A trace-driven methodology to evaluate and optimize memory management services of distributed operating systems for lightweight manycores

Emmanuel Podestá Junior, Pedro Henrique Penna, João Fellipe Uller, Marcio Castro

A trace-driven methodology to evaluate and optimize memory management services of distributed operating systems for lightweight manycores

Emmanuel Junior, Pedro Henrique Penna, João Uller, Marcio Castro

A Trace-Driven Methodology to Evaluate and Optimize Memory Management Services of Distributed Operating Systems for Lightweight Manycores

Emmanuel Podestá Junior
Federal University of Santa Catarina
Florianópolis, Santa Catarina, Brazil
emmanuel.podesta@posgrad.ufsc.br

Pedro Henrique Penna
Pontifical Catholic University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
pedro.penna@sga.pucminas.br

João Fellipe Uller
Federal University of Santa Catarina
Florianópolis, Santa Catarina, Brazil
joao.f.uller@grad.ufsc.br

Márcio Castro
Federal University of Santa Catarina
Florianópolis, Santa Catarina, Brazil
marcio.castro@ufsc.br

ABSTRACT

Lightweight manycores belong to a new class of emerging low-power processors for the Exascale era. These processors present several challenges for the development of applications, such as distributed memory architecture, limited amount of on-chip memory and no cache coherence. Recently, distributed Operating Systems (OSs) have been proposed to address these challenges in a transparent way. In these systems, different OS services are deployed across the processor cores, being the memory management service one of the most important ones. However, the intrinsic characteristics and memory limitations of lightweight manycores bring several challenges to the design, implementation and future optimizations of memory management services. In this work, we propose a trace-driven methodology to evaluate and optimize features of a memory management service of distributed OSs for lightweight manycores. By using a compact representation of the page access pattern of the applications, our methodology is capable of mimicking the memory access pattern of the original applications on the target distributed OS running on a lightweight manycore. We integrated our methodology in a distributed OS (Nanvix) and validated it using three applications from a specific benchmark for lightweight manycores (CAP Bench). Then, we applied our methodology to carry out a case study using a software-managed cache implementation available in Nanvix. Our methodology enabled us to evaluate different page replacement policies on Kalray MPPA-256, even without the required support from the architecture to implement them.

CCS CONCEPTS

• **Computer systems organization** → **System on a chip**; **Very long instruction word**; **Multiple instruction, multiple data**; **Parallel architectures**;

KEYWORDS

Trace-Driven Methodology, Distributed Operating System, Memory Management, Software-managed Cache, Lightweight Manycores

ACM Reference Format:
Emmanuel Podestá Junior, Pedro Henrique Penna, João Fellipe Uller, and Márcio Castro. 2021. A Trace-Driven Methodology to Evaluate and Optimize Memory Management Services of Distributed Operating Systems for Lightweight Manycores. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3412841.3441994

1 INTRODUCTION

Historically, hardware architects were able to design faster and more advanced uniprocessor systems by increasing the number of transistors per chip and scaling the clock frequency. In addition, they implemented features such as speculative execution, instruction-level parallelism, out-of-order execution and larger caches, which contributed to improve single-threaded performance. Unfortunately, these aggressive strategies led processors to hit a new constraint: the power wall [21]. As a response to this challenge, the industry transitioned toward Chip Multiprocessor (CMP) designs, which feature more than one processor core to increase the aggregate performance of the chip.

More recently, the increase in performance and parallelism in a single chip has been taken to a next level with the emergence of a new class of highly-parallel processors named lightweight manycores [3]. These processors integrate hundreds of low-power cores on a single chip, which allow to exploit both data and task parallelism, and have been proven to achieve better energy efficiency than CMPs [5]. In this design, cores are grouped into *clusters*, each one having its own limited local memory (a.k.a scratchpad memory) and address space, thus resulting in a distributed memory design. Clusters are interconnected by one or more Networks-on-Chip (NoCs) and communications between them take place through message exchanges. Examples of such processors are the Sunway SW26010 [9], which is the base processor of the Sunway TaihuLight supercomputer (featuring 10.6 million low-power cores in total, Adapteva Epiphany [16] and Kalray MPPA-256 [6].

One of the main problems that hinders the adoption of lightweight manycores by the industry is the lack of Operating System

(OS) support. Currently, software engineers have to explicitly deal with data tiling, data prefetching and low-level communication abstractions to extract reasonable performance out of a lightweight manycore processor [20]. Moreover, communications must take into account the NoC topology whenever possible to improve bandwidth and reduce latency.

Recently, distributed OSs were proposed to address these challenges in a transparent way [4, 10]. The multikernel is one of the distributed OS designs that has shown promising results. In this approach, multiple independent OS kernel instances are deployed on the lightweight manycore processor. Each kernel provides bare-minimum abstractions and fully-featured system services are implemented in a distributed fashion. Furthermore, the distributed architecture brings: (i) better scalability; (ii) hardware-neutral characteristics, enabling portability between different architectures; and (iii) explicit inter-core communication, which offers more room for optimizations and efficient use of the processor network.

Among the services offered by distributed OSs for lightweight manycores, the memory management is one of the most important ones. This service overcomes most programming intricacies of a distributed memory architecture, and it must exploit the NoC in order to achieve decent performance. However, designing and implementing a memory management subsystem that takes into account the intrinsic characteristics and memory limitations of lightweight manycores is still an open problem, paving the way for different performance optimizations. For instance, part of the local memory of clusters can be used as a software-managed cache to store recently used pages. Moreover, a software prefetching mechanism can be adopted to bring pages from the main memory to the local memory of clusters in advance to hide NoC communication costs. Towards these optimizations, a possible approach would be to port several applications of different domains to the target distributed OS and carry out experiments with them. However, actually porting applications to distributed OSs that target lightweight manycores is a time-consuming and error-prone task.

Thus, to help on these analyses and relieve developers from the burden of porting several applications to the target distributed OS, in this paper we propose a trace-driven methodology that can be used to evaluate and optimize features of a memory management service of distributed OSs for lightweight manycores. In our methodology, applications are first executed in a standard Linux environment and their memory accesses are traced. Then, the traced information is transformed into compacted structures called *heatmaps*, which represent the memory access pattern of the applications. Finally, the *heatmaps* are used by a *proxy application* in the target distributed OS running on the lightweight manycore processor to mimic the memory access pattern of the original applications. In summary, this work delivers the following new contributions to the state of the art on the evaluation of memory management services of distributed OSs for lightweight manycores:

(i) a new trace-driven methodology that helps distributed OS developers to evaluate and optimize features of a memory management services without the need of porting applications to the target distributed OS;
(ii) an integration of the proposed methodology in Nanvix, an open-source distributed OS that targets lightweight manycores [17]; and
(iii) a case study of a software-managed cache evaluation in Nanvix using the proposed methodology.

All experiments with Nanvix were executed on Kalray MPPA-256 [6], a NoC-based lightweight manycore processor that features a distributed memory architecture and integrates 288 cores in a single chip. We show that our methodology is able to correctly represent and mimic the memory access pattern of the original applications. Moreover, we show that the resolution of the *heatmap* can be fine-tuned to greatly reduce the number of memory accesses performed by the *proxy application* in the target distributed OS, while keeping the overall memory access pattern behavior of the original applications. This allows for much faster executions, enabling the evaluation of the memory management service in several scenarios in a feasible time. Finally, we show that the methodology can be applied to study the behavior of different page replacement policies of a software-managed cache implementation, even when the underlying lightweight manycore processor does not feature the necessary hardware support to implement them, helping hardware architects to decide whether or not is beneficial to include the necessary hardware support to allow the implementation of smarter page replacement policies in the software-managed cache.

The remainder of this work is organized as follows. Section 2 presents the background on lightweight manycores and distributed OSs. Section 3 describes our trace-driven methodology. Then, Section 4 discusses our evaluation methodology. Finally, Section 5 presents our results, Section 6 covers related works and Section 7 concludes this paper.

2 BACKGROUND

First, we discuss about lightweight manycores and their main characteristics (Section 2.1). Then, we present an overview of distributed OSs for these processors (Section 2.2).

2.1 Lightweight Manycore Processors

Lightweight manycore processors differ from other architectures that also have high core counts, such as Non-Uniform Memory Access (NUMA) platforms, Graphics Processing Units (GPUs) and other accelerators (e.g., Intel Xeon Phi). Their main distinctions are the following:

(i) they integrate hundreds or thousands of low-power cores in a single chip, which are tightly-coupled in groups called *clusters*;
(ii) they are designed to target Multiple Instruction Multiple Data (MIMD) workloads;
(iii) they have a distributed memory architecture;
(iv) they feature a constrained memory system with small local memories inside clusters and no cache coherency between clusters;
(v) they rely on high bandwidth and rich NoCs to carry out fast and reliable message-passing communication; and
(vi) they usually have a heterogeneous configuration, both in terms of Input/Output (I/O) and computing capabilities.

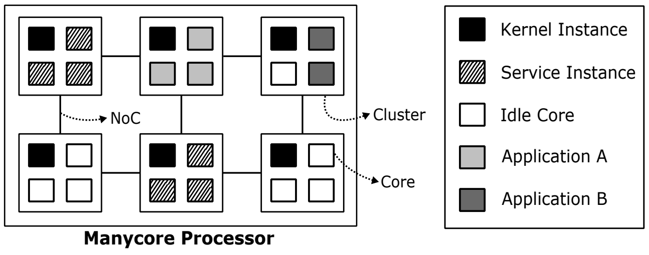


Figure 1: A simplified overview of a conceptual lightweight manycore running a multikernel OS.

Figure 1 shows an overview of a conceptual lightweight manycore. Some examples of lightweight manycores are the Sunway SW26010 [9] and the Kalray MPPA-256 [6]. The former is a custom-designed processor used in the Sunway TaihuLight, currently the world's fourth most powerful system according to TOP500[1]. The latter, on the other hand, is a commercially available lightweight manycore processor designed by Kalray.

2.2 Distributed Operating Systems

Distributed OSs recently emerged as an alternative approach to address the challenges in software development and deployment in lightweight manycore processors [1, 17]. These OSs are composed by a set of services that implement the main abstractions of a traditional OS. They are usually structured in three main layers. In the bottom layer, a *Hardware Abstraction Layer (HAL)* abstracts the underlying hardware, so as to provide portability across different architectures. In the middle layer, an *OS kernel* provides resource multiplexing as well as basic OS abstractions, such as processes and threads. Finally, the top layer features several *OS services* to provide a transparent programming environment for users.

In this paper, we are specially interested in distributed OSs designed specifically for lightweight manycores such as MOSSCA [12], M^3 [1] and Nanvix [17]. In these OSs, a microkernel design is employed in the middle layer so as to cope with the low amount of on-chip memory as shown in Figure 1. Usually, a single microkernel instance is deployed in each Compute Cluster (*kernel instance*) and all other cores of the Compute Cluster are used to run system services (*service instance*) or user-level applications. We chose Nanvix as our target distributed OS in this work because, to the best of our knowledge, it is currently the only open-source distributed OS that runs on commercially available baremetal lightweight manycores.

In Nanvix, the memory management service manages page allocation and sharing among Compute Clusters. The Distributed Paging System (DPS) is the main component of this service, which provides a shared memory abstraction and exposes a programming interface to manipulate these abstractions. In the next sections, we show how our trace-driven methodology can help distributed OS developers to evaluate and optimize features of this service without the need of porting applications to this distributed OS.

[1]Available at: https://www.top500.org

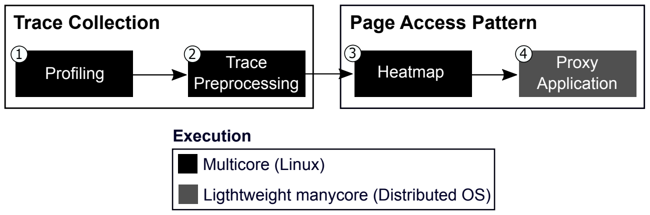


Figure 2: Overview of the trace-driven methodology.

3 TRACE-DRIVEN METHODOLOGY

Our trace-driven methodology is intended to aid developers to study, implement and optimize features of memory management services of distributed OSs. This methodology relieves developers from the burden of porting several applications to the target distributed OS, which is necessary to evaluate the memory management service under different scenarios.

Figure 2 presents an overview of this methodology. First, applications are compiled and executed in a standard Linux environment along with a memory profiling tool. The profiler collects memory accesses of the target application and dumps this information on trace files (Step ①). Then, each trace file is preprocessed (if necessary), so as to obtain a well-formatted output (Step ②).

Next, the well-formatted output is used to create a *heatmap*: a compact two-dimensional structure that describes how many times each memory page (or a set of pages) was accessed during some discrete time frames (Step ③). The resolution of the *heatmap* structure can be fine-tuned to significantly reduce the execution time of the *proxy application* (Step ④), at the cost of losing some details about the memory access pattern of the application. Then, a probabilistic approach is applied by the *proxy application*, which uses the *heatmap* structure to mimic the page access pattern of the original application in the target distributed OS running on the lightweight manycore processor (Step ④). Finally, the *proxy application* outputs statistics about the execution. In the next sections, we describe each of these steps in more details.

3.1 Trace Collection

This stage is composed of 2 main steps, which are described next.

3.1.1 Step 1: Profiling. This step consists in running the application of interest on a standard Linux OS and collecting information about its memory accesses (Step ①). An approach for obtaining such information without any application source code changes is to use a *binary instrumentation tool* such as *Valgrind* [15] or *Intel Pin* [2]. The output of this step is a raw trace file containing the *timestamp* and the address of every memory access made by the program.

3.1.2 Step 2: Trace Preprocessing. Some manipulations may be necessary on the raw trace file output in Step ① to produce a well-formatted trace file that contains only the needed data (Step ②). The complexity of these manipulations are related to the output produced by the profiler. There are mainly three manipulations that may be considered in the preprocessing step: (i) *segment selection*; (ii) *address/page translation*; and (iii) *page number normalization.*

The *segment selection* concerns the selection of user space segments of interest in the raw trace file (*stack*, *heap*, *BSS*, *data* and/or *text*). Depending on how applications are implemented, most of the memory accesses of interest may occur in the *heap* segment (if data is dynamically allocated), in the *BSS/data* segments (if data is statically allocated) or both. The choice of selecting one or more segments is related to both the application itself and to the access pattern that should be mimicked later on.

Since we are interested in the *page access pattern* of the application, the *address/page translation* is necessary only if the events registered by the profiler refer to memory addresses (instead of memory pages). In this case, a trace file manipulation is needed to translate subsequent memory access events on addresses that fall into the same memory page to a single page access event in the output trace file. The size of the memory page should be considered to perform such translation.

Finally, a *page number normalization* is performed, so as to simplify the allocation and management of these pages by the *proxy application* (Step ④). The *proxy application* considers that all pages have consecutive numbers.

3.2 Page Access Pattern

This stage is composed of 2 main steps, which are described next.

3.2.1 Step 3: Heatmap. The preprocessed trace output in Step ② is used to produce a *heatmap* structure in Step ③. The *heatmap* is a compact two-dimensional graphical representation of measured values of numerical data using a chosen color scheme, with one end of the color scheme representing the high values and the other end representing the low values [19]. The variation in color may be by hue or intensity, giving visual insights to the reader about how a phenomenon is clustered or varies over space and time.

We use *heatmaps* to represent the page access pattern of the application, which show how many times each page (or a group of pages) is accessed during specific discrete time periods (the darker the color of the cell of the *heatmap*, the higher the number of page accesses). The x-axis represents a *temporal behavior* (*timestamps* of events), whereas the y-axis represents a *space behavior* (pages or groups of pages). The *resolution* of the *heatmap* can be fine-tuned by grouping several events into *bins* in x and/or y axes. Thus, a maximum resolution is achieved if the *heatmap* uses individual events. The higher is the number of events grouped into *bins*, the lower will be the *heatmap* resolution, resulting in a less accurate page access pattern of the application. By adjusting the resolution of the *heatmap* we can significantly reduce the time spent on mimicking the memory access pattern of the application on the target distributed OS running on the lightweight manycore processor as well as the amount of memory footprint required to store the page access pattern – recall that this is an important constraint for lightweight manycores.

3.2.2 Step 4: Proxy Application. Finally, a *proxy application* mimics the page access pattern of the original profiled application on the distributed OS running on target lightweight manycore processor (Step ④). It takes as input a *heatmap*, which was obtained in Step ③, and a *trial factor* (tf). The *heatmap* is seen by the *proxy application* as a set of histograms (each time *bin* in the x-axis is actually a histogram of page accesses) and it uses a probabilistic approach to generate a variable number of random memory page accesses (*trials*) in each time *bin* based on its frequency distribution. The number of *trials* performed by the *proxy application* in each time *bin* x corresponds to

$$trials_x = \frac{\text{number of events in the original } heatmap \text{ in } x}{tf}. \quad (1)$$

Thus, the higher is the *trial factor*, the lower will be the number of page accesses (*trials*) generated by the *proxy application*. The value of tf impacts on the overall behavior of the memory access pattern being reproduced: the more random values are generated, the closer will be the behavior produced by the *proxy application* compared to the original execution. However, it is possible to reduce considerably the number of random page accesses without losing the original access pattern, as we show in Section 5.1. This allows us to improve considerably the execution time of the *proxy application* as well as to deal with very large *heatmaps*.

4 EVALUATION METHODOLOGY

In this section, we discuss the evaluation methodology to validate our trace-driven methodology. First, we present the applications considered in this study. Then, we describe our experiment environments in more details. Finally, we discuss the experimental design and methods applied in this paper.

4.1 Applications

We selected three applications from CAP Bench [18] that feature different memory access patterns. CAP Bench is an open-source benchmark suite that includes applications suitable to evaluate emerging lightweight manycore processors such as Kalray MPPA-256. These applications allow us to validate our trace-driven methodology as well as to show its potential to evaluate and optimize the memory management service of Nanvix.

In the following paragraphs we give a brief overview of these applications. In-depth descriptions of these applications can be found in [18].

Friendly Numbers (FN) This application computes the amount of friendly numbers in a range $[m, n]$. In number theory, two natural numbers are friendly if they share the same abundancy. In turn, the abundancy A of a given number n is defined as $A(n) = \frac{\sigma(n)}{n}$, where $\sigma(n)$ denotes the sum of divisors of n.

Gaussian Filter (GF) This program implements a Gaussian blur, which is an image smoothing filter. GF consists in applying a specially computed two-dimensional Gaussian mask (m) to an image (i) using a matrix convolution operation. Thus, this application features a stencil pattern.

K-Means (KM) This application implements the K-Means data clustering algorithm, which partitions n points into k partitions in a 2D space. Data points and centroids are evenly and randomly distributed in space. Then, data points are re-clustered into k partitions taking into account the minimum Euclidean distance between them and the centroids. Next, the centroid of each partition is recalculated taking the mean of all points within the partition. The whole procedure is repeated until all centroids remain unchanged.

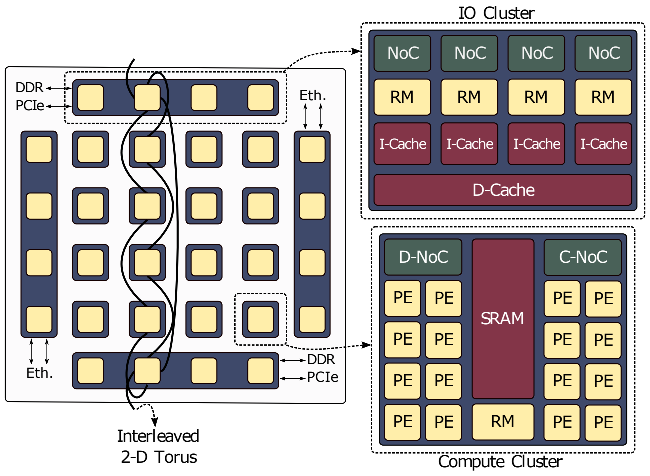


Figure 3: MPPA-256 lightweight manycore processor.

4.2 Experimental Environment

We carried out our experiments on two platforms:

Intel Xeon A 10-core Intel Xeon E5-2640v4 running at 2.40 GHz and 128 GB of RAM, running Ubuntu 16.04.6 LTS with kernel v4.4.0.

Kalray MPPA-256 A NoC-based lightweight manycore processor that features a distributed memory architecture and integrates 288 cores in a single chip, running Nanvix v1.0.

We used the former platform to collect the traces from the CAP Bench applications. Then, we employed the latter platform to validate and evaluate our trace-driven methodology in Nanvix.

Figure 3 pictures an architectural overview of Kalray MPPA-256 (Bostan architecture), which was the processor adopted in this paper. It features 272 general-purpose cores and 16 firmware-cores, called Processing Elements (PEs) and Resource Managers (RMs), respectively, all running at 400 MHz. Cores within the same cluster share some local hardware resources, such as Static Random Access Memory (SRAM) and NoC interfaces, and they have a uniform access latency to these local components. The processor presents two types of clusters:

Compute Clusters They feature 16 PEs, 2 MB of SRAM, 2 NoC interfaces and an RM. Overall, the processor has 16 Compute Clusters in total.

I/O Clusters They feature 4 RMs cores, 4 NoC interfaces, 4 instruction caches, a shared data cache and an SRAM. The processor has 4 I/O Clusters in total, where two of them are connected to a different Double Data Rate (DDR) controller and the other two are attached to Peripheral Component Interconnect (PCI) and Ethernet controllers.

PEs and RMs are designed to target different goals. RMs are dedicated to manage communications, whereas PEs are general-purpose cores, so they can run user programs. It is important to note that hardware cache coherence is not supported in Compute Clusters. Clusters have distinct address spaces, and they communicate with one another by explicitly exchanging messages through one of the available NoCs: a Data NoC (D-NoC), which should be used for system- and user-level large data transfers, and a Control NoC (C-NoC), which is reserved for small control messages. Kalray MPPA-256 has built-in Direct Memory Access (DMA) controllers in their NoC interfaces to enable asynchronous communications.

4.3 Experimental Design and Methods

We divided the experimental evaluation in two sets of experiments, each of which aiming at a different goal. In the following paragraphs we describe each set in more details.

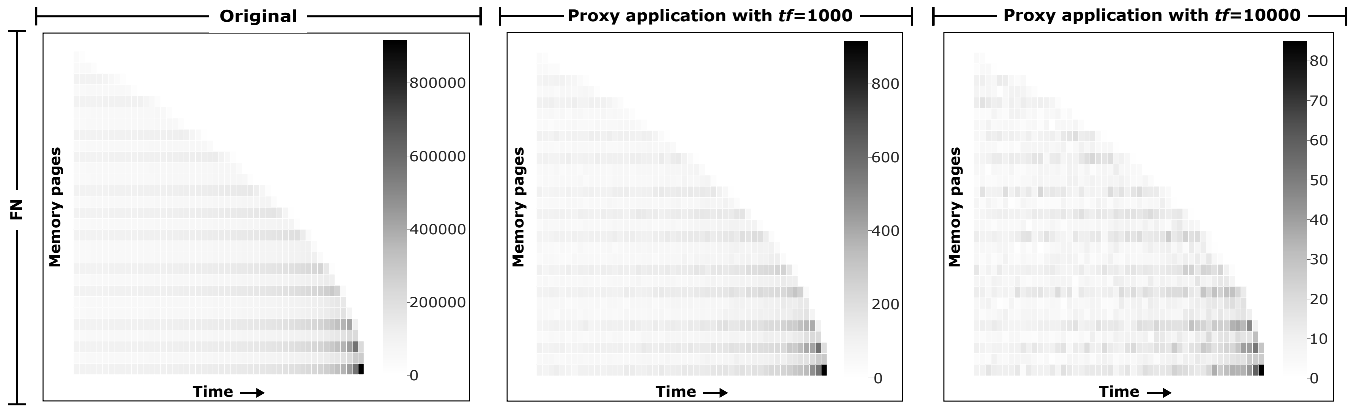
In the first set (Section 5.1), we aim at validating our trace-driven methodology by comparing the *heatmaps* of the applications obtained from the trace files with the *heatmaps* generated by the *proxy application*. If both *heatmaps* are similar, we can conclude that our trace-driven methodology is able to correctly represent and mimic the memory access pattern of the original applications. Moreover, we analyze the *heatmaps* produced by the *proxy application* with different *trial factors*, so as to evaluate if the overall memory access pattern behavior of the original applications are preserved.

In the second set (Section 5.2), we intend to show one case study where our trace-driven methodology can be applied. In this case study, we evaluate the software-managed cache implementation of the memory management service in Nanvix. This cache keeps the pages being accessed by the application in the local memory of Compute Clusters, so as to avoid the high latency of fetching pages from remote memories. Originally, the software-managed cache implementation used a simple First-in First-out (FIFO) page replacement policy, since Kalray MPPA-256 has important hardware limitations[2] that prevent OS developers to implement more sophisticated policies. However, since our *proxy application* mimics the memory access pattern of the original applications by making explicitly calls to the software-managed cache, we were able to implement other policies such as Not Frequently Used (NFU) and Aging. Based on the obtained results, hardware architects can decide whether or not is beneficial to include the necessary hardware support to allow the implementation of smarter page replacement policies at the OS-level.

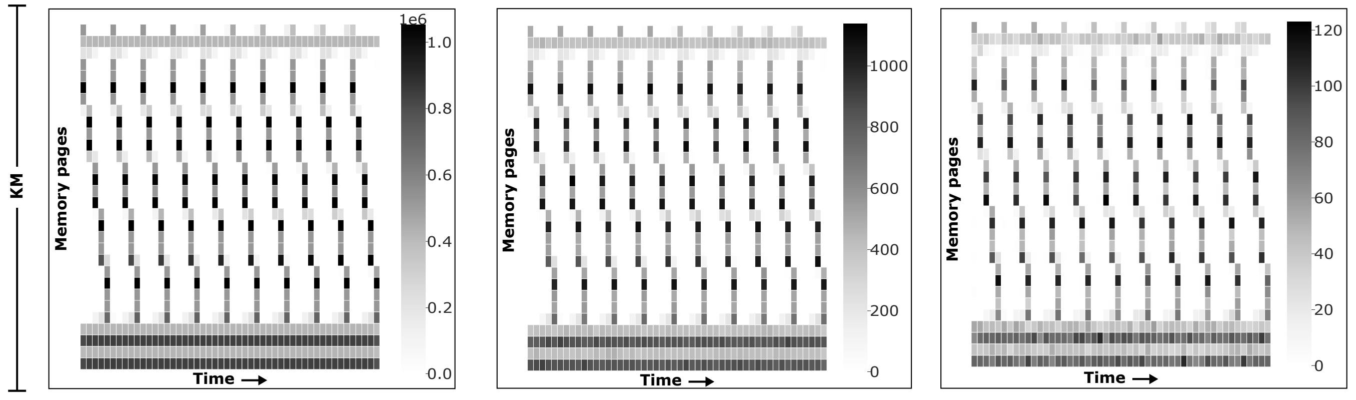
In NFU and Aging policies, there is a reference counter associated to every page, which is initially set to zero. The main difference between these policies is on how the reference counters of pages that have been accessed are updated at fixed time intervals: in NFU, the reference counter is incremented by 1 whereas in Aging, it is first shifted right and then, its most significant bit is set to 1. Due to these operations, Aging ensures that pages referenced more recently will have higher counters. In both policies, the page with the lowest reference counter is chosen to be evicted from the cache.

We ran CAP Bench applications on Intel Xeon to collect their traces and to build their *heatmaps* (Steps ①, ② and ③). We implemented our *proxy application* (Step ④) as user-level application in Nanvix and ran it on the target lightweight manycore processor (Kalray MPPA-256). The size of the software-managed cache in all experiments was 128 KB, which allows us to store 32 pages of 4

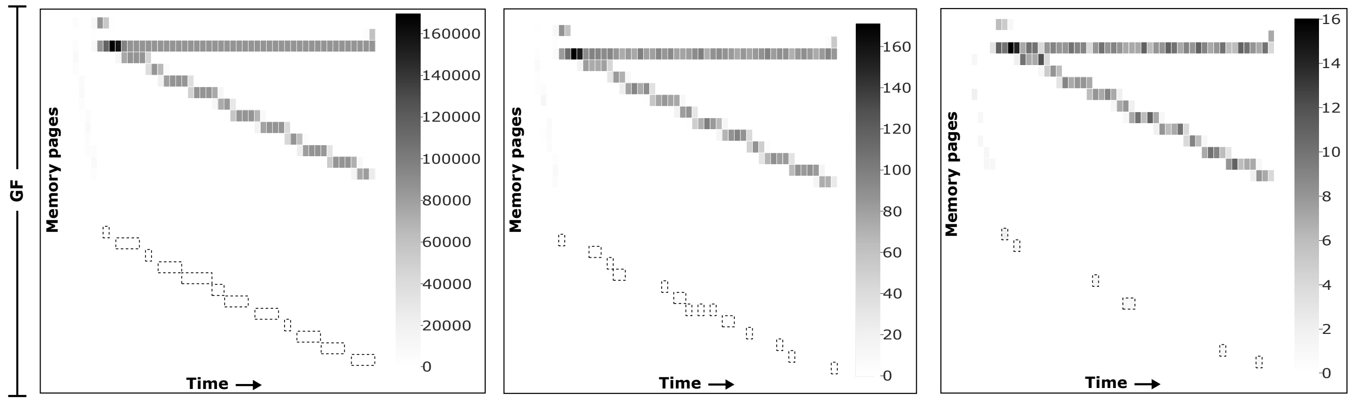
[2]The MMU hardware of this processor does not update page metadata when a page is accessed or modified. Therefore, the only information available for the page replacement policy is the time in which the page was admitted in the system.



(a) FN application.



(b) KM application.



(c) GF application.

Figure 4: *Heatmaps* of FN, GF and KM.

KB each. We selected only the *heap* segments of CAP Bench applications during the trace preprocessing step (Step ②), since data processed by these applications are allocated dynamically.

Our *proxy application* has a deterministic behavior. Thus, a single execution is enough to obtain the desired results. For the software-managed cache experiments, however, we carried out 10 trials of each experimental configuration to eliminate undesired cache warm-up effects. Then, we carried out a single execution to collect the results.

5 RESULTS

First, we discuss about the validation of our trace-drive methodology (Section 5.1). Then, we show the results obtained in our case study with a software-managed cache implementation (Section 5.2).

5.1 Methodology Validation

Figure 4 presents the *heatmaps* built from the traces collected from the execution of CAP Bench applications (*original*) as well as from our *proxy application* with two *trial factors* ($tf = 1000$ and $tf = 10000$). The former value was chosen so as to have a scenario with a moderate yet significant reduction factor. The latter, on the other hand, represents the near maximum possible value for the GF application, which had no more than 160K memory page accesses per cell in the original *heatmap*.

The x-axis in Figure 4 represents the *temporal behavior* (*timestamps* of memory page accesses), whereas the y-axis represents the *space behavior* (memory pages). Noteworthy, cells in Figure 4c were highlighted with dashed boxes to improve visualization, since their values were very close to 0 (i.e., white color). Finally, the resolutions of *heatmaps* were fine-tuned so as to have 30 *bins* in the y-axis and 50 *bins* in the x-axis.

Overall, the *proxy application* was able to mimic the page access pattern of the original applications, even with a high *trial factor*. However, some information about less frequently accessed pages were lost when we increased the *trial factor*, as it can be seen in Figure 4c with $tf = 1000$, and even more with $tf = 10000$ (dashed boxes). This happens due the probabilistic approach of our methodology, which tends to keep information about memory pages that were accessed more frequently.

Table 1 shows the root-mean-square deviation (RMSD) of the results obtained from the *proxy application* with different *trial factors* (the lower the RMSD value, the higher the similarity between the access pattern of the original application and the one produced by the *proxy application*). The very low RMSD values indicate that our methodology is capable of producing a memory access pattern behavior very close to the original applications, even with a high *trial factor*. As it can be noticed, GF showed the highest RMSD values among all applications considered in this study, specially with $tf = 10000$. The main reason for that is two-fold: (i) the number of accesses per page is much lower in this application compared to the other ones, which results in information losses when a high *trial factor* is employed; and (ii) this application has a set of pages that are accessed very few times throughout the execution, thus the probability of accessing a page in this set tends to be very low (near zero percent) with a high *trial factor*.

Another important aspect of our methodology that should be evaluated is the memory footprint to store the page access pattern of the applications. Table 2 presents the size of the original trace files and their respective representation with *heatmaps*. Overall, the size of the *heatmaps* is several orders of magnitude lower than the size of the original trace files. Since our *proxy application* uses *heatmaps* instead of trace files to mimic the memory access pattern of the original applications, it is able to cope with the limited amount of on-chip memory available in lightweight manycore processors. Moreover, it improves the execution of the *proxy application*, since *heatmaps* are allocated in memory and no I/O operation is needed.

Table 1: Root-mean-square deviation (RMSD) of the results obtained from the proxy application.

Application	RMSD ($tf = 1000$)	RMSD ($tf = 10000$)
FN	0.395	1.206
KM	0.198	0.649
GF	1.539	4.594

Table 2: Sizes of trace files and heatmaps.

Application	Trace	Heatmap
FN	1.4 GB	8.0 KB
KM	5.6 GB	7.2 KB
GF	119 MB	4.4 KB

5.2 Case Study: Software-managed Cache

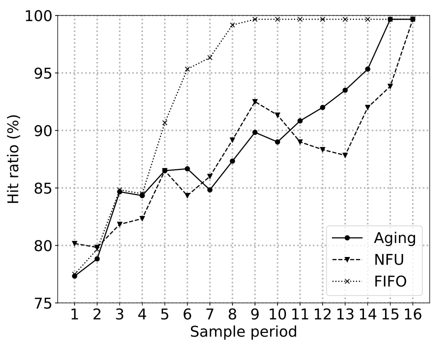
Figure 5 pictures the variation of the software-managed cache hit ratio obtained from our *proxy application* with three different replacement policies: FIFO, NFU and Aging. The hit ratio was computed periodically with a fixed period of $p = 600$ memory access events. Thus, each point in Figure 5 represents the cache hit ratio computed as follows

$$hit\_ratio = \frac{\text{number of cache hits in } p}{p}. \tag{2}$$

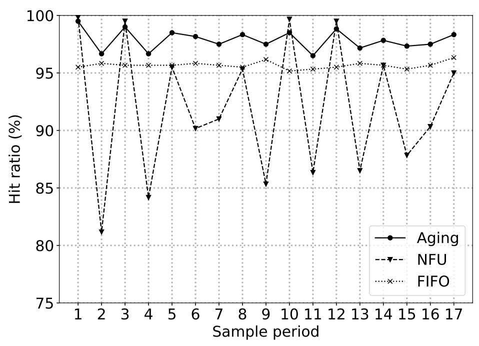
We also fine-tuned the input *heatmaps* used by the *proxy application* as follows: maximum resolution in y-axis (*i.e.*, 1 page per *bin*) and 200 *bins* x-axis. Our empirical tests with FN, KM and GF showed that this configuration was enough to obtain precise results.

Figure 5a shows the hit ratio obtained with FN. As shown in Figure 4a, this application performs memory accesses to every page in the beginning of the execution but its working set reduces as the execution proceeds. This is mainly due to the data access pattern of a nested loop that computes the abundancy of the numbers. This behavior is reflected in the hit ratio, which increases as the execution approaches its end because the working set starts fitting in the cache. The best results were achieved with FIFO, since the page access pattern of FN follows exactly the same behavior.

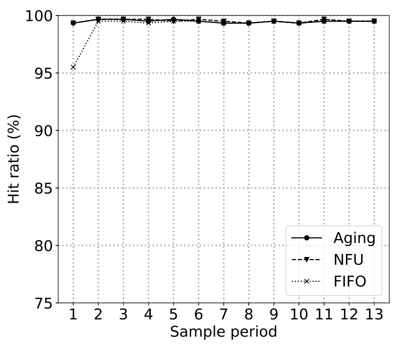
In Figure 5b, however, we observed that Aging was more beneficial for KM. This application has two well-defined page access behaviors: pages on top and bottom ranges in Figure 4b are always accessed throughout the execution whereas middle range pages are accessed in a FIFO ordering fashion. The FIFO page replacement policy is able to correctly keep middle range pages in cache, showing a stable cache hit ratio of 96% on average. NFU, on the other hand, presented a high hit ratio variation with several spikes. The main reason is that NFU constantly evicts middle range pages because they usually have lower reference counters. This problem is solved by Aging, since it is aware of the time span of page use.



(a) FN.



(b) KM.



(c) GF.

Figure 5: Software-managed cache hit ratio.

Finally, Figure 5c shows the results obtained with GF. As shown in Figure 4c, this application has few pages that are constantly accessed throughout the execution (top range) whereas the access pattern of all other pages follows the FIFO rule. We observed that all page replacement policies were able to keep a hit ratio very close to 100%, since the most accessed pages fit in the cache.

6 RELATED WORKS

Tracing is a well-known technique to collect information from the execution of applications. This information is used by researchers and developers to achieve different goals such as debugging, performance optimizations in hardware/software, as well as to allow for more realistic simulations of architectures and applications. When used in simulations, they provide means to evaluate several possible scenarios and configurations in a feasible time. In this section, we discuss some related works that make use of memory accesses traces to help to evaluate and/or optimize software and hardware solutions.

To the best of knowledge, the closest work to ours was proposed by M. M. Rahman and Dueck [13], whose goal was to evaluate and test Automated Memory Management (MM) runtime systems, such as the Java Virtual Machine (JVM). The proposed approach is based on collecting MM operations in traces from benchmark suites at run-time. Since these traces are very large, the authors proposed a trace synthesizer that produces synthetic trace files with basic MM operations for given configuration parameters. These parameters can be adjusted based on the profiling results of real trace files. Their results show that the trace synthesizer can generate more test scenarios, helping developers to evaluate the MM system of the JVM. In contrast to this work, we were interested in distributed OS-level optimizations for lightweight manycores, which brings other challenges due to the intrinsic characteristics and limitations of these processors. Furthermore, our approach is able to mimic the page access pattern of the original applications using *heatmaps* along with a probabilistic approach. Because of that, we neither need to deal with large trace files when reproducing the behavior of the applications nor create new synthetic trace files.

Diener et al. [7] proposed CDSM, a mechanism that uses page faults to detect communications between threads and uses this information to map threads to cores. They used *heatmaps* to store information about the number of communication events between two processes or threads. *Heatmaps* are used to map processes and threads to close processing units according to their communication behavior at run-time. As it can be noticed, Diener et al. [7] used *heatmaps* to achieve a different goal, although sharing some aspects that are similar to our page access pattern representation.

Traces have also been used to build more realistic cache simulators [8, 11, 14]. Moeng et al. [14] proposed the use of GPUs to accelerate a trace-based cache simulator conceived to study the cache coherence in multithreaded workloads and multilevel cache implementations. The information is collected once using a functional simulator and a trace of events is generated. Then, the simulator replays the events registered in the traces. Similarly, Keramidas et al. [11] used both CPU and GPU processors to speedup cache simulation. Finally, Dumas et al. [8] proposed a trace-driven simulation method to accurately compare cache coherence protocols in NoC-based manycores. Their solution helps manycore architects to select and dimension the best cache coherence protocol for their application considering performance and hardware related costs. In contrast to these works, we were interested in evaluating a software-managed cache by carrying out experiments on a baremetal hardware without any cache-level simulation.

7 CONCLUSION

The development of applications for lightweight manycores is very challenging. Software engineers have to explicitly deal with the limited amount of on-chip memory, no cache coherence and multiple address spaces. To tackle these challenges, distributed OSs have been proposed to ease development and improve portability.

In this context, the memory management is one of the most important services offered by distributed OSs. However, the intrinsic characteristics and memory limitations of lightweight manycores bring several challenges to its design and implementation, opening new opportunities for optimizations. A possible approach to help the design and evaluation of the memory management service is to port several applications of different domains to the target distributed OS and carry out experiments with these applications.

Notwithstanding, porting software to distributed OSs for lightweight manycores is a time-consuming and error-prone task.

Aiming at this problem, in this work, we proposed a trace-driven methodology that can be used to evaluate and optimize features of a memory management service of distributed OSs for lightweight manycores. Thanks to a compact representation of memory access patterns, our methodology is capable of mimicking the memory access pattern of the original applications on the target distributed OS running on a lightweight manycore with low footprints. We integrated our methodology in Nanvix and validated it using three applications from CAP Bench. Furthermore, we carried out a case study using a software-managed cache implementation available in Nanvix. Our methodology enabled us to evaluate different page replacement policies on Kalray MPPA-256, even without the required support from the architecture to implement them.

As future work, we intend to apply our methodology to help the design and evaluation of new optimizations in Nanvix, such as the software prefetching module and a more sophisticated page sharing algorithm. We also intend to consider other applications from CAP Bench and other benchmarks. Finally, we intend to apply our methodology to other distributed OSs and/or lightweight manycore processors.

ACKNOWLEDGMENTS

This work was partially supported by *Conselho Nacional de Desenvolvimento Científico e Tecnológico* – Brasil (CNPq) and by *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* – Brasil (CAPES) – Finance Code 001 and under the CAPES-PrInt Program Grant No.: 88881.310783/2018-01.

REFERENCES

[1] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Conference on Architectural Support for Programming Languages and Operating Systems (APLOS)*. ACM, New York, USA, 189–203.

[2] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C. Luk, G. Lyons, H. Patil, and A. Tal. 2010. Analyzing Parallel Programs with PIN. *Computer* 43, 3 (2010), 34–41.

[3] Shekhar Borkar. 2007. Thousand core chips. In *Design Automation Conf. (DAC)*. ACM Press, New York, USA, 746.

[4] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: An Operating System for Many Cores. In *Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, San Diego, USA, 43–57.

[5] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoît Ganne, Pierre Guironnet de Massas, Francois François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frederic Riss, and Thierry Strudel. 2013. A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications . In *IEEE High Performance Extreme Computing Conference (HPEC) (HPEC '13)*. IEEE, Waltham, USA, 1–6.

[6] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. 2013. A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. In *International Conference on Computational Science (ICCS)*, Vol. 18. Elsevier, Barcelona, Spain, 1654–1663.

[7] Matthias Diener, Eduardo H.M. Cruz, Philippe O.A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2015. Communication-aware process and thread mapping using online communication detection. *Parallel Comput.* 43 (2015), 43 – 63.

[8] Julie Dumas, Eric Guthmuller, César Fuguet Tortolero, and Frédéric Pétrot. 2017. Trace-driven exploration of sharing set managementstrategies for cache coherence in manycores. In *International New Circuits and Systems Conference (NEWCAS)*. IEEE Computer Society, Strasbourg, France, 77–80.

[9] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei Ge, Jian Zhang, Yangang Wang, Chunbo Zhou, and Guangwen Yang. 2016. The Sunway TaihuLight Supercomputer: System and Applications. *Science China Information Sciences* 59, 7 (jul 2016), 72001–720016. https://doi.org/10.1007/s11432-016-5588-7

[10] Simon J Hollis, Edward Ma, and Radu Marculescu. 2016. nOS: A Nano-Sized Distributed Operating System for Many-Core Embedded Systems. In *International Conf. on Computer Design (ICCD)*. IEEE, Scottsdale, USA, 177–184.

[11] Georgios Keramidas, Nikolaos Strikos, and Stefanos Kaxiras. 2011. Multicore cache simulations using heterogeneous computing on general purpose and graphics processors. In *Digital System Design (DSD)*. IEEE, Oulu, Finland, 270–273.

[12] Florian Kluge, Mike Gerdes, and Theo Ungerer. 2014. An Operating System for Safety-Critical Applications on Manycore Processors. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, Reno, USA, 238–245.

[13] K. B. Kent M. M. Rahman, K. Nasartschuk and G. W. Dueck. 2016. Trace Files for Automatic Memory Management Systems. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Suita, Japan, 9–12.

[14] Michael Moeng, Sangyeun Cho, and Rami Melhem. 2011. Scalable multi-cache simulation using GPUs. In *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, Singapore, Singapore, 159–167.

[15] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Notices*, Vol. 42. Association for Computing Machinery, New York, NY, USA, 89–100.

[16] Andreas Olofsson, Tomas Nordstrom, and Zain Ul-Abdin. 2014. Kickstarting High-performance Energy-efficient Manycore Architectures with Epiphany. In *Asilomar Conference on Signals, Systems and Computers (ACSSC)*. IEEE, Pacific Grove, USA, 1719–1726.

[17] Pedro H. Penna, João V. Souto, Davidson F. Lima, Márcio Castro, François Broquedis, Henrique C. de Freitas, and Jean-François Méhaut. 2019. On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. In *Brazilian Symposium on Computing Systems Engineering (SBESC)*. IEEE Computer Society, Natal, Brazil, 1–8.

[18] Matheus A. Souza, Pedro Henrique Penna, Matheus M. Queiroz, Alyson D. Pereira, Luís Fabricio W. Góes, Henrique C. Freitas, Márcio Castro, Philippe O. A. Navaux, and Jean-François Méhaut. 2017. CAP Bench: A Benchmark Suite for Performance and Energy Evaluation of Low-power Many-core Processors. *Concurrency and Computation: Practice and Experience* 29, 4 (2017), e3892.

[19] H. Suematsu, S. Yagi, T. Itoh, Y. Motohashi, K. Aoki, and S. Morinaga. 2014. A Heatmap-Based Time-Varying Multi-variate Data Visualization Unifying Numeric and Categorical Variables. In *International Conference on Information Visualisation (IV)*. IEEE, Paris, France, 84–87.

[20] Anish Varghese, Bob Edwards, Gaurav Mitra, and Alistair P Rendell. 2014. Programming the Adapteva Epiphany 64-Core Network-on-Chip Coprocessor. In *International Parallel Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, Phoenix, USA, 984–992.

[21] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally. 2014. Scaling the Power Wall: A Path to Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, New Orleans, USA, 830–841.