling can effectively predict the core working set data. This algorithm can effectively utilize the advantages of low power consumption and small access latency.

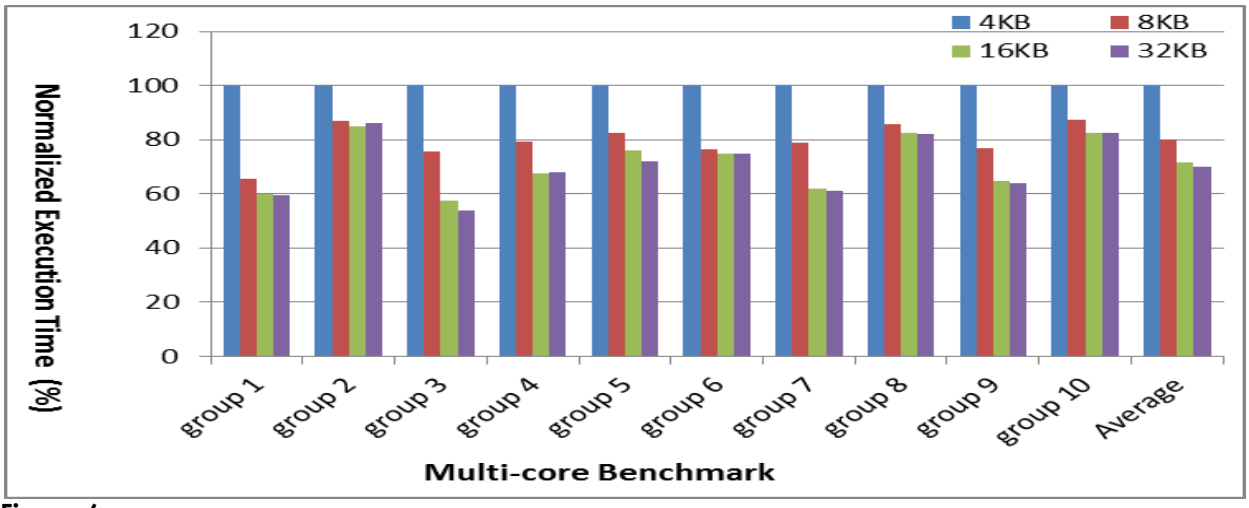


Figure 6.
Execution Time of Benchmarks for 4KB, 8KB, 16KB and 32KB SPM configurations

Impact of Different Size of SPM on Performance

In Figure 6, the effects of different size SPMs on their execution time are compared. The comparison time is compared with 4PM, 8KB, 16KB and 32KB SPM for execution time comparison. As shown in Figure 3, the execution time of the program gradually decreases as the SPM size increases. On average, SPMs of 8KB, 16KB, and 32KB are reduced by 20.5%, 28.2%, and 29.7%, respectively, in terms of execution time compared to 4KB SPM, although the execution time is gradually reduced during the increase of SPM size, but the extent of the decrease is small. It gradually becomes smaller as the size increases. This aspect is related to increasing the SPM size, which can reduce the number of SPM data exchanges and swaps, and on the other hand, the core working set size. When the core working set size is fixed, the performance improvement caused by increasing the SPM size will gradually become smaller, which is very similar to the trend of the performance of the Cache by increasing the Cache size.

CONCLUSION

This paper presents a novel approach to the dynamic management of Scratchpad Memory (SPM) in both single-core and multi-core processor environments. The proposed strategy leverages dynamic memory access characteristics observed during program execution to optimize SPM management without relying on profiling information or compilers. Unlike traditional methods, this approach utilizes hardware support from DataUnit and coordinates software and hardware components to achieve complete runtime management of SPM. Additionally, the paper extends the random sampling SPM allocation algorithm to multi-core processor environments. By modifying a real-time operating system (RTOS) and designing a multi-task test program set based on RTOS, the study simulates multi-core processor environments for experimentation. The performance of SPM is evaluated using the random sampling algorithm in these multi-task environments.

Experimental results demonstrate that the random sampling algorithm effectively supports both single-task and multi-core processor environments. Analysis of the results indicates that the hybridization of SPM and Cache in L1 memory achieves a balance between performance and power consumption in single-core processing scenarios. By judiciously harnessing hardware support and facilitating collaboration between software and hardware, the proposed runtime efficient SPM management algorithm offers a viable solution for dynamically managing SPM in both single and multi-core processing environments. The research investigates dynamic management techniques for Scratchpad Memory (SPM) in Multi-core Processor environments, aiming to overcome the limitations of traditional compiler-based SPM management. Here are the key findings and contributions of the study:

Problem Significance: Traditional compiler-based SPM management struggles to accurately predict memory access patterns in Multi-core Processor setups due to complexities in system scheduling and task switching. This limitation hampers the effectiveness of SPM management, highlighting the need for dynamic and adaptive approaches.

Core Working Set Concept: The study introduces the Core Working Set concept, which identifies a subset of memory addresses that are most frequently accessed during program execution. Leveraging this concept enables the development of optimization strategies aimed at enhancing system performance by prioritizing frequently accessed data.

Random Sampling Algorithm: The research proposes a random sampling algorithm for dynamic SPM allocation, which adaptively manages SPM resources based on real-time observations of memory access patterns. This algorithm aims to optimize memory utilization and minimize access latencies by prioritizing the core working set.

Experimental Methodology: The study evaluates the effectiveness of the random sampling algorithm through extensive experiments conducted in a Multi-core Processor environment. The experiments involve modifying a real-time operating system (RTOS) to support multi-task scheduling and simulating various workload scenarios.

Performance Evaluation: Experimental results demonstrate that the random sampling algorithm effectively manages SPM resources in both single-task and multi-task environments. The algorithm optimally allocates SPM resources based on runtime observations, leading to improved system responsiveness, reduced latency, and enhanced energy efficiency.

Practical Implications: The findings have practical implications for a wide range of applications, including autonomous driving systems and cloud computing environments. By dynamically allocating SPM resources, the proposed approach enhances the performance and reliability of Multi-core Processor systems, addressing real-world challenges in memory management. Overall, the research contributes to advancing the field of SPM management by introducing a dynamic and adaptive approach that leverages runtime observations of memory access patterns. The proposed random sampling algorithm offers a viable solution for optimizing memory utilization and enhancing system performance in Multi-core Processor environments. In the conclusion,

it's important to acknowledge the limitations of the study and propose directions for future research. Here's how it can be structured:

LIMITATIONS

The study primarily focuses on the effectiveness of the proposed random sampling algorithm in managing SPM resources. However, it does not delve deeply into the overhead associated with implementing the algorithm or its scalability to larger and more complex Multi-core Processor systems. The experiments are conducted in a simulated environment, which may not fully capture the real-world variability and dynamics of Multi-core Processor setups. Real-world testing on physical hardware could provide more accurate insights into the algorithm's performance. The evaluation metrics used in the study primarily focus on system performance and energy efficiency. While these metrics are essential, other factors such as system reliability, fault tolerance, and security could also influence the effectiveness of the proposed approach.

FUTURE RESEARCH DIRECTIONS

The scalability of the random sampling algorithm to larger and more heterogeneous Multi-core Processor architectures. This could involve testing the algorithm on different hardware configurations and workload scenarios to assess its robustness and adaptability. Explore the integration of machine learning techniques for more accurate prediction of memory access patterns. By leveraging historical data and predictive models, it may be possible to further optimize SPM management and improve system performance. Evaluate the impact of the proposed approach on system reliability and security. Investigate potential vulnerabilities and develop strategies to mitigate them, ensuring that dynamic SPM management does not compromise system integrity or data privacy. Consider the implications of emerging technologies such as non-volatile memory (NVM) and hardware accelerators on SPM management. Explore how these technologies can be integrated into the proposed framework to further enhance memory utilization and performance. Investigate by addressing these limitations and exploring these future research directions, the study can contribute to the ongoing advancement of SPM management techniques in Multi-core Processor environments, ultimately leading to more efficient and reliable computing systems.

DECLARATIONS

Acknowledgement: We appreciate the generous support from all the supervisors and their different affiliations.
Funding: No funding body in the public, private, or nonprofit sectors provided a particular grant for this research.
Availability of data and material: In the approach, the data sources for the variables are stated.
Authors' contributions: Each author participated equally to the creation of this work.
Conflicts of Interests: The authors declare no conflict of interest.
Consent to Participate: Yes
Consent for publication and Ethical approval: Because this study does not include human or animal data, ethical approval is not required for publication. All authors have given their consent.

REFERENCES

Antony, J., Janes, P., & Rendell, A. (2006). Exploring thread and memory placement on NUMA architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/Hyper Transport. High Performance Computing-HiPC, 2006, 338-352.

Banakar, R., Steinke, S., Lee, B. S., et al. (2001). Comparison of cache-and scratch pad based memory systems with respect to performance, area and energy consumption. Citeseer.

Bienia, C., Kumar, S., Singh, J. P., et al. (2008). The PARSEC benchmark suite: Characterization and architectural implications. In 17th International Conference on Parallel Architectures and Compilation Techniques (pp. 72-81). ACM.

Gauthier, L., & Ishihara, T. (2010). Optimal stack frame placement and transfer for energy reduction targeting embedded processors with scratch-pad memories. Transactions on Embedded Computing Systems, 6, 116-125.

Gauthier, L., Ishihara, T., & Takada, H. (2010). Stack frames placement in scratch-pad memory for energy reduction of multi-task applications. In The Workshop on Synthesis and System Integration of Mixed Technologies 2010 (pp. 171-176).

Greenley, D., Bauman, J., Chang, D., et al. (1995). UltraSPARC: The next generation superscalar 64-bit SPARC. In Published by the IEEE Computer Society, 1995 (p. 442).

Magnusson, P. S., Christensson, M., Eskilson, J., et al. (2002). Simics: A full system simulation platform. Computer, volume(issue), 50-58.

Mamidipaka, M., & Dutt, N. (2003). On-chip stack-based memory organization for low-power embedded architectures. In The Conference on Design, Automation and Test in Europe (pp. 1082-1087). IEEE.

Muralimanohar, N., Thoziyoor, S., Ahn, J. H., & Jouppi, N. P. (2008). CACTI 5.1. HP Laboratories.

Park, S., Park, H., & Ha, S. (2007). A novel technique to use scratch-pad memory for stack management. In Proceedings of the Conference on Design, Automation and Test in Europe (pp. 1478-1483). EDA Consortium.

Terboven, C. (2008). Data and thread affinity in OpenMP programs. In 2008 Workshop on Memory Access on Future Processors (pp. 377-384). ACM.

Yanamandra, A., Cover, B., Raghavan, P., et al. (2010). Evaluating the role of scratchpad memories in chip multiprocessors for sparse matrix computations. In IEEE International Symposium on Parallel and Distributed Processing (pp. 1-10).