BenchCouncil Transactions on Benchmarks, Standards and Evaluations 4 (2024) 100189

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Research Article

# MultiPoint: Enabling scalable pre-silicon performance evaluation for multi-task workloads☆

Chenji Han [a,b,*], Xinyu Li [a,b], Feng Xue [a,b], Weitong Wang [a,b], Yuxuan Wu [c], Wenxiang Wang [b,c], Fuxin Zhang [a]

[a] SKLP, Institute of Computing Technology, CAS, Beijing, China
[b] University of Chinese Academy of Sciences, Beijing, China
[c] Loongson Technology, Beijing, China

ARTICLE INFO

*Keywords:*
Performance modeling
Performance evaluation
Multi-task workloads

ABSTRACT

With the core numbers integrated within single processors growing and the fast development of cloud computing, performance evaluation for multi-core systems is increasingly crucial. It is typically conducted by executing multi-task workloads, exemplified by SPEC CPU Rate, to measure metrics like system's throughput. In response, several sampling-based methods have been developed for their pre-silicon performance evaluation. Nevertheless, these methods involve directly capturing multi-task checkpoints, which presents scalability issues of significant storage and time overheads. Therefore, enabling more scalable performance evaluation remains a critical problem.

In this work, we propose MultiPoint to enable scalable pre-silicon performance evaluation for multi-task workloads. It is noted that in the multi-task workloads of interest, each task executes independently without inter-task communication. Therefore, MultiPoint is motivated to construct the required multi-task checkpoints by recovering multiple single-task checkpoints across different cores and guarantee their smooth execution through address remapping and shuffling. We implemented MultiPoint on the Emulator Accelerator and assessed its evaluation accuracy against its post-silicon Loongson 3A6000 processor. Using SPEC CPU 2017 as the benchmark, MultiPoint achieved the estimation errors of 6.20%, 5.45%, and 6.99% for Rate 2, Rate 4, and Rate 8, respectively, achieving comparable accuracy compared to direct multi-task checkpointing but in a more scalable manner with substantially 86.0% lower storage and 93.7% less time overheads.

## 1. Introduction

**Background**. Pre-silicon performance evaluation is becoming increasingly critical, considering the continuous rise in fabrication costs and prolonged verification periods. For the single-task workloads, many representative sampling-based methods [1–8], such as SimPoint [1–5], have been developed. These methods involve profiling and clustering program's code signatures and eventually selecting the simulation points, as detailed in Section 2.1. In our practice, SimPoint achieved an average performance estimation error of 1.85% for SPEC CPU 2017 Rate 1 [9] with a speedup of 477 times. Besides, with the core numbers integrated within single processors growing and the fast development of cloud computing, performance evaluation for multi-core systems is becoming increasingly crucial. It is typically conducted by concurrently executing multiple workloads [10,11], exemplified by SPEC CPU 2017 Rate, to measure metrics like system's throughput [12,13].

**Research Problem**. In response, several SimPoint-like method [14–19] have been developed for pre-silicon performance evaluations of multi-task workloads. These methods typically concatenate code signatures of concurrently executed programs to cluster and select representative simulation points. The multi-task checkpoints of selected simulation points are then captured, containing the architecture-level status of register values and the memory content of these tasks, which could be recovered on the target multi-core designs to conduct performance evaluations. However, the direct checkpointing of multi-task workloads involved in these methods presents **scalability issues** of significant storage and time overheads.

☆ The authors would like to thank the helpful discussions with Ruiyang Wu, Yuxiao Chen, and Hongze Tan. This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDC05020100).

* Corresponding author at: University of Chinese Academy of Sciences, Beijing, China.
*E-mail address:* hanchenji16@mails.ucas.ac.cn (C. Han).

https://doi.org/10.1016/j.tbench.2025.100189
Received 16 October 2024; Received in revised form 27 December 2024; Accepted 23 January 2025
Available online 13 February 2025

Firstly, checkpoints of multi-task workloads can only be employed to evaluate multi-core system with specific core numbers. For instance, the checkpoints captured on a four-core system can only be used to evaluate four-core designs. Therefore, any change in the assessed core numbers would necessitate recapturing the corresponding multi-task checkpoints, leading to a waste of computational resources as well as extra storage and time overheads.

Secondly, directly captured multi-task checkpoints require large storage overheads. The total size of a multi-task checkpoint is approximately equal to the combined size of checkpoints for the involved individual tasks. Besides, the content in multi-task checkpoints could be duplicated with certain single-task checkpoints. The storage overheads would be exacerbated as the core number increases.

**Key Idea**. It is noticed that in the multi-task workloads of interest, such as SPEC CPU Rate, each task executes independently without inter-task communication. It motivates us to restore multiple single-task checkpoints on different cores to construct the required multi-task checkpoints, which essentially involves concurrently running multiple single-core operating systems on the multi-core system with the shared memory. Consequently, the scalability issues of substantial storage and time overheads the direct multi-task checkpointing could be eliminated.

**Requirements**. However, composing the required multi-task checkpoint by multiple single-task checkpoints presents several requirements, as discussed below.

Firstly, it is required to guarantee multiple single-task checkpoints to smoothly execute at a multi-core system with shared memory. In the single-task checkpoint, the operating system manages memory within a fixed range. When multiple single-task checkpoints are restored simultaneously without special handles, they would inadvertently attempt to use the same memory regions. This overlap in memory spaces can prevent the tasks from executing normally.

Secondly, it is necessary to make the memory address characteristics similar to that in realistic multi-task workload executions, which are scattered across the memory and interleaved with each other, as shown in Fig. 3. This is because, differences in memory address characteristics could result in varying impacts on certain $\mu$Arch structures, like the last-level cache, thus compromising the accuracy of performance evaluation, as discussed in Section 6.3.

**Our Work**. Corresponding to these requirements, we propose MultiPoint to enable scalable pre-silicon performance evaluation for multi-task workloads. MultiPoint is capable of composing the required multi-task workloads by simultaneously recovering multiple single-task workloads across different cores and ensuring their smooth concurrent execution through physical address remapping and shuffling. Specifically, MultiPoint introduces a checkpoint loader and proposes a synchronization mechanism to support the concurrent recovery of multiple single-task checkpoints at the Emulator Accelerator and enable their simultaneous initiation of execution. Besides, MultiPoint introduces a software-transparent address transform layer to support the concurrent smooth execution of multiple single-task checkpoints. To ensure the normal execution of these single-task checkpoints, MultiPoint remaps the memory requests from different cores to separate memory regions to avoid their interference. Furthermore, to mirror the actual memory access address characteristics of multi-task workloads, MultiPoint shuffles their memory addresses to make them scatter across the memory space and interleave with each other.

To sum up, the contributions of this work include:

1. We proposed MultiPoint to compose multi-task checkpoints through multiple single-task checkpoints, which enables scalable pre-silicon performance evaluation for multi-task workloads.
2. We implemented the evaluation routine of MultiPoint on the Emulator Accelerator and assessed its performance evaluation accuracy against its post-silicon Loongson 3A6000 commercial processor [20].
3. We evaluated MultiPoint on the SPEC CPU 2017 Rate benchmark. MultiPoint achieved the score estimation errors of 6.20%, 5.45%, and 6.99% for Rate 2, Rate 4, and Rate 8, respectively, achieving comparable estimation accuracy compared to the direct multi-task checkpointing but in a more scalable manner with 86.0% lower storage and 93.7% less time overheads.

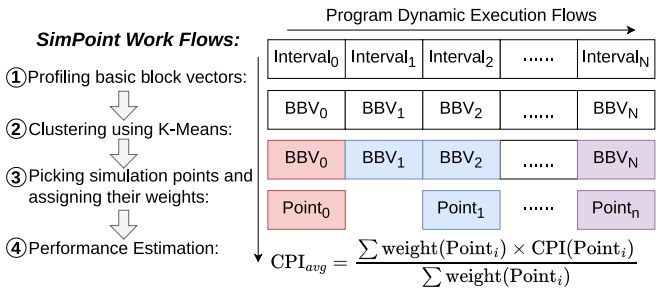



**Fig. 1.** Procedures of the representative sampling method, using SimPoint as an illustrative example.

The remaining part of this work is organized as follows. Sections 2 and 3 give the background and motivation of MultiPoint. Section 4 details the method of MultiPoint. Section 5 introduces the experiment environment. Section 6 discusses the evaluation results. Section 7 lists the related works. Section 8 concludes this work.

## 2. Background

### 2.1. Evaluation for single-task workload

For pre-silicon performance evaluation of single-task workload, many representative sampling-based methods [1–8], such as SimPoint [1–5], are well-developed and widely employed in both the academia [21] and industries [22]. The motivation behind representative sampling is that programs' execution is composed of several recurring phases, instead of being chaos. Fig. 1 illustrates general procedures of the representative sampling, using SimPoint as the example. Specifically, SimPoint divides the program's dynamic execution flows into non-overlapping intervals with fixed lengths. For each interval, SimPoint profiles its frequency vector of basic blocks (BBVs) (①), which is a sequence of consecutive instructions with only one entrance and one exit. After profiling, the K-Means algorithm is leveraged to cluster these program intervals (②), after setting parameters of the maximum allowed cluster numbers `maxK`, projected dimension `dim`, and Bayesian information criterion threshold `BIC`. As a result, program intervals closest to the centroid of each cluster are selected as the simulation points to represent the average performance of each cluster. Besides, the simulation points are assigned weights according to the number of program intervals that they represent (③). Finally, the program's performance could be extrapolated by the weighted average of performance of these representative simulation points' performance (④). In our practice, SimPoint achieved an average performance estimation error of 1.85% for SPEC CPU 2017 benchmark [9] with a speedup of 477 times.

### 2.2. Evaluation for multi-task workload

Sampling-like methods [14–19] have also been developed for homogeneous and heterogeneous multi-task workloads. Specifically, as demonstrated in Fig. 2, these methods involve concatenating BBVs of concurrently executed tasks (①) and utilizing the resultant concatenated vectors as program's code signatures (②). Following the SimPoint-like procedures in Fig. 1, these methods cluster these signatures and select the representative simulation points. Next, their corresponding multi-task checkpoints are captured (③). However, the direct checkpointing of multi-task workloads involved in these methods brings scalability issues of significant storage and time overheads.

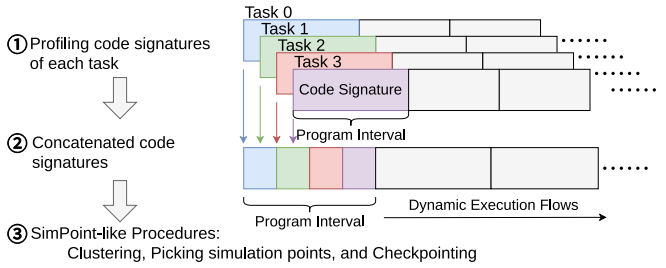



**Fig. 2.** Procedures of the SimPoint-like method for multi-task workload evaluations, using 4 tasks as the example.

## 3. Key idea of MultiPoint

It is noticed that, in scenarios of multi-task workloads such as SPEC CPU Rate, each task executes independently without inter-task communication. It motivates us to utilize multiple single-task checkpoints to agilely compose required checkpoints of multi-task workloads, which essentially involves concurrently running multiple single-core operating systems on the multi-core system with the shared memory. Correspondingly, such an approach must satisfy the following two requirements to ensure the correctness of program execution and maintain the evaluation accuracy.

### 3.1. Motivations of address isolation

Firstly, it is noted that in the single-task checkpoint, the operating system manages memory within a fixed range and is typically immutable during runtime. When multiple single-task checkpoints are restored simultaneously without special handles, they would inadvertently attempt to use the same memory regions. Consequently, these concurrently executed tasks would interfere with each other, preventing them from executing successfully.

> ***Requirement 1.*** *Physical addresses of memory requests from different cores should be isolated from each other.*

Address isolation can be achieved by assigning different memory offsets to requests originating from different cores.

### 3.2. Motivations of address interleaving

Secondly, it is necessary to make the memory address characteristics of concurrently executed multiple single-task checkpoints similar to that in realistic multi-task workload executions, which are scattered across the memory and interleaved with each other. This is because differences in memory address characteristics could result in varying impacts on certain $\mu$Arch structures. For example, in homogeneous workloads such as SPEC CPU 2017, if only the address isolation is implemented, the low bits of address accessed by different single-task checkpoints for semantically identical memory are identical, and only their highest address bits are distinct. This would result in significant cache set conflicts in the shared last-level cache.

Specifically, Fig. 3 presents the probability density distributions of physical address usage by different cores of four memory-intensive programs in SPEC CPU 2017 Rate 4. Other programs in SPEC CPU 2017 behave similarly and are thus not presented here. In Fig. 3, the program's physical address usage is collected via the Linux kernel interface, and the probability density distributions are calculated by the Gaussian KDE method [23]. Fig. 3 illustrates that memory addresses utilized by different cores are interleaved with each other rather than being distinctly isolated. It is noted that the specific distributions of memory usage by multi-task workloads can vary across

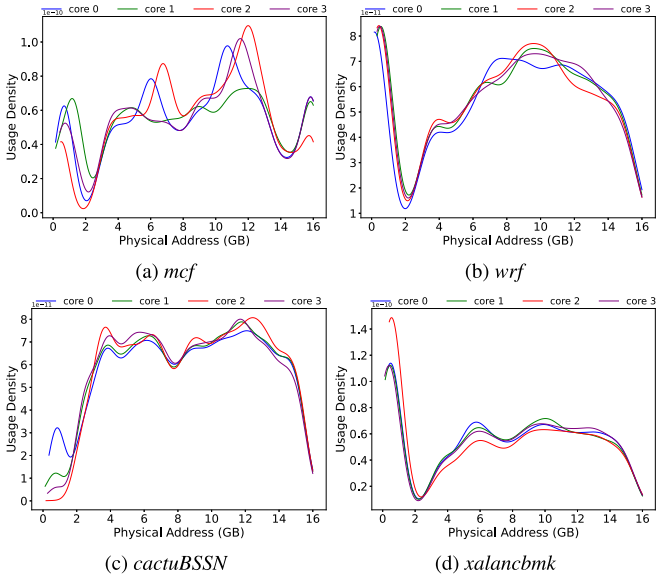



**Fig. 3.** Probability density distributions of physical address usage by different cores of four memory-intensive programs in SPEC CPU 2017 Rate 4. Instead of being isolated, their memory addresses are interleaved with each other.

different executions based on the real-time situation of the memory fragmentation [24]. Despite these variations, physical addresses allocated for different tasks are typically interleaved as a result of interleaved processing of their memory allocation requests.

> ***Requirement 2.*** *Physical addresses of memory requests from different cores should be interleaved with each other.*

Address interleaving can be achieved by employing software-transparent address shuffling algorithm.

## 4. Method of MultiPoint

In this work, we propose MultiPoint to enable scalable performance evaluation for multi-task workloads. As illustrated in Fig. 4, MultiPoint is composed of three critical procedures as follows. ***Checkpoint Recovery***: Multiple checkpoints of different or identical single-task workloads are concurrently recovered across different processor cores (①). Besides, a sync mechanism is implemented among the processor cores to guarantee their simultaneous initiation of evaluation. ***Address Remapping***: To guarantee the smooth execution of these concurrently executed single-task checkpoints, MultiPoint remaps memory requests from different processor cores to different memory regions to prevent them from interfering with each other (②). ***Address Shuffling***: To maintain the performance evaluation accuracy of constructing multi-task checkpoints via multiple single-task checkpoints, MultiPoint shuffles memory requests from different cores to make them scatter and interleave in the memory space (③). Collectively, MultiPoint introduces a software-transparent address transform mechanism to support the concurrent smooth execution of multiple single-task checkpoints. Consequently, the scalability issues of substantial storage and time overheads the direct multi-task checkpointing could be avoided.

### 4.1. Checkpoint recovery

MultiPoint is designed to concurrently recover multiple single-task checkpoints across different processor cores and employs a sync mechanism to guarantee their simultaneous initiation of evaluation. The detailed recovery process of a single-task checkpoint by the proposed checkpoint loader is illustrated in Fig. 5. The checkpoint consists of

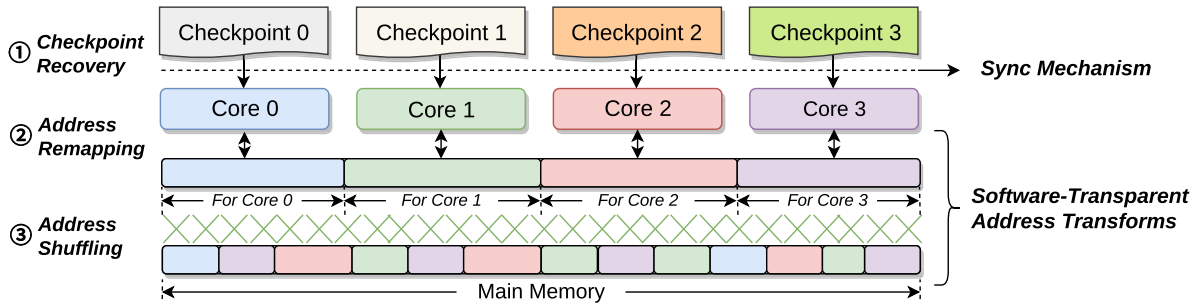



**Fig. 4.** Procedures of MultiPoint for performance evaluation of multi-task workload.

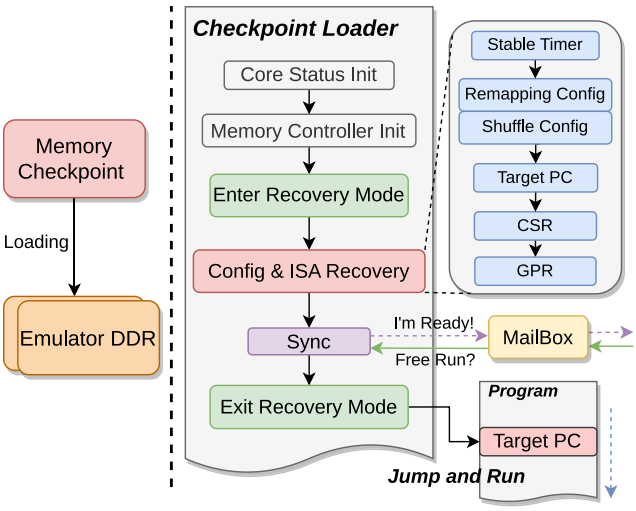



**Fig. 5.** Procedures of single-task checkpoint recovery by the checkpoint loader.

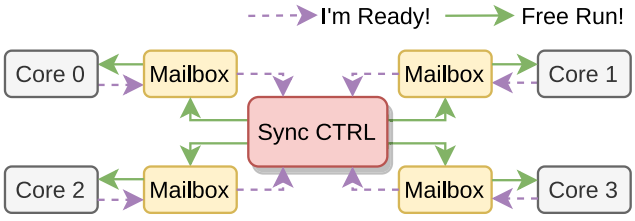



**Fig. 6.** Sync mechanism for simultaneous multiple single-task checkpoint recovery.

the complete architecture-level status required by the program's normal execution, including the memory contents and the values of various registers specified by the Instruction Set Architecture, such as general-purpose registers and control status registers. Specifically, for the recovery of memory content, MultiPoint utilizes the programming interface provided by the Emulator Accelerator to load the full contents of the main memory into the corresponding positions in the DDR system of the Emulator Accelerator.

For the recovery of register, MultiPoint introduces a checkpoint loader, which is firmware that is responsible for initializing the DDR system and preparing the execution environment before handing over control to the operating system in the checkpoint to be recovered. Specifically, in the checkpoint loader, after completing the system initialization, the processor core enters the recovery mode, in which all register values can be modified via instructions, regardless of their writable properties defined in the Instruction Set Architecture (ISA). Subsequently, the checkpoint loader begins executing instructions related to status configuration and register recovery.

The stable timer register, which supplies the wall clock time for Linux, is restored. Inaccurate restoration of this register can lead Linux to perceive the current time as earlier or significantly later than the actual last recorded time during subsequent system checks. This discrepancy can induce kernel panic and disrupt the normal operation of programs. Following this, remapping and shuffling settings are configured for the subsequent run-time physical transform mechanism, as detailed in Section 4.2. Next, the program counter (PC) for the first instruction in the checkpoint execution is logged. ISA registers, including control status registers (CSRs) and general purpose registers (GPRs) [25], are then restored by respective instructions. It is noted that since the recovery process is instruction-based, which requires modification of GPR values, GPRs must be restored at the last step. As the speed of checkpoint recovery may vary across different cores, each processor core, upon completing its checkpoint recovery, would sync with other cores, ensuring all cores have finished their recovery. Once synchronization is achieved, a special instruction is executed, enabling all processor cores to exit recovery mode and simultaneously jump to the target PC and initialize their execution. It is required that this special instruction does not modify the values of any registers visible to the ISA. Collectively, when the processor cores begin their execution from the first instruction of the checkpoints, the values in memory, various registers, and the necessary operating system state have all been carefully restored, allowing the program to commence the normal execution.

During the synchronization procedure, processor cores communicate through their mailboxes, which is a hardware mechanism of the asynchronous inter-core communication. Specifically, as shown in Fig. 6, once a processor core completes its checkpoint recovery, it registers its readiness in its own mailbox and waits for a free run signal to trigger its execution. The Sync controller would monitor all cores' mailboxes. Upon detecting all the cores have been prepared, the Sync controller would send the free run message to all cores' mailboxes, thus simultaneously initiating executions of the multiple multi-task workloads.

### 4.2. Address remapping and shuffling

MultiPoint proposes a software-transparent physical address transform mechanism to support the concurrent and smooth execution of multiple single-task checkpoints. As illustrated in Fig. 7, after the virtual address translation through the Translation Lookaside Buffer (TLB), the cached memory read or write requests issued by the processor core are remapped and then shuffled. For the uncached memory access requests, which typically originate from Linux kernel interactions with I/O devices, their physical addresses are not translated to ensure accurate access to peripherals. Typically, the only peripheral in the pre-silicon performance evaluation is the serial port, which is used for program output printing. It is noted that above address transforms in MultiPoint are hardware-only and software-transparent, thus requiring no special software modifications and imposing no additional requirements for checkpoint capture.

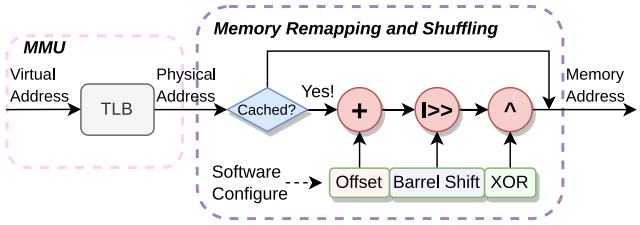


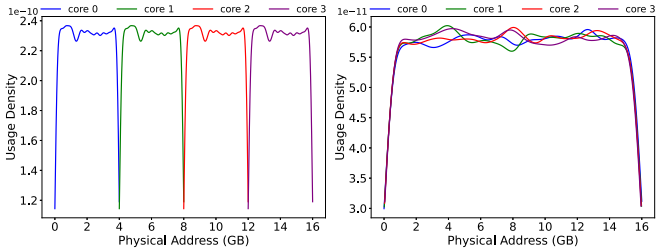

**Fig. 7.** Mechanism of physical address transforms in MultiPoint. The detailed procedures are presented in Algorithm 1.

**Algorithm 1** Procedures of address transforms in MultiPoint

```
1: function LBS(addr, from_bit, to_bit, lbs_bits)
2:     origin ← addr [from_bit : to_bit)
3:     shuffled ← LeftBarrelShift(origin, lbs_bits)
4:     addr [from_bit : to_bit) ← shuffled
5:     return addr
6: end function
7: function XOR(addr, from_bit, to_bit)
8:     first_slice ← addr [start : start + to_bit - from_bit)
9:     addr [from_bit, to_bit) ⊕ = first_slice
10:    return addr
11: end function
12: function RemappingAndShuffling(paddr, core_id)
13:    # Address Remapping
14:    paddr ← paddr + core_id × (1 ≪ task_mem_bit)
15:    # Address Left Barrel Shift
16:    slices ← Max(⌊(task_mem_bit - start) / copies⌋, 1)
17:    for slice from 0 to slices do
18:        from_bit ← start + slice × copies
19:        to_bit ← Min(from_bit+copies, task_mem_bit)
20:        lbs_bits ← core_id + slice
21:        paddr ← LBS(paddr, from_bit, to_bit, lbs_bits)
22:    end for
23:    # Address Xor
24:    paddr ← XOR(paddr, to_bit, total_mem_bit)
25:    return paddr
26: end function
```

The detailed procedures of address remapping and shuffling in MultiPoint are illustrated in Algorithm 1. Specifically, for address remapping, the physical addresses *paddr* of memory requests issued by different cores are added offsets that are multiplied by *core_id* and the memory size for each task (**line 14**), ensuring that physical addresses across different cores do not overlap with each other. For the address shuffling, the remapped addresses *paddr* starting from the `start` bit are divided into several slices, each containing bit numbers of *copies*. In this algorithm, contiguous physical addresses within the $2^{\texttt{start}}$-aligned ranges would retain their continuity after the address shuffling. The *slice* number is calculated by dividing the shuffled address bits by *copies* (**line 16**). For single-task checkpoint with 4 GB memory (*task_mem_bit* being 32), the shuffled address bits are correspondingly $32 - \texttt{start}$. Each slice in *paddr* is conducted the left barrel shift (**line 21**), with the number of shifted bits determined by the sum of the *core_id* and the *slice* index (**line 20**). The remaining *paddr* bits starting from the boundary of left barrel shift to the end of effective physical address bits are XORed (**line 24**) with the corresponding bits stating from the `start` bit of *paddr*. Collectively, the physical addresses are remapped and shuffled in such a software-transparent and hardware-friendly manner.

Fig. 8 illustrates the impact of physical address remapping and shuffling when restoring four identical single-task checkpoints. Specifically, Fig. 8 depicts the distributions of physical addresses allocated to different cores. The physical addresses are randomly generated addresses




(a) After address remapping (b) After address shuffling

**Fig. 8.** Illustration of address distributions after remapping and shuffling for randomly generated addresses within [0, 4 GB].

within the range [0, 4 GB]. After address remapping, different cores operate duplicated data within distinct memory regions, as shown in Fig. 8(a). Subsequent address shuffling results in the physical addresses being scattered and interleaved across the memory among different cores, as presented in Fig. 8(b).

It is important to note that the address shuffling algorithm used in MultiPoint is empirically determined, because barrel shift and xor are hardware-friendly operations. Alternate algorithms can also be utilized, provided they ensure that the physical addresses from different cores are effectively interleaved and the guarantee the transforms are one-to-one mappings.

## 5. Methodology

### 5.1. Benchmarks

In this work, the SPEC CPU 2017 Rate [9] is employed as the benchmark and is compiled using GCC-13 at the Loongnix system with Linux 4.19. The simulation points of SPEC CPU 2017 are selected by SimPoint, utilizing parameters from previous studies [7,22] with the program interval length `N` being 100 million, the maximum allowed cluster numbers `maxK` being 30, the dimension of random linear projection `dim` being 15, and the Bayesian information criterion threshold `BIC` being 0.95. The checkpoints for these simulation points are captured by the modified system-level QEMU [26,27] with the equipped memory of 4 GB. When dumping the full-system checkpoint, physical pages containing all zero content would be suppressed. Besides, the ISA utilized in this work is the LoongArch [25].

It is noted that MultiPoint is not limited to evaluations of homogeneous workloads. Heterogeneous workloads can also be evaluated by restoring distinct single-task checkpoints at different cores to compose the required multi-task checkpoints determined by the sampling methods. We want to emphasize that MultiPoint is proposed to provide a more scalable alternative to the direct multi-task checkpointing. It is orthogonal to studies on how to sample and select the simulation points for multi-task workloads.

### 5.2. Metric and evaluation platform

We implemented the evaluation routine of MultiPoint on the Emulator Accelerator and assessed its performance evaluation accuracy of multi-task workloads against its post-silicon Loongson 3A6000 four-core processor [20], whose $\mu$Arch specifications are presented in Table 1. The Emulator Accelerator is a commercial hardware platform designed to accelerate and verify complex chip designs through accurate simulation and real-time debugging. The absolute score error is utilized for evaluating performance estimation accuracy, with the definition given below:

$$Error = \frac{|Score_{MultiPoint} - Score_{3A6000}|}{Score_{3A6000}} \quad (1)$$

**Table 1**
Specifications of Loongson 3A6000 processor [20].

| Components | Features |
|---|---|
| Core | 4 LA664 Cores, with SMT2 |
| Issue width | 6 Insts per Cycle |
| Function unit | 4 Fix, 4 Vec, 4 Mem |
| Reorder buffer | 256 Entries |
| L1 Cache | 64 kB DCache and ICache |
| L2 Cache | 256 kB |
| Last level cache | 16 MB |
| Main memory | 2 Channel, DDR4-3200 |

**Table 2**
Memory bandwidth and latency of Emu.(Emulator Accelerator) and its post-silicon 3A6000 processor.

| Benchmark | Emu. | 3A6000 | Error |
|---|---|---|---|
| Stream copy (MB/s) | 36 397 | 35 977 | 1.17% |
| Stream scale (MB/s) | 26 247 | 26 402 | 0.59% |
| Stream add (MB/s) | 28 477 | 28 163 | 1.11% |
| stream triad (MB/s) | 29 611 | 29 674 | 0.21% |
| Memory Latency (ns) | 91.2 | 91.8 | 0.65% |

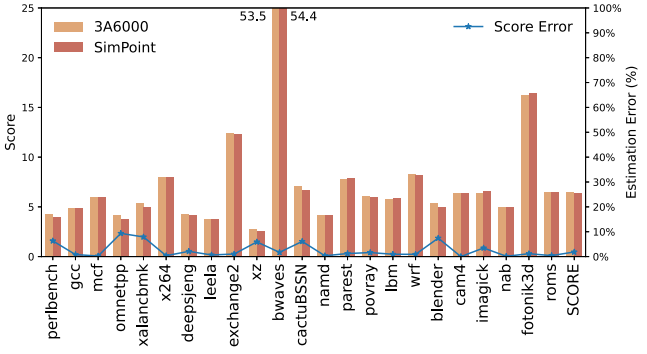



**Fig. 9.** Scores and estimation errors of SimPoint for each program in SPEC CPU 2017 Rate 1, where the SCORE is the geometric mean of individual program scores.

In SPEC CPU 2017 Rate, the score is calculated by multiplying the Rate numbers and ratios of the evaluated processor's runtime for each benchmark against a reference processor's runtime, which reflects system's throughput and scalability. The final SCORE is calculated by computing the geometric mean of the individual program scores. Due to potential variations in execution speeds among different cores, the runtime reported under Rate $N$ mode is determined by the time taken by the slowest-running core.

The Emulator Accelerator is equipped with 32 GB memory, i.e., addr_bit in Algorithm 1 being 35, which enables the concurrent execution of up to eight checkpoint copies. Besides, the parameter `start` is assigned the value of 24, which indicates that contiguous physical addresses within the 16 MB-aligned space would retain their continuity after the address remapping and shuffling. Different parameter values of `start` are discussed in Section 6.3.

To ensure the reliability of the experimental results, we calibrated the Emulator Accelerator against its post-silicon 3A6000 processor. Given that the Emulator Accelerator and the 3A6000 processor share identical core logic, our calibration efforts were concentrated on the memory system. Specifically, we aligned the parameters of the memory controllers between these two systems. The calibration outcomes for their memory system are detailed in Table 2. We evaluated the alignment of their memory systems using the `stream` and `lat_mem_rd` benchmarks to assess memory bandwidth and latency, respectively. The results, as shown in Table 2, demonstrate that the memory systems of the Emulator Accelerator and its post-silicon 3A6000 processor are fundamentally aligned. Furthermore, the estimation accuracy of SimPoint on SPEC CPU 2017 Rate 1 is validated on the calibrated Emulator Accelerator platform. The corresponding scores and estimation errors are presented in Fig. 9, where SimPoint yields the total score error of only 1.85%, which is consistent with previous studies [8,28].

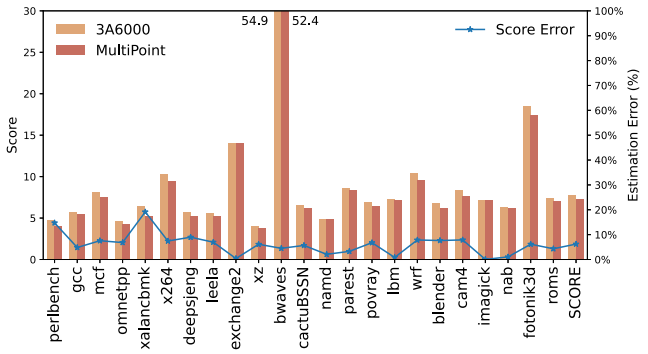



**Fig. 10.** Scores and errors of SPEC CPU 2017 Rate 2.

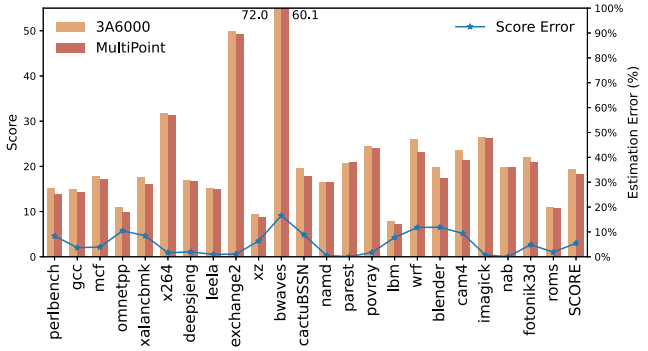



**Fig. 11.** Scores and errors of SPEC CPU 2017 Rate 4.

## 6. Evaluation

### 6.1. Analysis of performance evaluation accuracy

We employed the SPEC CPU 2017 Rate 2, Rate 4, and Rate 8 benchmarks to evaluate MultiPoint and assessed its score estimation accuracy against the post-silicon 3A6000 commercial processors. For Rate 2, these workloads are executed on two logic threads of one processor core. For Rate 4, these workloads are executed on four logic threads of four processor cores. For Rate 8, these workloads are executed on eight logic threads of four processor cores. The scores and estimation errors of these multi-task workloads are presented in Figs. 10, 11, and 12, respectively. In these figures, SCORE is the final SPEC CPU score. Specifically, for these multi-task workloads, MultiPoint achieved score errors of 6.20%, 5.45%, and 6.99%, respectively.

It is observed that for most programs in these multi-task workloads, MultiPoint could achieve estimation errors within 10%; while for several programs, such as *xalancbmk*, *wrf*, and *cam4* in Rate 8, MultiPoint yields relatively large estimation errors, with the value of 22.79%, 25.02%, and 15.11%, respectively, Nevertheless, studies [7,29] have illustrated that SimPoint-based method could exhibit stable relative errors across different $\mu$Arch designs. It is important to note that, in pre-silicon $\mu$Arch performance evaluations, the consistency of estimation errors across different $\mu$Arch designs holds greater significance than the magnitude of the error itself [15].

***Evaluations across different $\mu$Arch designs.*** To evaluate the cross-$\mu$Arch consistency of MultiPoint, programs *xalancbmk*, *wrf*, and *cam4* are evaluated at five distinct $\mu$Arch designs and their estimation errors are demonstrated in Fig. 13. These programs are presented because their estimation errors are the highest in the Rate 8 evaluations. Besides

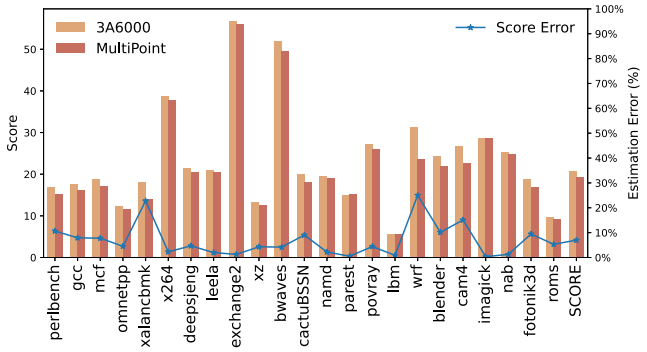



**Fig. 12.** Scores and errors of SPEC CPU 2017 Rate 8.

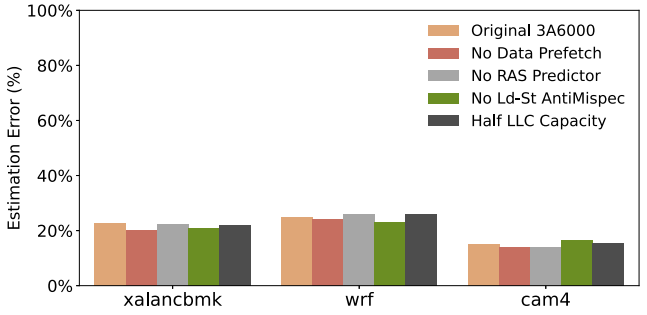



**Fig. 13.** Score estimation errors across five different $\mu$Arch designs of MultiPoint for programs *xalancbmk*, *wrf*, and *cam4* in SPEC CPU 2017 Rate 8, whose estimation errors are the highest as shown in Fig. 12.

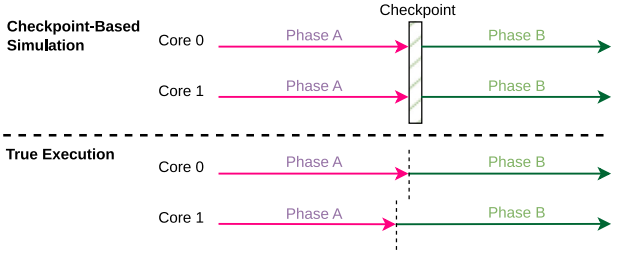



**Fig. 14.** Illustration of the discrepancy between checkpoint-based simulation and true execution for multi-task workloads.

of the original 3A6000, the introduced $\mu$Arch designs involve various structure changes of **(a)** turning off data prefetching, **(b)** turning off branch predictor for return branch, **(c)** turning off load-store memory dependence prediction, and **(d)** reducing half of the LLC capacity, respectively. The post-silicon 3A6000 processor is configured these $\mu$Arch changes through firmware modifications. As shown in Fig. 13, MultiPoint yields relatively stable estimation errors for these three programs across these five different $\mu$Arch designs, with the standard variance being 1.04%, 1.13%, and 0.97%, respectively. Besides, the estimation errors of these programs exhibit same-sign bias across different $\mu$Arch changes. To sum up, these consistent biases in estimation errors could enable designers to make correct trade-off decisions in the design space explorations of multi-core processors.

***Error Comparisons with single-task workload.*** It is observed that for many programs, estimation errors in multi-task workloads of Rate 2, 4, and 8 are higher than those in single-task workloads of Rate 1. For example, the estimation errors for *wrf* in Rate 1, 2, 4, and 8 workloads are 0.91%, 7.85%, 11.80%, and 25.02%, respectively. This discrepancy arises from the inherent limitations of checkpoint-based methods, as discussed in previous studies [30–33]. Fig. 14 illustrates

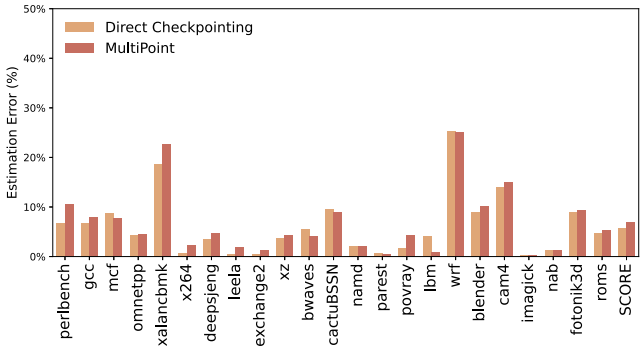



**Fig. 15.** Comparison of score errors of SPEC CPU 2017 Rate 8 for MultiPoint and direct multi-task checkpointing.

**Table 3**
Summary on estimation errors of MultiPoint and direct multi-task checkpointing for SPEC CPU 2017 Rate 2/4/8.

| Error | MultiPoint | Direct checkpointing |
|---|---|---|
| Rate 2 | 6.20% | 5.23% |
| Rate 4 | 5.45% | 4.98% |
| Rate 8 | 6.99% | 5.79% |
| **Average** | **6.21%** | **5.33%** |

the discrepancies between checkpoint-based simulation and true execution for multi-task workloads. In realistic executions of multi-task workloads, variability in the execution speeds of individual tasks can occur due to differences in memory response order and memory access latency across cores. Therefore, to make inter-task relative progress independent of specific $\mu$Arch implementations, current methods [34, 35] tried to enforce identical execution speeds across all cores. This approach introduces divergences between the actual runtime state and the initial checkpoint state. Such discrepancies lead to variations in memory access timing and patterns, resulting in different performance behaviors on $\mu$Arch structures, such as the last-level cache and memory controller, during subsequent executions. Consequently, there can be significant performance differences between checkpoint-based simulations and true executions. In contrast, for single-task workloads, where inter-workload relative execution speed is not a concern, the initial state of the checkpoint aligns consistently with the realistic runtime state. As a result, checkpoint-based evaluation methods exhibit lower estimation errors for single-task workloads compared to multi-task workloads. Nevertheless, MultiPoint is still effective in pre-silicon performance evaluations for multi-task workloads, considering that it could yield stable estimation errors across different $\mu$Arch designs, as shown in Fig. 13.

### 6.2. Comparisons with direct checkpointing

In this subsection, MultiPoint is evaluated against the direct multi-task checkpointing, as introduced in Section 2.2, in terms of their estimation accuracy and overheads.

***Estimation Accuracy.*** The comparisons of score errors of each program in SPEC CPU 2017 Rate 8 for MultiPoint and direct multi-task checkpointing are depicted in Fig. 15, where MultiPoint achieves comparable estimation errors compared to the direct multi-task checkpointing. Specifically, the total score estimation errors of direct checkpointing and MultiPoint are 6.99% and 5.79%, respectively. The comparisons of estimation errors for Rate 2 and Rate 8 exhibit similar tendencies are hence not presented. The summary on estimation errors of MultiPoint and direct multi-task checkpointing for SPEC CPU 2017 Rate is given in Table 3, where their average total score errors are 6.21% and 5.33%, respectively. As demonstrated in Table 3, MultiPoint yields same-level estimation accuracy compared to the direct multi-task

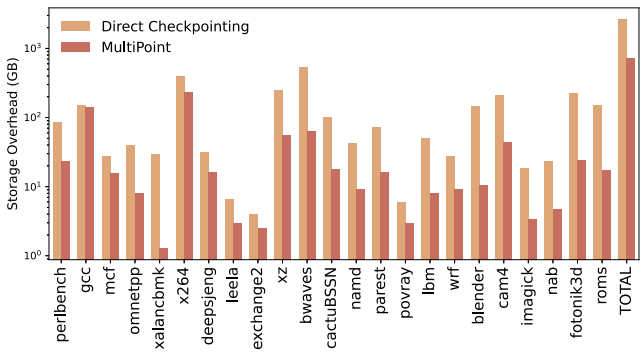



**Fig. 16.** Storage overheads of SPEC CPU 2017 Rate 8 for MultiPoint and direct multi-task checkpointing, where the *y*-axis is logarithmically scaled for better visualization.

**Table 4**
Summary on storage overheads of MultiPoint and direct multi-task checkpointing for SPEC CPU 2017 Rate 2/4/8.

| Storage (TB) | MultiPoint | Direct checkpointing |
|---|---|---|
| Rate 2 | 0.71 | 0.98 |
| Rate 4 | 0.71 | 1.51 |
| Rate 8 | 0.71 | 2.58 |
| **Total** | **0.71** | **5.07** |

checkpointing. The estimation error discrepancies between MultiPoint and direct multi-task checkpointing mainly originate from two issues, as discussed below.

Firstly, while MultiPoint incorporates an address shuffling mechanism, its memory address usage is inevitably not identical that of realistic multi-task workloads. The differences in memory access patterns can result in varying performance behaviors on physical-address-aware $\mu$Arch structures, such as caches, hardware data prefetchers, and memory controllers. Consequently, compared to direct multi-task checkpointing, MultiPoint demonstrates larger but still comparable estimation errors for most programs, as illustrated in Fig. 15. Besides, it is noted that MultiPoint's estimation errors could be improved if more sophisticated address shuffling algorithm is introduced.

Secondly, it is noted that the execution paths of profiling the code signatures and capturing the checkpoints of multi-task workloads are inevitably different due to non-deterministic events, such as random variations in task scheduling and memory accesses [30,32]. Therefore, the captured checkpoints of multi-task workloads do not strictly correspond to their simulation points. In contrast, single-task workloads exhibit more deterministic and repeatable execution paths. MultiPoint constructs the required multi-task checkpoints through multiple single-task checkpoints, which mitigates the misalignment between multi-task simulation points and their checkpoints. Besides, due to the statistical nature [29] of K-Means clustering in SimPoint, the simulation points selected by direct multi-task checkpoint is not identical to that constructed by MultiPoint, thereby introducing random discrepancies in their performance estimations. Collectively, compared to the direct checkpointing, MultiPoint can occasionally produce lower estimation errors for certain programs, such as the *lbm*, *bwaves*, and *mcf*.

***Storage Overheads.*** The comparison of storage overheads of each program in SPEC CPU 2017 Rate 8 for MultiPoint and direct multi-task checkpointing are presented in Fig. 16, where the *y*-axis is logarithmically scaled for better visualization. Specifically, their total storage requirements are 0.71 TB and 2.58 TB, respectively. It is noted that MultiPoint significantly reduces the storage overheads by 72.5%. This is attributed to that MultiPoint composes the multi-task checkpoint through the combination of multiple single-task checkpoints, therefore its storage overheads of SPEC CPU 2017 Rate 8 is the same as Rate 1, which is much smaller. In contrast, direct multi-task checkpointing requires dumping the real run-time multi-task checkpoint, thus necessitating much more storage overheads.

Moreover, when evaluating different Rate $N$ workloads, direct checkpointing requires distinct checkpoints for different rate number, resulting in redundant storage overheads. In contrast, for MultiPoint, when evaluating different Rate $N$ workloads, there is no necessity to recapture respective checkpoints, thus avoiding the extra computing resources, timing, and storage overheads. For SPEC CPU Rate benchmark, the storage overhead of MultiPoint is always that of the Rate 1 workload, irrespective of how many different Rate numbers to be evaluated. The summary on storage overheads of MultiPoint and direct multi-task checkpointing for SPEC CPU 2017 Rate is presented in Table 4, where their total storage requirements are 0.71 TB and 5.07 TB, respectively. For these workloads, MultiPoint substantially decreases the storage requirements by 86.0%, while their estimation accuracy is comparable.

**Table 5**
Summary on time overheads of checkpoint capture by MultiPoint and direct multi-task checkpointing for workloads of SPEC CPU 2017 Rate 2, 4, and 8.

| Time (h) | MultiPoint | Direct checkpointing |
|---|---|---|
| Rate 2 | 9.0 | 19.4 |
| Rate 4 | 9.0 | 37.5 |
| Rate 8 | 9.0 | 85.9 |
| **Total** | **9.0** | **142.8** |

***Time Overheads.*** Table 5 summarizes the time overheads for checkpoint capture using MultiPoint and direct multi-task checkpointing for workloads of SPEC CPU 2017 Rate 2, 4, and 8. It is noted that checkpoints are captured concurrently, with the total checkpoint capture time determined by the runtime of the longest-running program. Specifically, the total time overheads are 9.0 h and 142.8 h for MultiPoint and direct multi-task checkpointing, respectively. For these workloads, MultiPoint reduces the time overheads by 93.7% while maintaining comparable performance evaluation accuracy. This significant reduction is attributed to the property that MultiPoint's checkpoint capture time is independent of the Rate numbers being evaluated. By constructing the required multi-task checkpoints from multiple single-task checkpoints, MultiPoint eliminates the need for additional checkpoint capture time for workloads of Rate 2, 4, and 8, once the Rate 1 checkpoints have been captured. In contrast, for direct multi-task checkpointing, the capture time for Rate N increases linearly with the Rate number under the QEMU `icount` mode [26].

To sum up, MultiPoint achieves larger yet still comparable estimation accuracy compared to the direct multi-task checkpointing. However, MultiPoint achieves agile evaluations with substantially 86.0% fewer storage and 93.7% less timing overheads. Compared to direct multi-task checkpointing, MultiPoint enables more scalable performance evaluations for multi-task workloads.

### 6.3. Discussions of address shuffling

In this subsection, we analyzed the benefits and parameter sensitivity of introduced address shuffling on the accuracy of performance evaluation for multi-task workloads.

***Benefits of address shuffling.*** Fig. 17 illustrates the estimation errors of MultiPoint with and without the address shuffling on the SPEC CPU 2017 Rate 8. The comparisons of estimation errors for Rate 2 and Rate 8 exhibit similar tendencies and are hence not presented. For MultiPoint without implementing address shuffling, only the address remapping is conducted to guarantee the normal concurrent execution of evaluated multi-task workloads. Specifically, as depicted in Fig. 17, incorporating address shuffling could markedly diminish the evaluation errors, significantly reducing total score errors from 43.60% to 6.99%. The error reductions are especially pronounced for programs like *bwaves*, *povray*, and *parest*, whose errors are significantly decreased by 76.33%, 73.30%, and 67.41%, respectively. It

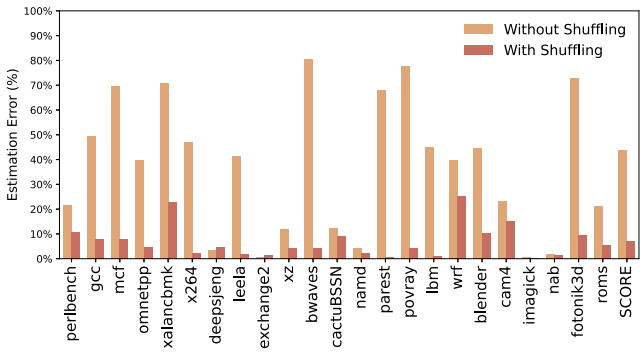



**Fig. 17.** Comparison of score estimation errors of SPEC CPU 2017 Rate 8 for MultiPoint ***with*** and ***without*** address shuffling.

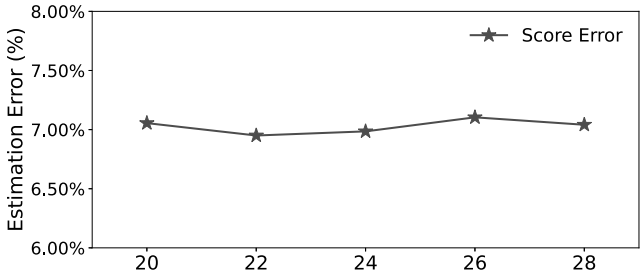



**Fig. 18.** Score estimation errors of SPEC CPU 2017 Rate 8 under different values of parameter `start` in Algorithm 1, which determines the granularity of address shuffling.

is noted that these programs own the common feature of extensive memory access demands. In contrast, for the program *deepsjeng* and *exchange2*, which exhibit moderate memory access demands, their estimation errors remained unaffected by the address shuffling.

The substantial decreases in estimation errors after introducing address shuffling are because that different physical memory address characteristics can impact the performance of certain $\mu$Arch structures that are sensitive to the physical addresses. For instance, under homogeneous workloads, like SPEC CPU 2017 Rate, if address shuffling is not implemented, the low bits of address accessed by different single-task checkpoints for semantically identical memory are identical, with only their highest bits differing. This would result in significant cache set conflicts in the shared last-level cache. In contrast, in the actual multi-task workload executions, the physical addresses of memory requests issued by different cores are interleaved with each other. This allows the memory requests to be more evenly distributed across different cache sets, thereby resulting in fewer set conflicts. Fewer cache set conflicts would bring fewer cache misses and mitigate the performance degradation associated with the serial operations required by the cache coherence protocol for memory requests within the same cache set.

***Sensitivities of Shuffling Algorithm.*** Fig. 18 presents the final score estimation errors of SPEC CPU 2017 Rate 8 under different values of the parameter `start` in Algorithm 1. The parameter `start` determines the granularity of address shuffling, where contiguous physical addresses within $2^{\texttt{start}}$-aligned ranges maintain their continuity after shuffling. As shown in Fig. 18, the final score estimation errors exhibit low sensitivity to the specific value of `start`. In this work, the default value of `start` is set to 24 because its barrel shift bits, calculated as $32 - 24 = 8$, are divisible by 2, 4, and 8, which are the evaluated Rate numbers. This selection simplifies boundary condition handling in RTL coding, considering that different `start` values yield comparable results.

Collectively, Figs. 17 and 18 demonstrate that the address shuffling mechanism significantly influences the accuracy of performance evaluation, while the specific values of `start` have only a minor impact on estimation accuracy. In essence, the coarse-grained isolation of memory usage illustrated in Fig. 8(a) fundamentally differs from the interleaved characteristics of real multi-task workloads depicted in Fig. 3. However, when the physical addresses of different tasks become interleaved, the granularity of interleaving has only a limited effect on the final performance outcomes. In this study, the address shuffling algorithm in MultiPoint is empirically selected, as barrel shift and XOR operations are hardware-friendly. Despite this, current method is effective in producing comparable evaluation accuracy to direct multi-task checkpointing. Exploring improved shuffling algorithms is a promising direction for further reducing the estimation error gap between MultiPoint and direct multi-task checkpointing.

## 7. Related work

***Evaluation for Single-Task Workloads***: There are mainly two categories of sampling methods that are widely used for single-task workload, or the single-threaded programs. **Representative Sampling** [1–8]: This method leverages the recurrent phases exhibited in the program's dynamic execution and selectively chooses several intervals to represent and reconstruct the complete execution behavior of the program. **Systematic Sampling** [36–38]: This method involves systematically extracting many short program intervals to collectively represent the behavior of the entire program. This approach could statistically ensure a broad coverage of the program's behavior, making it possible to estimate the overall performance more accurately.

***Evaluation for Multi-Task Workloads***: The multi-task workloads are primarily categorized into two types: homogeneous workloads, where different cores execute the same program, as exemplified by SPEC CPU 2017 Rate; and heterogeneous workloads, where different cores execute distinct workloads. For **homogeneous workloads**, Perelman et al. [15] introduced the parallel SimPoint method for efficient performance evaluation. For **heterogeneous workloads**, Jacobvitz et al. [10] provided a rigorous definition for benchmark evaluation purposes. Velásquez et al. [11] proposed a method involving random combinations of task loads to construct representative heterogeneous workloads. Eyerman et al. [12,39] developed several system-level performance evaluation metrics for multi-task workloads. Van et al. [19], NamKung et al. [14], Tawk et al. [17] introduced a series of sampling methods for heterogeneous workloads aimed at reducing the number of phase combinations needed for simulation, thereby enhancing simulation efficiency. Prieto et al. [40,41] proposed extracting core loops from programs and established a statistical model to validate the consistency.

***Evaluation for Multi-Threaded Workloads***: Because of the synchronization and communication among threads, the dynamic instruction counts of multi-threaded workloads can fluctuate significantly with each execution [42], and the execution paths are unpredictable [43]. Besides, the relative execution relationship among threads is microarchitecture-dependent [30,31]. These factors introduce many challenges to the performance evaluation of the multi-threaded workloads. Currently, pre-silicon performance evaluation methods for multi-threaded workloads are divided into three categories: **Timing-based** Sampling [30,31] methods periodically switch processor cores between fast-forward and detailed simulation states until the program execution concludes. **Communication primitive-based** Sampling [32,44,45] involves segmenting the program based on statements like barriers, loops, and tasks, thereby obtaining simulation points that are naturally independent of the program's execution path. **Work-based** Sampling [33–35] divides the program according to the total effective works so as to acquire simulation points that are independent of the dynamic instruction counts of the program.

## 8. Conclusion

In this work, we propose MultiPoint to enable pre-silicon performance evaluation for multi-task workloads. The key idea of MultiPoint is to construct the required multi-task workloads by combining multiple single-task workloads. To guarantee the smooth concurrent execution of these single-task checkpoints, MultiPoint introduces the mechanisms of address remapping. Furthermore, to maintain the accuracy of performance evaluation, MultiPoint shuffles memory requests from different cores to make them scatter and interleave in the memory space. MultiPoint is evaluated on the SPEC CPU 2017 and yields score estimation errors of 6.20%, 5.45%, and 6.99% for Rate 2, Rate 4, and Rate 8, respectively, achieving comparable performance evaluation accuracy compared to direct multi-task checkpointing but in a more scalable manner with substantially 86.0% lower storage and 93.7% less time overheads.

## CRediT authorship contribution statement

**Chenji Han:** Writing – original draft, Methodology, Investigation. **Xinyu Li:** Writing – review & editing, Methodology. **Feng Xue:** Writing – original draft, Methodology. **Weitong Wang:** Writing – review & editing, Methodology. **Yuxuan Wu:** Writing – review & editing, Methodology. **Wenxiang Wang:** Writing – review & editing, Methodology. **Fuxin Zhang:** Writing – review & editing, Methodology, Funding acquisition.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References


[1] Timothy Sherwood, Erez Perelman, Greg Hamerly, Brad Calder, Automatically characterizing large scale program behavior, in: ASPLOS X, 2002.

[2] Jose Renau, Fangping Liu, Hongzhang Shan, Sang Wook Stephen Do, Enabling reduced simpoint size through LiveCache and detail warmup, BenchCouncil Trans. Benchmarks Stand. Eval. 2 (4) (2022) 100082.

[3] Harish Patil, Alexander Isaev, Wim Heirman, Alen Sabu, Ali Hajiabadi, Trevor E Carlson, ELFies: executable region checkpoints for performance analysis and simulation, in: 2021 IEEE/ACM International Symposium on Code Generation and Optimization, CGO, IEEE, 2021, pp. 126–136.

[4] Odysseas Chatzopoulos, Maria Trakosa, George Papadimitriou, Wing Shek Wong, Dimitris Gizopoulos, SimPoint-based microarchitectural hotspot & energy-efficiency analysis of RISC-v OoO CPUs, in: 2024 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, IEEE, 2024, pp. 1–12.

[5] Charles Yount, Harish Patil, Mohammad S Islam, Aditya Srikanth, Graph-matching-based simulation-region selection for multiple binaries, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, IEEE, 2015, pp. 52–61.

[6] Kenneth Hoste, Lieven Eeckhout, Microarchitecture-independent workload characterization, IEEE Micro 27 (3) (2007) 63–72.

[7] Weihua Zhang, Jiaxin Li, Yi Li, Haibo Chen, Multilevel phase analysis, ACM Trans. Embed. Comput. Syst. 14 (2015) 31:1–31:29.

[8] Hongwei Cui, Yujie Cui, Honglan Zhan, Shuhao Liang, Xianhua Liu, Chun Yang, Xu Cheng, MBAPIS: Multi-level behavior analysis guided program interval selection for microarchitecture studies, in: 2023 32nd International Conference on Parallel Architectures and Compilation Techniques, PACT, IEEE, 2023, pp. 297–308.

[9] James Bucek, Klaus-Dieter Lange, Jóakim v. Kistowski, SPEC CPU2017: Next-generation compute benchmark, in: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 41–42.

[10] Adam N. Jacobvitz, Andrew D. Hilton, Daniel J. Sorin, Multi-program benchmark definition, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, IEEE, 2015, pp. 72–82.

[11] Ricardo A. Velásquez, Pierre Michaud, André Seznec, Selecting benchmark combinations for the evaluation of multicore throughput, in: 2013 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2013, pp. 173–182.

[12] Stijn Eyerman, Pierre Michaud, Wouter Rogiest, Multiprogram throughput metrics: A systematic approach, ACM Trans. Archit. Code Optim. ( TACO) 11 (3) (2014) 1–26.

[13] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, Josep Torrellas, $\mu$Manycore: A cloud-native CPU for tail at scale, in: Proceedings of the 50th Annual International Symposium on Computer Architecture, 2023, pp. 1–15.

[14] Jeffrey Namkung, Dohyung Kim, Rajesh Gupta, Igor Kozintsev, Jean-Yves Bouget, Carole Dulong, Phase guided sampling for efficient parallel application simulation, in: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, 2006, pp. 187–192.

[15] Erez Perelman, Marzia Polito, J-Y Bouguet, Jack Sampson, Brad Calder, Carole Dulong, Detecting phases in parallel applications on shared memory architectures, in: Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, IEEE, 2006, pp. 10–pp.

[16] Alvaro Wong, Dolores Rexachs, Emilio Luque, Extraction of parallel application signatures for performance prediction, in: 2010 IEEE 12th International Conference on High Performance Computing and Communications, HPCC, IEEE, 2010, pp. 223–230.

[17] Melhem Tawk, Khaled Z. Ibrahim, Smail Niar, Multi-granularity sampling for simulating concurrent heterogeneous applications, in: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2008, pp. 217–226.

[18] Melhem Tawk, Khaled Z. Ibrahim, Smail Niar, Adaptive sampling for efficient mpsoc architecture simulation, in: 2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, IEEE, 2007, pp. 186–192.

[19] M. Van Biesbrouckt, Lieven Eeckhout, Brad Calder, Considering all starting points for simultaneous multithreading simulation, in: 2006 IEEE International Symposium on Performance Analysis of Systems and Software, IEEE, 2006, pp. 143–153.

[20] Loongson, Loongson LS3a6000 processor, 2024, URL https://www.loongson.cn/EN/product/show?id=11.

[21] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, Yungang Bao, Towards Developing High Performance RISC-V Processors Using Agile Methodology, in: 2022 55th IEEE/ACM International Symposium on Microarchitecture, MICRO, 2022, pp. 1178–1199.

[22] Brian Grayson, Jeff Rupley, Gerald D. Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, P. W. Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, Ankit Ghiya, Evolution of the samsung exynos CPU microarchitecture, in: 2020 ACM/ IEEE 47th Annual International Symposium on Computer Architecture, ISCA, 2020, pp. 40–51.

[23] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R.J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E.A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, SciPy 1.0 Contributors, SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, Nature Methods 17 (2020) 261–272, http://dx.doi.org/10.1038/s41592-019-0686-2.

[24] Mel Gorman, Physical page allocation, 2013, URL https://www.kernel.org/doc/gorman/html/understand/understand009.html. (Accessed 24 December 2024).

[25] Loongson Technology, LoongArch documentation, 2022, URL https://loongson.github.io/LoongArch-Documentation/.

[26] Fabrice Bellard, QEMU, a fast and portable dynamic translator., in: USENIX Annual Technical Conference, FREENIX Track, Vol. 41, Califor-nia, USA, 2005, p. 46.

[27] Xinyu Li, Plugin for checkpoint dumping in QEMU, 2024, URL https://github.com/rrwhx/qemu_plugins_loongarch.

[28] Björn Gottschall, Silvio Campelo de Santana, Magnus Jahre, Balancing accuracy and evaluation overhead in simulation point selection, in: 2023 IEEE International Symposium on Workload Characterization, IISWC, IEEE, 2023, pp. 43–53.

[29] Erez Perelman, Greg Hamerly, Brad Calder, Picking statistically valid and early simulation points, in: 2003 12th International Conference on Parallel Architectures and Compilation Techniques, 2003, pp. 244–255.

[30] Ehsan K. Ardestani, Jose Renau, ESESC: A fast multicore simulator using time-based sampling, in: 2013 IEEE 19th International Symposium on High Performance Computer Architecture, HPCA, 2013, pp. 448–459.

[31] Trevor E. Carlson, Wim Heirman, Lieven Eeckhout, Sampled simulation of multi-threaded applications, in: 2013 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2013, pp. 2–12.

[32] Trevor E Carlson, Wim Heirman, Kenzo Van Craeynest, Lieven Eeckhout, Barrierpoint: Sampled simulation of multi-threaded applications, in: 2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, IEEE, 2014, pp. 2–12.

[33] Changxi Liu, Alen Sabu, Akanksha Chaudhari, Qingxuan Kang, Trevor E. Carlson, Pac-Sim: Simulation of multi-threaded workloads using intelligent, live sampling, ACM Trans. Arch. Code Optim. (2024).

[34] Alen Sabu, Harish Patil, Wim Heirman, Trevor E. Carlson, LoopPoint: Checkpoint-driven sampled simulation for multi-threaded applications, in: 2022 IEEE International Symposium on High-Performance Computer Architecture, HPCA, 2022, pp. 604–618.

[35] Alen Sabu, Changxi Liu, Trevor E. Carlson, Viper: Utilizing hierarchical program structure to accelerate multi-core simulation, IEEE Access (2024).

[36] Sina Hassani, Gabriel Southern, Jose Renau, LiveSim: Going live with microarchitecture simulation, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2016, pp. 606–617.

[37] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, James C. Hoe, SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling, in: 30th Annual International Symposium on Computer Architecture, 2003. Proceedings, 2003, pp. 84–95.

[38] Uday Kumar Reddy Vengalam, Anshujit Sharma, Michael C. Huang, LoopIn: A loop-based simulation sampling mechanism, in: 2022 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2022, pp. 224–226.

[39] Stijn Eyerman, Lieven Eeckhout, System-level performance metrics for multiprogram workloads, IEEE Micro 28 (3) (2008) 42–53.

[40] Pablo Prieto, Pablo Abad, Jose Angel Herrero, Jose Angel Gregorio, Valentin Puente, SPECcast: A methodology for fast performance evaluation with SPEC cpu 2017 multiprogrammed workloads, in: Proceedings of the 49th International Conference on Parallel Processing, 2020, pp. 1–11.

[41] Pablo Prieto, Pablo Abad, Jose Angel Gregorio, Valentin Puente, Fast, accurate processor evaluation through heterogeneous, sample-based benchmarking, IEEE Trans. Parallel Distrib. Syst. 32 (12) (2021) 2983–2995.

[42] Alaa R. Alameldeen, David A. Wood, IPC considered harmful for multiprocessor workloads, IEEE Micro 26 (4) (2006) 8–17.

[43] Weihua Zhang, Xiaofeng Ji, Bo Song, Shiqiang Yu, Haibo Chen, Tao Li, Pen-Chung Yew, Wenyun Zhao, Varcatcher: A framework for tackling performance variability of parallel workloads on multi-core, IEEE Trans. Parallel Distrib. Syst. 28 (4) (2016) 1215–1228.

[44] Miguel Tairum Cruz, Sascha Bischoff, Roxana Rusitoru, Shifting the barrier: extending the boundaries of the barrierpoint methodology, in: 2018 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, IEEE, 2018, pp. 120–122.

[45] Thomas Grass, Trevor E Carlson, Alejandro Rico, German Ceballos, Eduard Ayguade, Marc Casas, Miquel Moreto, Sampled simulation of task-based programs, IEEE Trans. Comput. 68 (2) (2018) 255–269.