UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Emmanuel Podestá Junior

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

Florianópolis
2022

Emmanuel Podestá Junior

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

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.
Orientador: Prof. Márcio Bastos Castro Dr.

Florianópolis
2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Podestá Junior, Emmanuel
A Trace-Driven Methodology to Evaluate Memory Management Services of Distributed Operating Systems for Lightweight Manycores / Emmanuel Podestá Junior ; orientador, Márcio Bastos Castro, 2022.
82 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2022.

Inclui referências.

1. Ciência da Computação. 2. Metodologia baseada em traces. 3. Sistema Operacional Distribuído. 4. Gerenciamento de Memória. 5. Lightweight Manycores. I. Castro, Márcio Bastos. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Emmanuel Podestá Junior

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

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Mestre em Ciência da Computação.

Documento assinado digitalmente
PATRICIA DELLA MEA PLENTZ
Data: 20/12/2022 14:30:42-0300
CPF: ***.243.700-**
Verifique as assinaturas em https://v.ufsc.br

Prof^{a} Patricia Della Méa Plentz, Dra.
Coordenadora do Programa

Documento assinado digitalmente
Marcio Bastos Castro
Data: 20/12/2022 10:37:09-0300
CPF: ***.707.720-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Márcio Bastos Castro Dr.
Orientador

Florianópolis, 2022.

Para Lady

AGRADECIMENTOS

À minha família que esteve do meu lado nos momentos mais difíceis. Aproveito para agradecer todo o amor, suporte e carinho que me deram durante toda a minha vida.

Ao meu orientador, Prof. Márcio Castro, pelas reuniões construtivas, diversos ensinamentos e o fantástico apoio durante vários anos. Com toda certeza, tais fatores me tornaram tanto uma pessoa melhor, quanto um melhor profissional.

Ao meu colega, Pedro Penna, que me auxiliou em vários momentos e, no curto período que estive pessoalmente com ele, me ensinou diversas coisas para o meu amadurecimento pessoal e profissional.

Ao Jean-François Méhaut que, em conjunto com meu orientador e meu colega, Pedro Penna, possibilitou uma das melhores experiências da minha vida ao realizar parte do meu mestrado na França.

Aos meus colegas de laboratório que me ajudaram através de diversas indagações e outros auxílios durante o desenvolvimento desta dissertação.

À minha psicóloga, Manuella Bahls, que esteve comigo durante todo esses anos e auxiliou imensamente o meu amadurecimento.

Aos meus amigos que me proporcionaram vários momentos de descontração e deram o apoio necessário para eu continuar enfrentando meus desafios.

A todos que não pude citar aqui que me auxiliaram com dicas, sugestões ou de qualquer outra forma o meu amadurecimento e o desenvolvimento desta dissertação.

Às minhas cachorras, Liza, Tess e Linda, que, por mais que não falem, me deram um grande apoio emocional durante tempos difíceis. Em especial para a Lady, que me ajudou em várias madrugadas.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) - Código de Financiamento 001.

"All we have to decide is what to do with the time that is given to us."
Gandalf

RESUMO

Os *lightweight manycores* pertencem a uma nova classe de processadores emergentes de baixa potência para a era *Exascale*. Esses processadores apresentam vários desafios para o desenvolvimento de aplicações, como arquitetura de memória distribuída, quantidade limitada de memória no chip e nenhuma coerência de *cache*. Recentemente, Sistemas Operacionais distribuídos foram propostos para enfrentar esses desafios de forma transparente. Nesses sistemas, diferentes serviços do Sistema Operacional são implantados nos núcleos do processador, sendo o serviço de gerenciamento de memória um dos mais importantes. No entanto, os desafios citados anteriormente sobre *lightweight manycores* trazem vários obstáculos para o *design*, implementação e otimizações futuras de serviços de gerenciamento de memória. Esta dissertação propõe uma metodologia baseada em *traces* para avaliar e otimizar recursos do serviço de gerenciamento de memória em Sistemas Operacionais distribuídos para *lightweight manycores*. Usando uma representação compacta do padrão de acesso às páginas das aplicações, a metodologia consegue imitar o padrão de acesso à memória das aplicações originais no Sistema Operacional distribuído rodando em um *lightweight manycore*. A metodologia foi integrada em um Sistema Operacional distribuído (Nanvix) e validada usando cinco aplicações de um *benchmark* específico para *lightweight manycores* (CAP Bench). Em seguida, a metodologia foi aplicada para realizar um estudo de caso usando uma implementação de *cache* gerenciada por *software* disponível no Nanvix. A metodologia permitiu avaliar várias configurações e diferentes políticas de substituição de páginas no processador Kalray MPPA-256, mesmo sem o suporte necessário da arquitetura para implementá-los.

Palavras-chave: Metodologia baseada em *traces*. Sistema Operacional Distribuído. Gerenciamento de Memória. *Cache* gerenciada por *software*. *Lightweight Manycores*.

RESUMO ESTENDIDO

Introdução

Historicamente, a comunidade de computação de alto desempenho aumentava o desempenho de sistemas escalando a frequência do *clock*. Infelizmente, o aumento linear da frequência do *clock* aumenta quadraticamente a temperatura do *chip*. Desta forma, a incansável necessidade por desempenho conduziu ao problema denominado *power wall* (VILLA et al., 2014). Em resposta a esse problema, a indústria migrou para outros *designs* de processadores, como os *multicores* e, posteriormente, os *manycores*.
Neste contexto surgiu uma nova classe de processadores com foco em alto desempenho e baixa potência, denominados *lightweight manycores*. Contudo, apesar de apresentarem um ambiente interessante e melhor eficiência de potência em comparação à *multicores* de propósito geral (FRANCESQUINI et al., 2015), os *lightweight manycores* apresentam certas características que prejudicam o desenvolvimento de *software* para o processador (VARGHESE et al., 2014; CASTRO et al., 2016; CASTRO et al., 2014). Devido à isso, Sistemas Operacionais distribuídos foram introduzidos para trazer abstrações e diversos outros serviços que solucionam este problema.
Dentre todos os serviços dos Sistemas Operacionais distribuídos para *lightweight manycores*, o serviço de gerenciamento de memória é um dos mais importantes. O *design* e implementação de um subsistema do serviço de gerenciamento de memória é um problema em aberto e abre caminho para diversas otimizações. Neste contexto, uma solução clássica para avaliar otimizações é adaptar diversas aplicações de diferentes domínios para a arquitetura subjacente e realizar experimentos com técnicas de *profilling*. Contudo, existem dois problemas com essa abordagem: (i) *lightweight manycores* podem não suportar bibliotecas *runtime* específicas de aplicações; e (ii) aplicações não levam em consideração a API dos *lightweight manycores*. Desta forma, adaptar diversas aplicações é uma tarefa onerosa e susceptível a erros.
Uma possível solução para esse problema é usar simulações baseadas em *traces* (UHLIG; MUDGE, 1997) para: (i) imitar características de um grande conjunto de aplicações; (ii) auxiliar no estudo do desempenho de diferentes otimizações; (iii) ajustar cada otimização e verificar seu desempenho sem a necessidade de adaptar aplicações; e (iv) ter conhecimento prévio do comportamento da aplicação e realizar decisões mais precisas sobre ela e a necessidade de sua portabilidade.

Objetivos

Esta dissertação tem como principal objetivo propor uma metodologia baseada em *traces* usada para avaliar e otimizar recursos do serviço de gerenciamento de memória em Sistemas Operacionais distribuídos para *lightweight manycores*. Desta forma, o conjunto completo de contribuições é listado abaixo:

(i) uma nova metodologia baseada em *traces* que ajuda desenvolvedores de Sistemas Operacionais distribuídos a avaliar e otimizar recursos de serviços de gerenciamento de memória sem a necessidade de adaptar aplicações;
(ii) uma integração da metodologia proposta em um Sistema Operacional distribuído de código aberto para *lightweight manycores*, denominado Nanvix (PENNA et al., 2019); e
(iii) um estudo de caso de uma avaliação de *cache* gerenciada por *software* no Nanvix utilizando a metodologia proposta.

Metodologia

Os seguintes passos metodológicos foram seguidos para cumprir os objetivos citados:

(i) Estudar e avaliar os mecanismos de *tracing* utilizados regularmente no contexto de High Performance Computing (HPC);
(ii) Selecionar o mecanismo de *tracing* mais apropriado para coletar informações sobre o padrão de acesso à memória das aplicações;
(iii) Estudar e propor técnicas para filtrar o arquivo de *trace* para reduzir o seu tamanho;
(iv) Propor e implementar um *benchmark* genérico baseado em *traces* para imitar o padrão de acesso à memória da aplicação original;
(v) Integrar o *benchmark* baseado em *traces* com o Nanvix; e
(vi) Implementar uma otimização no serviço de gerenciamento de memória do Nanvix e avaliá-lo utilizando a metodologia baseada em *traces* proposta.

Resultados e Discussão

Nesta dissertação, foi escolhido o CAP Bench (SOUZA et al., 2010) para estudar a efetividade da metodologia. O *benchmark* possui diversas aplicações, onde foram escolhidas cinco delas. Nossos experimentos foram realizados em duas plataformas: (i) Intel Xeon, uma plataforma comum de alto desempenho; e (ii) Kalray MPPA-256, um processador *lightweight manycore*.

Os resultados foram divididos em dois conjuntos. O primeiro visa a validação da metodologia comparando o padrão de acessos à memória obtidos com o *benchmark* genérico baseado em *traces* e o *trace* original. Por outro lado, o segundo mostra o estudo de caso de uma *cache* gerenciada por *software*, onde a metodologia proposta pode ser aplicada.

No primeiro conjunto, utilizou-se a métrica Root-mean Square Deviation para obtermos a quantificação da diferença entre ambos os padrões de acesso à memória. A métrica determina a diferença entre valores previstos por um modelo e os valores observados, onde quanto mais perto de zero o valor, menor a diferença entre eles. Os resultados mostraram valores inferiores a 1.5 para todas as aplicações. Desta forma, podemos concluir que a metodologia consegue imitar o padrão de acesso à memória das aplicações originais. Além disso, ela consegue reduzir significativamente o espaço de armazenamento em disco secundário necessário para armazenar os *traces*.

Em contraste, o segundo conjunto mostrou ser possível avaliar diversas configurações e algoritmos de substituição de *cache* no Kalray MPPA-256 com a metodologia, até mesmo, sem o suporte da arquitetura para implementá-los. Pode-se concluir que os algoritmos FIFO e *Aging* são os melhores algoritmos, aproximando-se de quase 100% de taxa de *hits* na *cache* para a maioria das aplicações utilizadas.

Considerações Finais

Esta dissertação apresentou uma metodologia baseada em *traces* que pode ser usada para avaliar e otimizar recursos de serviços de gerenciamento de memória em Sistemas Operacionais distribuídos para *lightweight manycores*. A metodologia consegue imitar o padrão de acesso à memória das aplicações originais em um Sistema Operacional distribuído para *lightweight manycores* com baixa quantidade de armazenamento em disco secundário. Nós integramos a metodologia no Nanvix e a validamos utilizando cinco aplicações do CAP Bench. Nossos resultados mostram que a diferença entre o *trace* original e o gerado pelo *benchmark* genérico baseado em *traces* é baixa com um Root-mean Square Deviation inferior a 1.5 para todas as aplicações. Além disso, realizamos um estudo de caso de uma implementação de *cache* gerenciada por *software* no Nanvix. A metodologia conseguiu

avaliar diversas configurações e algoritmos de substituição de *cache* no Kalray MPPA-256, mesmo sem o suporte da arquitetura para implementá-los.
Como trabalhos futuros, nós pretendemos aplicar a metodologia para auxiliar no *design* e avaliação de novas otimizações no Nanvix, como *software prefetching* e um algoritmo de compartilhamento de páginas mais sofisticado. Ademais, pretendemos considerar outras aplicações do CAP Bench e outros *benchmarks*. Por fim, queremos aplicar a metodologia em outros Sistemas Operacionais distribuídos e/ou *lightweight manycores*.

Palavras-chave: Metodologia baseada em *traces*. Sistema Operacional Distribuído. Gerenciamento de Memória. *Cache* gerenciada por *software*. *Lightweight Manycores*.

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 have been proposed to address these challenges in a transparent way. In these systems, different Operating Systems services are deployed across the processor cores, being the memory management service one of the most important. However, the aforementioned challenges of lightweight manycores bring several demands to the design, implementation and future optimizations of memory management services. This dissertation proposes a trace-driven methodology to evaluate and optimize features of a memory management service of distributed Operating Systems for lightweight manycores. By using a compact representation of the page access pattern of applications, our methodology is capable of mimicking the memory access pattern of the original applications on the target distributed Operating System running on a lightweight manycore. The methodology was integrated in a distributed Operating System (Nanvix) and validated using five applications from a specific benchmark for lightweight manycores (CAP Bench). Then, the methodology was applied to carry out a case study using a software-managed cache implementation available in Nanvix. The methodology enables evaluation of several configurations and different page replacement policies on Kalray MPPA-256 processor, even without the support from the architecture to implement them.

Keywords: Trace-Driven Methodology. Distributed Operating System. Memory Management. Software-managed Cache. Lightweight Manycores.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface.	23, 48
C-NoC	Control NoC.	49
CC-NUMA	Cache-Coherent Non-Uniform Memory Access.	26
CPU	Central Process Unit.	25, 26, 28, 33
D-NoC	Data NoC.	49
DDR	Double Data Rate.	49
DMA	Direct Memory Access.	28, 50
DPS	Distributed Paging System.	30
DRAM	Dynamic Random Access Memory.	27
FAST	Features from Accelerated Segment Test.	48
FIFO	First-in First-out.	12, 34, 50, 57, 58, 62, 63, 66, 67, 68, 69, 71, 72, 73, 74
FN	Friendly Numbers.	12, 48, 52, 53, 57, 58, 61, 62, 63, 66, 67, 69
FOS	Factored Operating System.	29
FPGA	Field Programmable Gate Array.	22, 30, 76
GF	Gaussian Filter.	12, 13, 48, 52, 53, 57, 58, 59, 61, 63, 65, 67, 69, 70, 71
GPU	Graphics Processing Unit.	28, 75, 76
HAL	Hardware Abstraction Layer.	29
HPC	High Performance Computing.	9, 24
I/O	Input/Output.	27, 28, 55
IPC	Inter-Process Communication.	30
IS	Integer Sort.	12, 13, 48, 49, 52, 54, 57, 59, 60, 62, 64, 65, 66, 67, 68, 71, 72
JIT	just-in-time.	33
JVM	Java Virtual Machine.	75
KM	K-Means.	12, 13, 48, 52, 53, 54, 61, 62, 63, 64, 66, 67, 68, 69, 70
LRU	Least Recently Used.	35
LU	LU Factorization.	12, 13, 48, 49, 52, 54, 60, 62, 64, 67, 71, 72, 73
MIMD	Multiple Instruction Multiple Data.	28

NC-NUMA	Non Cache-Coherent Non-Uniform Memory Access.	26
NFU	Not Frequently Used.	12, 13, 34, 35, 51, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 73
NoC	Network-on-Chip.	21, 22, 23, 27, 28, 37, 49, 50, 75, 76
NUMA	Non-Uniform Memory Access.	12, 25, 26, 28
OS	Operating System.	12, 22, 23, 24, 25, 29, 30, 31, 36, 37, 38, 42, 44, 45, 46, 51, 75, 76, 77
PCI	Peripheral Component Interconnect.	49
PE	Processing Element.	49
POSIX	Portable Operating System Interface.	22
RAM	Random Access Memory.	25, 26
RM	Resource Manager.	49
RMem	Remote Memory.	30
RMSD	Root-mean Square Deviation.	14, 54, 55, 77
RV	Runtime Verification.	31
SC	Second Chance.	34
SHM	Shared Memory.	30
SRAM	Static Random Access Memory.	27, 49
TSP	Traveling-Salesman Problem.	48
UMA	Uniform Memory Access.	12, 25, 26
VFS	Virtual File System.	30

CONTENTS

1 INTRODUCTION

Historically, the industry was able to design faster and more advanced uniprocessor systems by increasing the number of transistors per chip and by scaling up the clock frequency. In addition, features such as speculative execution, instruction-level parallelism, out-of-order execution and larger caches have contributed to improve the overall performance of these processors. Unfortunately, a linear increase in clock frequency results in a exponential growth of power consumption (BROOKS et al., 2000; BOSE, 2011) and affects considerably the chip temperature, thus contributing to the arise of the well-known power wall problem (VILLA et al., 2014). As a response to this challenge, the industry transitioned towards multicore designs, which feature more than one processor core operating at lower clock frequency to increase the aggregate performance. In contrast to the previous approach, this design is better suited to deal with power efficiency, because a linear increase in the number of cores leads to a linear increase in power consumption (BORKAR, 2007; GSCHWANDTNER et al., 2015).

As the number of cores increases to achieve more performance, the multicore design begins to consume more power and, thus, more energy. To solve this challenge, many-cores processors emerged, integrating several less complex cores instead of complex ones. These processors are more energy efficient since the power consumption reduces linearly as performance reduces by the square-root of core complexity (BORKAR, 2007). Therefore, with a small reduction on performance, the processor can reduce power consumption significantly. As new ways to tackle power consumption and performance were unveiled, a new subclass of manycores processors emerged, named lightweight manycores (BORKAR, 2007).

Lightweight manycores belong to a new class of highly parallel low-power many-core chips suitable for environments with power constraints. Processors in this class have hundreds of low-power processing cores that deal with data and task parallelism. These processing cores are grouped in *clusters*, each one having its own limited local memory and Network-on-Chip (NoC) interfaces to allow communications with other clusters. This design may lead to a distributed memory configuration, with limited amount of on-chip memory and no cache coherence protocols among different clusters. Indeed, lightweight manycores achieve better energy efficiency than general-purpose multicore processors (FRANCESQUINI et al., 2015). Examples of such processors are Sunway SW26010 (FU et al., 2016), Adapteva Epiphany (OLOFSSON; NORDSTROM; UL-ABDIN, 2014) and Kalray MPPA-256 (DINECHIN et al., 2013), which feature hundreds of low-power cores in a single chip. Specially, Sunway SW26010 is used in the Sunway TaihuLight, one of the fastest supercomputers to date, featuring 10.6 million low-power cores (FU et al., 2016).

Although lightweight manycores may present better energy efficiency than general-purpose multicore processors (FRANCESQUINI et al., 2015), they present architectural characteristics that hinder the development of applications (VARGHESE et al., 2014; CASTRO et al., 2016; CASTRO et al., 2014). For instance, 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 (FRANCESQUINI et al., 2015). Moreover, communications must take into account the NoC topology whenever possible to improve bandwidth and reduce latency.

Therefore, lightweight manycores call out for a distributed Operating System (OS) design to address these challenges in a transparent fashion (BOYD-WICKIZER et al., 2008; WENTZLAFF; AGARWAL, 2009; HOLLIS; MA; MARCULESCU, 2016; PENNA et al., 2019). The multikernel design (BAUMANN et al., 2009) was introduced to heed the call. This design deploys independent OS kernel instances on the processor. Each kernel provides a bare-minimum abstraction and fully featured system services are implemented in a distributed fashion. Moreover, the distributed architecture brings: (i) better scalability; (ii) hardware-neutral characteristic enabling portability between several architectures; and (iii) explicit inter-core communication which provides more room for optimizations and efficient use of the processor network (BAUMANN et al., 2009). Each kernel functionality in this design can be further extended to achieve a higher level of abstraction (PENNA et al., 2019). For instance, a distributed OS may provide caching, paging and communication services to achieve a suitable environment for users. Finally, since the multikernel OS design is very flexible, it can feature specific OS services to fulfill users' needs.

1.1 MOTIVATION

Among the services offered by distributed OSs for lightweight manycores, the memory management is one of the most important. This service overcomes most programming intricacies of a distributed memory architecture, and it must exploit the NoC in order to achieve decent performance. Unfortunately, current open-source distributed OSs for manycore processors such as MOSSCA (KLUGE; GERDES; UNGERER, 2014) and M^3 (ASMUSSEN et al., 2016) do not feature a software stack with a transparent memory management service compliant with Portable Operating System Interface (POSIX). Additionally, these OSs run on an in-house simulator and a processor prototype implemented in a Field Programmable Gate Array (FPGA), respectively. The only exception found in the literature that features a POSIX-compliant kernel with the first efforts on providing a memory management service is Nanvix[1] (PENNA et al., 2019). Nanvix is a research OS for lightweight manycores based on a multikernel structure (BAUMANN et al., 2009), which is currently developed by Brazilian (PUC Minas and UFSC) and French (*Université de Grenoble Alpes*) institutions. Its system is based on asymmetric microker-

[1] https://github.com/nanvix

nel instances, where each kernel communicates with each other through a message-passing approach. Nanvix supports multiple baremetal[2] architectures, such as Kalray MPPA-256. Furthermore, it features a built-in lightweight manycore simulator to enable OS development and debugging on top of Linux. To the best of our knowledge, Nanvix is the only open-source distributed OS that runs on commercially available baremetal lightweight manycores.

The design and implementation of 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. In this context, a classical solution to evaluate different optimizations is to port several applications of different domains to the underlying architecture. Then, experiments with a profiling technique should be applied to understand their behavior (SOUZA et al., 2010). There are two main problems with this approach: (i) lightweight manycores may not support specific application runtime libraries; and (ii) applications do not take into consideration underlying lightweight manycore Application Programming Interface (API). Hence, it hinders the evaluation in a more general-purpose scenario.

To solve this problem, the chosen approach has to remove the need to manually port applications. One possibility is to use trace-driven simulations (UHLIG; MUDGE, 1997) to: (i) mimic the characteristics of a broad range of applications; (ii) help to study the performance of different optimizations; (iii) adapt each optimization and verify its performance without the need of porting several applications; and (iv) know beforehand about application behavior and the need for portability.

Following this idea, Dumas et al. (2017) proposed a trace-driven simulation for cache evaluation. However, their solution did not consider the limited amount of on-chip memory in lightweight manycores. Due to the size of a trace file, a new method should be proposed to decrease the trace file size without consuming several resources from the distributed OS or losing the overall representation of the application behavior. In contrast, M. M. Rahman K. Nasartschuk & Dueck (2016) proposed a trace-driven approach to evaluate and test Automated Memory Management (MM) runtime systems. To deal with large trace files, they used a trace synthesizer that produces configurable synthetic trace files. However, they did not consider distributed OS-level optimizations for lightweight manycores. We argue that a trace-driven approach concerned with memory limitations and designed to mimic the behavior of applications may provide a means to analyze and study different scenarios for several applications in distributed OS for lightweight manycores.

[2] Architectures without a base OS.

1.2 GOALS AND CONTRIBUTIONS

To help with applications analyzes and relieve developers from the burden of porting several applications to the target distributed OS, in this work 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. The complete set of goals are listed bellow:

(i) Study and evaluate broadly used tracing mechanisms in High Performance Computing (HPC);
(ii) Select the most appropriate tracing mechanism to collect information about the memory access pattern of applications;
(iii) Study and propose techniques to filter the trace file in order to reduce the file size;
(iv) Propose and implement a generic trace-driven benchmark to mimic the memory access pattern of the original application;
(v) Integrate the trace-driven benchmark into Nanvix; and
(vi) Implement an optimization in Nanvix memory management service and evaluate it using the proposed trace-driven methodology.

Overall, 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 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 (PENNA et al., 2019); and
(iii) a case study of a software-managed cache evaluation in Nanvix using the proposed methodology.

1.3 WORK ORGANIZATION

The remainder of this work is organized as follows. Chapter 2 shows core concepts of our work. Chapter 3 demonstrates our proposal and its main stages. Chapter 4 presents our experiments and results. Chapter 5 discusses related work. Finally, Chapter 6 concludes this work.

2 BACKGROUND

In this chapter, we present the background on important subjects related to this dissertation. Sections 2.1 and 2.2 present an overview of multiprocessors and lightweight manycores. Section 2.3 discusses distributed OSs for lightweight manycores. Section 2.4 provides a background on system analysis and monitoring. Finally, Section 2.5 presents basic information about cache memories and their placement techniques, which will be used as a case study to apply the trace-driven methodology proposed in this work.

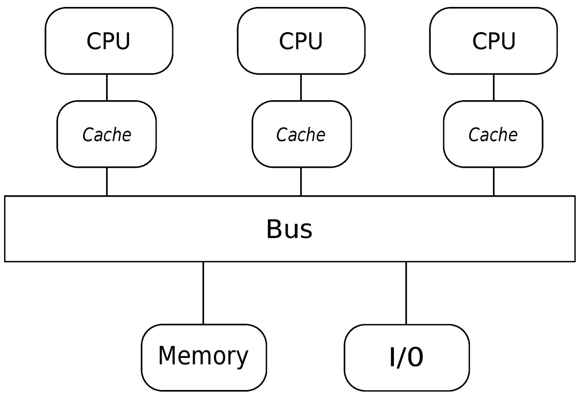
2.1 MULTIPROCESSORS

Multiprocessors consist of one or more Central Process Units (CPUs) that share a Random Access Memory (RAM). Hence, CPUs may communicate through read/write memory operations in the same physical address space. The interconnection between CPUs and RAM can be more or less complex depending on the architecture design. On Uniform Memory Access (UMA) architectures, CPUs can address the RAM in a near-constant time, being the bus one of the most common solutions. Figure 1 shows a simplified UMA multiprocessor architecture, where CPUs use shared memory to communicate with each other through a bus. When a CPU performs a communication, the bus is checked to see if it is available. If the bus is busy, the CPU waits until it is available. Then, the CPU puts the required word address in the bus, uses control signals and waits until the memory sends the word through the bus. This method is suitable for multiprocessors composed of few CPUs. Thus, this design is not scalable as the bus becomes the system bottleneck with a high number of CPUs.

Caches can alleviate this problem, enabling read requisitions to be fulfilled by the cache, thus decreasing the number of communications. To achieve consistency inside the system, coherence protocols are used to keep all caches synchronized. For instance, a cache-coherence protocol can be defined as follows: when a CPU wants to write a word, other caches that have the same word are notified. If a cache has a modified copy, the word will be written in memory. Otherwise, other caches can discard the word and let the writer fetch the cache block from memory before modifying it. Furthermore, more cache levels can be added to reduce access to main memory and cache size may be increased to improve performance. However, due to architectural limits, a high quantity of cache levels is undesirable, and a huge cache size slows access time and harms overall system performance. In addition, new interconnections, such as crossbar switches, can be introduced to connect more CPUs and alleviate further the bottleneck problem (TANENBAUM; BOS, 2015).

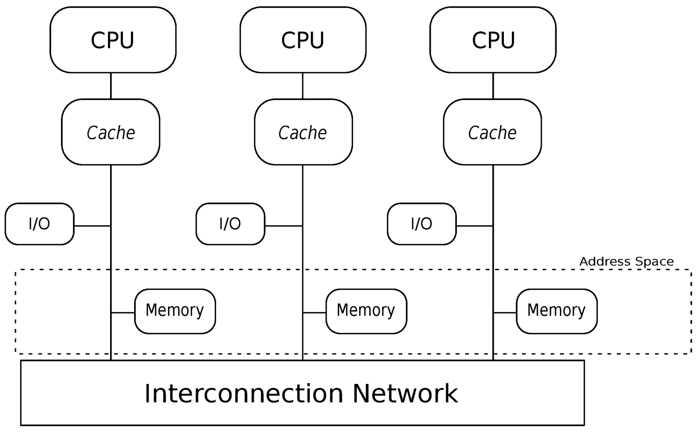
On Non-Uniform Memory Access (NUMA) architectures, the RAM is divided into NUMA nodes and each node is placed close to a CPU. Because of that, the time taken to address the memory will vary, depending on the distance between the node and the CPU.

Figure 1 – Simplified UMA multiprocessor.



Source: the author.

Figure 2 – Simplified NUMA multiprocessor.



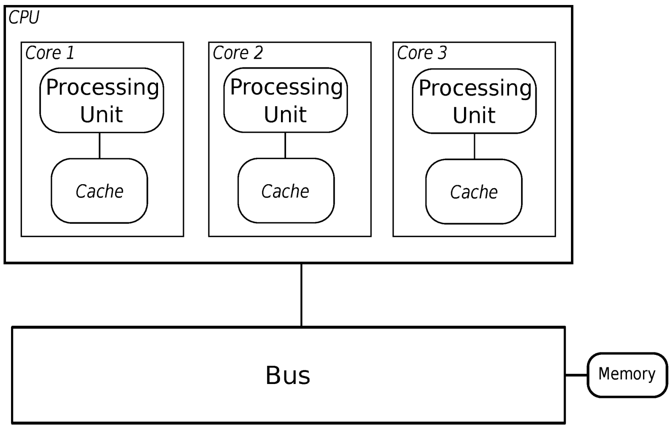
Source: the author.

Figure 2 illustrates the NUMA architecture, where NUMA nodes are connected through an interconnection network. In the same context as UMA, Cache-Coherent Non-Uniform Memory Access (CC-NUMA) is a NUMA multiprocessor with a cache in each node to reduce the access time in a remote memory data. In contrast, a Non Cache-Coherent Non-Uniform Memory Access (NC-NUMA) multiprocessor does not have a cache.

With the development of new technologies, the size of transistors reduced significantly, which enabled a greater quantity of them inside a single chip. As this quantity increased, chips began to consist of several CPU cores defining a new class of chips, named multicores. Figure 3 illustrates this design. In contrast with multiprocessors, a multicore has two or more cores inside a single CPU, which communicate with each other using a shared RAM or the cache directly. Each core consists of a processing unit and a cache.

A multicore processor enables instructions to run on separate cores at the same time, which increases performance. In this context, power consumption became a paramount problem in new systems design, because of excessive sub-threshold leakage current, and supply voltage scaling slowing down (BORKAR, 2007). To solve this challenge, a new

Figure 3 – Simplified multicore processor.



Source: the author.

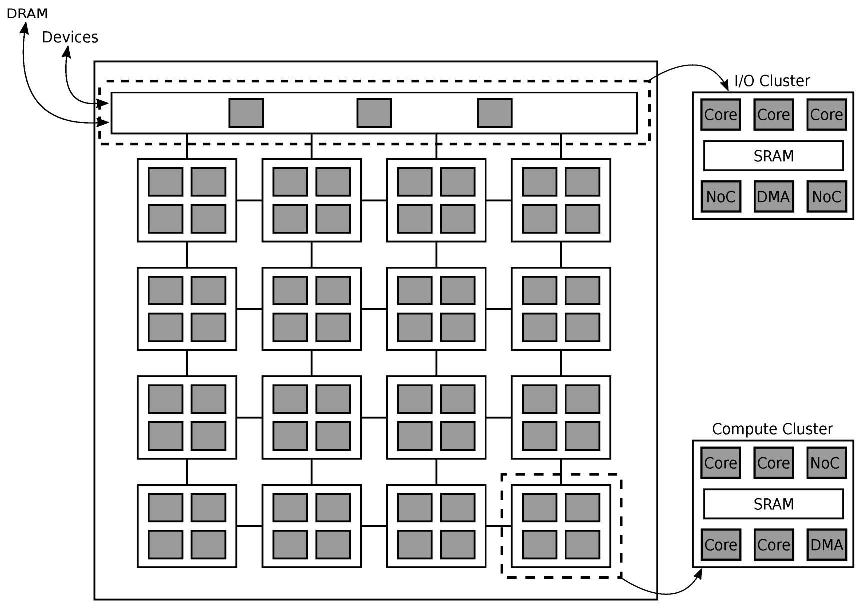
design for processors emerged, named manycore processors. It provides hundreds to thousands of cores in a single chip and bundle several less complex cores, instead of high complex ones. As the search for more power efficiency increased, other solutions became necessary to approach the trade-off between power and performance, such as lightweight manycores.

2.2 LIGHTWEIGHT MANYCORES

Lightweight manycores is a new emerging class of manycores concerned with energy efficiency and performance. Figure 4 illustrates a simplified concept of a lightweight manycore processor. The concept processor features 67 cores bundled in 17 clusters, where cores inside each cluster share resources, such as Static Random Access Memory (SRAM) and NoC interface. In this example, clusters are divided into two groups: (i) Compute Cluster; and (ii) I/O Cluster. Each group has characteristics that are defined by its main goal. Thus, these groups may differ in structure, Input/Output (I/O) connectivity and computing capabilities. In the given example, Compute Clusters present more cores and focus on user workloads. These workloads are stored inside the SRAM that is a few orders of magnitude smaller compared to the external Dynamic Random Access Memory (DRAM). In contrast, I/O Clusters focus on I/O operations with the external DRAM and devices. Finally, inter-cluster communication is performed through a NoC and managed in each cluster by NoC interfaces.

Clusters perform communication with one another sending hardware-level messages through the NoC. For instance, due to Compute Clusters small memory size, I/O Clusters are responsible for providing tiled user workloads to Compute Clusters. Each workload is sent through the NoC and computed by its respective Compute Cluster. After computation, I/O Clusters may merge or perform other operation on the output data from all Compute Clusters. Conversely, communication between Compute Clusters is

Figure 4 – A simplified overview of a conceptual lightweight manycore.



Source: Penna et al. (2021)

made without I/O Cluster intervention. This message passing process can be onerous and impact performance due to its synchronicity. Therefore, to provide a higher bandwidth and improve overall performance, Direct Memory Access (DMA) engines may be provided inside the NoC, which enables asynchronous communication.

Overall, lightweight manycores have several characteristics that provide means to achieve high performance and power efficiency. The comprehensive set of characteristics from lightweight manycores is as follows (PENNA et al., 2021):

(i) they integrate hundreds to thousands of low-power cores in a single chip, which are tightly-coupled in groups called *clusters*;
(ii) they feature a constrained memory system with small local memories inside clusters and no cache coherency;
(iii) they rely on high-bandwidth NoCs to carry out communication;
(iv) they present a distributed memory architecture (i.e., multiple address spaces);
(v) they are designed to target Multiple Instruction Multiple Data (MIMD) workloads; and
(vi) they may have a heterogeneous configuration (e.g., I/O and computation clusters present distinct programmability).

Lightweight manycores differ from other architectures that feature high core counts. In contrast to Graphics Processing Units (GPUs), lightweight manycores present support to MIMD workloads. Unlike NUMA architectures that rely upon dedicated interconnections outside the processor chip to perform communication between multiple CPUs (e.g., NumaConnect or NUMAlink), lightweight manycores rely on a high-bandwidth NoC to carry out communication between thousands of cores. Furthermore, manycore processors, such as Tilera TILE-Gx100, Intel Xeon Phi and Intel Single-Cloud Computer, do not consist of clusters with a constrained memory. Some examples of lightweight manycores are Sunway SW26010 (FU et al., 2016), Adapteva Epiphany (OLOFSSON; NORDSTROM; UL-ABDIN, 2014) and Kalray MPPA-256 (DINECHIN et al., 2013).

Although, the aforementioned characteristics improve performance and power efficiency for lightweight manycores, they also introduce some challenges:

(i) they usually feature *heterogeneous cores* with different computing and communication capabilities, making software development more complex;
(ii) the *memory is distributed* across Compute Clusters and data tiling, data prefetching and remote data accesses have to be performed explicitly by the developer;
(iii) the *constrained on-chip memory* requires several software optimizations to reduce the memory footprint of applications; and
(iv) the *absence of hardware cache coherence* forces developers to explicitly deal with data coherency.

2.3 DISTRIBUTED OPERATING SYSTEMS FOR MANYCORES

Distributed OSs have recently gained attention of the research community to address the challenges in software development in lightweight manycore processors (KLUGE; GERDES; UNGERER, 2014; ASMUSSEN et al., 2016; PENNA et al., 2019). Examples of these OSs are Barrelfish (BAUMANN et al., 2009), Factored Operating System (FOS) (WENTZLAFF; AGARWAL, 2009), Helios (NIGHTINGALE et al., 2009), MOSSCA (KLUGE; GERDES; UNGERER, 2014), M^3 (ASMUSSEN et al., 2016) and Nanvix (PENNA et al., 2019). They detach complex OS services from the OS kernel, which commonly provides bare-minimum OS abstractions and primitives. Thus, OS services and application instances are spread out on processor cores to increase scalability and performance.

These OSs are commonly structured in three main layers. In the bottom layer, a Hardware Abstraction Layer (HAL) abstracts the underlying hardware, to provide portability across different architectures. In the middle layer, an OS kernel provides basic OS abstractions, such as resource multiplexing, processes and threads. Finally, the top layer features many OS libraries to provide a transparent programming environment for users.

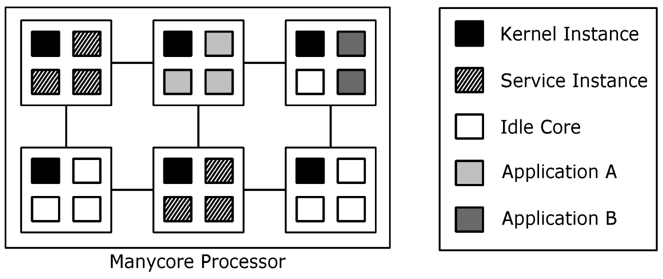
In this dissertation, we are specially interested in distributed OSs designed specifically for lightweight manycores such as MOSSCA (KLUGE; GERDES; UNGERER, 2014), M^3 (ASMUSSEN et al., 2016) and Nanvix (PENNA et al., 2019). Due to scarce resources inside lightweight manycores, a microkernel design is employed in the middle layer to cope with the low amount of on-chip memory. Then, system services are implemented at the top layer and are distributed across the cores of the processor. This approach provides more availability for user applications. Among the aforementioned OSs for lightweight manycores, MOSSCA and M^3 can only be executed on an in-house simulator and a processor prototype implemented in FPGA, respectively. In contrast, Nanvix is currently the only open-source distributed OS that runs on commercially available bare-metal lightweight manycores. Because of this feature, we will focus on Nanvix in this dissertation.

2.3.1 Nanvix

Nanvix is an open-source OS for lightweight manycores based on a multikernel design (PENNA et al., 2019). Figure 5 depicts the main idea behind this design. As it can be noticed, there is a single Nanvix microkernel instance running in each Compute Cluster (*kernel instance*). Each instance is deployed only in one core of the Compute Cluster, configuring an asymmetric microkernel design. Moreover, it provides bare-minimum OS abstractions and primitives for the Compute Cluster. All other cores of the Compute Cluster, on the other hand, can be used to run system services as servers (*service instance*) or user-level applications. Examples of system services implemented in Nanvix are: (i) *Shared Memory (SHM) Server*, which manages page sharing among different Compute Clusters; (ii) *Remote Memory (RMem) Server*, which manages physical memory; (iii) *Name Server*, which manages process naming; (iv) *Spawn Server*, which manages threads; (v) *Virtual File System (VFS) Server*, which manages secondary storage; and (vi) *Inter-Process Communication (IPC) Server*, which manages communications between processes running on different Compute Clusters.

Distributed OSs for lightweight manycores is an ever-increasing field, which has several features that can be evaluated and optimized. For instance, Nanvix remote memory service enables applications to communicate and to share pages effectively. The Distributed Paging System (DPS) is essential for a distributed OS because it enables memory virtualization and brings better memory usage and optimizations. However, extensive use of page sharing among clusters can result in suboptimal memory access performance. In this context, a software-managed cache would bring important improvements to overcome this barrier and optimize the overall performance. Unfortunately, it is not trivial to design and evaluate new features in a distributed OS. In the next section, we discuss possible approaches to do so.

Figure 5 – A simplified overview of a conceptual lightweight manycore running a multikernel OS.



Source: the author

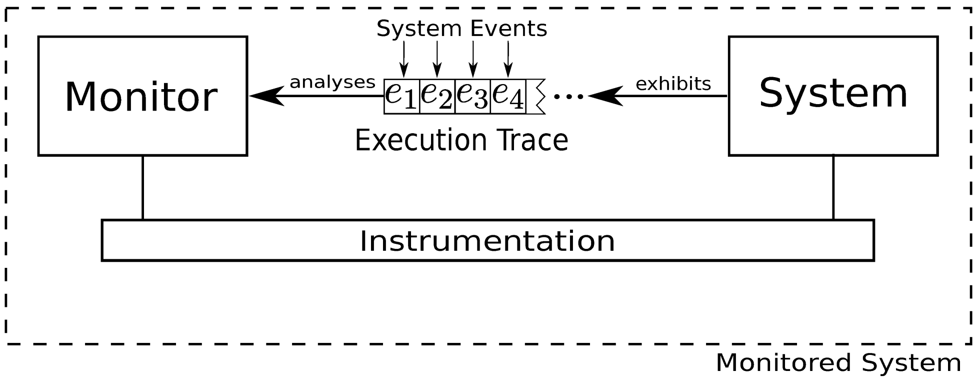
2.4 RUNTIME VERIFICATION

Common verification analysis techniques used by developers to evaluate a system are model checking, theorem proving and Runtime Verification (RV) (JAHIC et al., 2018). In contrast to the others, RV is widely used to evaluate a system during runtime based on its execution trace. Figure 6 illustrates how RV analysis is applied in a system. Overall, the monitored system has three main components:

- Target system that is being analyzed;
- instrumentation method; and
- system monitor.

First, the target system being analyzed by the monitor is one of the main components of a RV analysis. Next, the instrumentation component provides means to record relevant information about the target system, which is used by the monitor to analyze the overall system behavior. Finally, the system monitor analyzes the target system behavior and may declare if a property or characteristic of the system is being satisfied (BARTOCCI et al., 2018).

Figure 6 – Monitored system overview.



Source: Bartocci et al. (2018)

In this context, several features from a system may be evaluated. For instance, in OSs, a software developer may use this technique to evaluate the memory system behavior based on memory events inside an execution trace. In a broad manner, these events can be used as input to a memory system feature, such as a software-managed cache, to check and evaluate its behavior without running the system repeatedly. Furthermore, this approach brings a means to evaluate an already designed feature, but does not provide a way for design choices and possible optimizations of new features. In the remaining of this section, we will discuss in more detail how the monitor and instrumentation components are connected with the target system analysis.

2.4.1 Monitors

The monitor component executes in parallel with the target system and evaluates its execution. Overall, this component checks if a property was satisfied or not by the target system. After enough behaviors are verified, the monitor arrives at a verdict (acceptance or rejection) about the target system. Furthermore, an automated synthesis procedure is used to generate monitors that can verify a set of properties. The procedure takes the properties as input and returns the executable code of the monitor as a result (BARTOCCI et al., 2018).

Monitor setups come in several flavors that can be explored by the developer. Online and offline monitoring dictate where the evaluation will be made during or after system execution, respectively. An online approach can determine during execution if a property is satisfied, which enables the monitor to deal with property violations at runtime. In contrast, an offline approach stores, after the target system execution, its trace inside a permanent file, which is passed to the monitor for evaluation.

Furthermore, online monitoring has to dictate how the monitor and the target system may execute simultaneously. In a synchronized manner, the monitor waits for the target system to generate events. With each event, the target system continues executing after monitor evaluation. In contrast, an asynchronous approach provides the target system an uninterrupted execution as the monitor evaluates events. Due to the unpredictable behavior of offline and online asynchronous monitoring, the monitor may miss opportunities to deal with unsatisfied properties as they occur.

These aforementioned monitor flavors can be synthesized by two main approaches. A monolithic approach is the most common one. The monitor is synthesized as one monolithic block to represent a specification of properties. In contrast, monitors may follow a parallel approach to provide a better performance, where each monitor is responsible for a set of properties. In this context, a concurrent and distributed layout may be used. A distributed layout can be coordinated in two different ways. An orchestrated strategy promotes an entity to coordinate all events, whereas a choreographed approach promotes more than one entity to it.

2.4.2 Instrumentation

As the target system executes, the instrumentation component will record relevant system information, such as program variable write/read, memory operations, communication operations, function calls and returns. Each system information is stored as events inside an execution trace. Thus, an execution trace is composed by an ordered stream of events that are provided to the system monitor. Moreover, this component dictates how monitor and target system relate with each other. For instance, to better evaluate the system, the execution needs to terminate before the monitor system begins to evaluate the execution trace. In a more comprehensive manner, instrumentation may be used individually for software developers to evaluate manually applications or a whole system. To achieve this goal, there are widely known instrumentation software such as Pin (LUK et al., 2005) and Valgrind (NETHERCOTE; SEWARD, 2003).

2.4.2.1 Pin

Pin is an instrumentation platform that enables the build of several analysis tools (*pintools*), which are used to evaluate applications and the system. The platform is an engine that has a just-in-time (JIT) compiler and instruments applications, whereas the *pintool* communicates with Pin, and has the instrumentation and analyzes routines. Moreover, the *pintool* evaluation gives important information about memory, function calls and other resources. For instance, a developer may need to evaluate the memory system. Hence, function calls can be added before and after a memory operation. With these calls, the tool captures memory addresses, allocation size and other important information. In addition, the tool may manipulate each collected information to a more readable one and add them inside a file for further manually evaluation.

This instrumentation platform uses a JIT compiler to insert and optimize code without the need for the developer to intervene. In this context, the *pintool* initializes and requests Pin to execute the application. Pin intercepts the code execution on the first instruction and compiles new code to take its place. This code compilation is performed one straight line of code at a time, which is defined by a sequence of instructions until a branch exits it. The compiled code is similar to the original but has given instrumentations by the *pintool* and is stored inside a software-managed code cache. Also, only code inside the code cache is executed instead of the original. After a compiled code is executed, Pin regains control, proceeds to generate and execute more code until the end of the application execution.

2.4.2.2 Valgrind

Pin was designed to have low impact on overall performance. In contrast, Valgrind (NETHERCOTE; SEWARD, 2003) was designed to provide support to rich analysis tools with performance as a secondary concern. Mainly, Valgrind is an instrumentation platform that is capable of building several analysis tools and has two main components: (i) core; and (ii) skin. The core component provides four base features for Valgrind. First, an JIT compiler which enables Valgrind to run a simulated code inside a synthetic CPU and maintain program control. It does not run the original code directly. Second, a C library, because the platform does not support GNU standard C library (glibc). This decision was made to avoid possible bugs and obscure dependencies from the standard library (NETHERCOTE; SEWARD, 2003). Third, a signal interception feature that acts as an intermediary handler to keep the program under Valgrind control. Finally, a Pthread implementation which provides all standard pthread abstractions and a user-space thread package. All threads run on a single kernel thread while keeping the program under the platform control. Furthermore, Valgrind works directly with executables removing the need to recompile the program. In contrast, the skin component is written by the user and defines how the program will be instrumented. Moreover, this component must provide specific functions for initialization, instrumentation and finalization, which are not provided by the core. There are eight widely used standard skins: (i) Memcheck; (ii) Cachegrind; (iii) Callgrind; (iv) Helgrind; (v) DRD; (vi) Massif; (vii) DHAT; and (viii) BBV.

2.5 CACHES

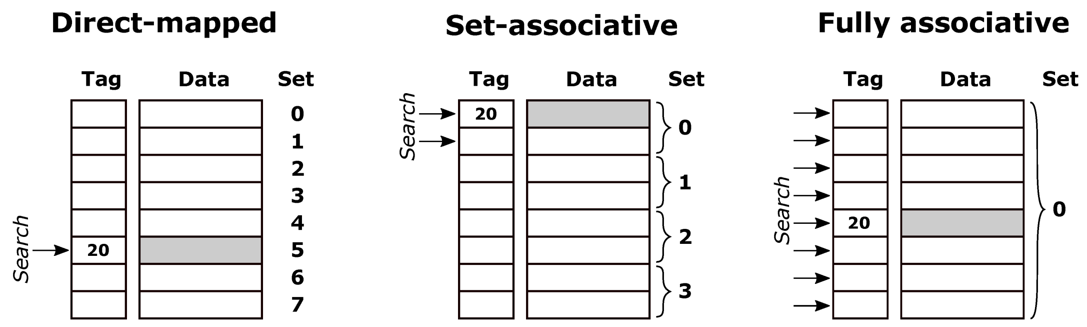
Caching is a technique that improves performance on data access by maintaining frequently used data in a memory module that features lower latency than other memory modules in the architecture. This technique may be implemented either in hardware or software and is employed in a wide range of scenarios, varying from processor design to web applications. Notwithstanding, the structure and policies concerning this technique hold in all these contexts, thus in the paragraphs that follow we cover its common background.

The cache memory is typically smaller than the underlying memory, thus a policy is required to state how data from a lower-level memory is mapped to the cache. This decision concerns the placement policy of the data blocks onto the cache and some possible designs are pictured in Figure 7. In a direct-mapped cache, blocks of memory are directly mapped to a given slot in the cache. Since each block may be placed in only one slot in the cache, searching takes a constant time, but different blocks may be mapped to the same slot, thus causing conflicts. In a set-associative cache, the amount of conflicts is mitigated by enabling a block to be mapped on a set of slots in the cache. As a consequence, a search procedure is needed in all slots within the set to access data. Finally, in a fully associative cache, a memory block can be mapped to any slot in the cache, thus eliminating placement conflicts at a cost of increasing hardware complexity.

Whenever a memory block should be brought to the cache and no placement slot is available, some block in the cache should be evicted. Such decision concerns the replacement policy of the cache and multiple algorithms for this task exist, each one with its own strengths and weaknesses.

In First-in First-out (FIFO) replacement policy, the first block brought to the cache is chosen to be evicted. While this policy is relatively simple to implement it may present some performance anomalies if the very first blocks brought to the cache end up being the more frequently used ones. The Second Chance (SC) policy attempts to overcome this problem by evicting FIFO blocks that were not accessed after the last replacement walk. The Not Frequently Used (NFU) policy maintains a reference counter

Figure 7 – Cache placement techniques.



Source: Adapted from Penna, Francis & Souto (2019).

for each memory block in the cache and, at each clock interval, all memory blocks that have been referenced within that interval will have their counter incremented by 1. Thus, the memory block with the lowest counter is evicted whenever necessary. The main problem with NFU is that it keeps track of the frequency of use without regard to the time span of use, so memory blocks that have been referenced several times in a distant past will be kept in the cache for a long period of time even though they are not referenced anymore. The Aging replacement policy is a descendant of NFU, with modifications to make it aware of the time span of use. In this policy, the reference counter is first shifted right and then, the reference bit replaces the most significant bit of the reference counter. This ensures that memory blocks referenced more recently will have higher counters. Conversely, the Least Recently Used (LRU) policy maintains a wallclock register for each memory block in the cache. The register of a block is incremented with the current time whenever a reference to this block occurs. Then, whenever a replacement decision should be made, the least recently used block is evicted. This policy oftentimes produces optimal replacement decisions but features a great implementation overhead due to the linear cost of updating the wallclock registers at each memory reference. Pseudo-LRU algorithms solve this problem by using approximate measures for each wallclock register instead of its exact value inside the cache.

Furthermore, whenever data is written into a memory block in the cache, it should be written back to underlying memory later on. The decision on when this task should happen concerns the *write policy* and overall there are two alternatives. In the first one, called *write-though*, the underlying memory is updated whenever a write occurs in the cached memory block. This policy ensures strong data consistency between the memory cache and the underlying memory in exchange for more costly write operations. The second policy, named *write-back*, relaxes this requirement by writing data back from time to time, either whenever a block replacement happens or at frequent time intervals.

As it can be noticed, a cache has several factors that need to be taken into consideration before a prototype can be produced. Therefore, simulations and runtime verification solutions can be introduced to discover the best design choice and analyze cache behavior. The former enables cache evaluation without the need to prototype every single choice, whereas the latter provides a way to verify memory accesses and, for example, use them as input in a software-managed cache to evaluate several choices. The software-managed cache inherits these core concepts of a cache and may be used inside a distributed OS to provide a means for optimizations.

3 TRACE-DRIVEN METHODOLOGY

Providing optimizations for the memory management service of distributed OSs that target lightweight manycores is an open problem. In this dissertation, we propose a trace-driven methodology to aid developers to study, implement and optimize features for memory management services of distributed OSs. Our trace-driven methodology simplifies the trial-and-error approach of finding the proper configurations for different features in the memory management service based on the characteristics of the applications as well as the memory and NoC restrictions of lightweight manycores, relieving distributed OSs developers from the burden of porting several applications to the target distributed OS.

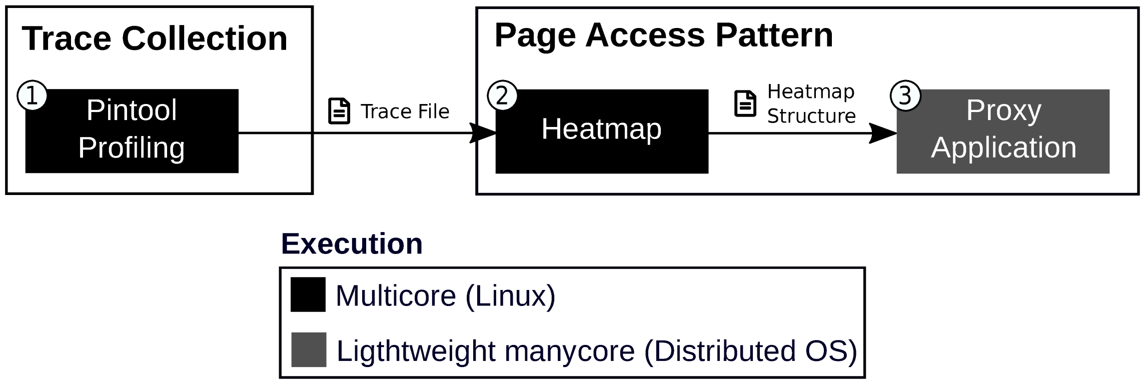
In this chapter, we describe the trace-driven methodology proposed in this dissertation. First, we present an overview of the methodology, which is composed of two main stages: *trace collection* and *page access pattern*. Then, we discuss the details of each of these stages. Finally, we argue about specific implementation details of our methodology.

3.1 OVERVIEW

Our trace-driven methodology allows one to mimic the memory access pattern of applications in distributed OSs for lightweight manycores without the need to port them to the target distributed OSs. The main goal is to provide a means to study and implement different optimization features for the memory management service in the context of distributed OSs for lightweight manycores.

Figure 8 presents an overview of our methodology. Overall, the methodology is composed of two main stages, each composed of one and two steps, respectively. 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). Next, one selected output is used to create a *heatmap*: a compact two-dimensional structure that describes how many times

Figure 8 – Overview of the trace-driven methodology.



Source: the author.

Listing 1 – An example of a simple C program that allocates three arrays in three different memory segments (each of which occupying 20 memory pages) and performs a variable number of store operations in each page.

```c
1  #define N_ITEMS 20
2  struct item {
3    uint32_t _[1023]; // padding
4    uint32_t value;
5  };
6
7  /* Static Data allocation of N_ITEMS pages */
8  static struct item static_v[N_ITEMS];
9
10 int main(void) {
11   /* Heap allocation of N_ITEMS pages */
12   struct item *heap_v = (struct item *) malloc (sizeof(struct item)*N_ITEMS);
13
14   /* Stack allocation of N_ITEMS pages */
15   struct item stack_v[N_ITEMS];
16
17   for (int i = 0; i < N_ITEMS; i++)
18     for (int j = 0; j < (i+1)*100; j++) {
19       static_v[i].value = i;
20       heap_v[i].value = i;
21       stack_v[i].value = i;
22   }
23
24   free(heap_v);
25
26   return 0;
27 }
```

Source: the author.

each memory page (or a set of pages) was accessed during some discrete time frames (Step ②). 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 ③). Finally, the *proxy application* outputs statistics about the execution.

To better understand the proposed methodology, let us consider the simple C program shown in Listing 1. We assume in this example a Linux x86_64 system, where an integer variable and a memory page occupy 4 bytes and 4 kB in memory, respectively. This program allocates three arrays (`static_v`, `heap_v` and `stack_v`) containing 20 elements of type `struct item` (lines 8, 12 and 15). Since the size of `struct item` is 4 kB, each item is stored in a different memory page. Therefore, 60 memory pages will be allocated to store these arrays. Then, the program performs a variable number of store operations on the `value` field of `static_v[i]`, `heap_v[i]` and `stack_v[i]` (lines 17–22). This is done by two `for` loops: the outermost loop (line 17) iterates over the items of the arrays whereas the innermost loop (line 18) performs repeated store operations on the `value` field of `struct item`. Since the number of store operations on each array index `i` is equal to `(i+1)*100`, the higher the value of `i`, the higher will be the number of store operations at index `i`. As a result, the number of store operations at indexes $i = \{0, 1, 2, \ldots, 19\}$ of each array will be $\{100, 200, 300, \ldots, 2000\}$.

In the next sections, we describe in more details the *trace collection* and *page access pattern* stages using this program as an example.

3.2 TRACE COLLECTION

The trace collection stage is composed of one main step, which is described bellow.

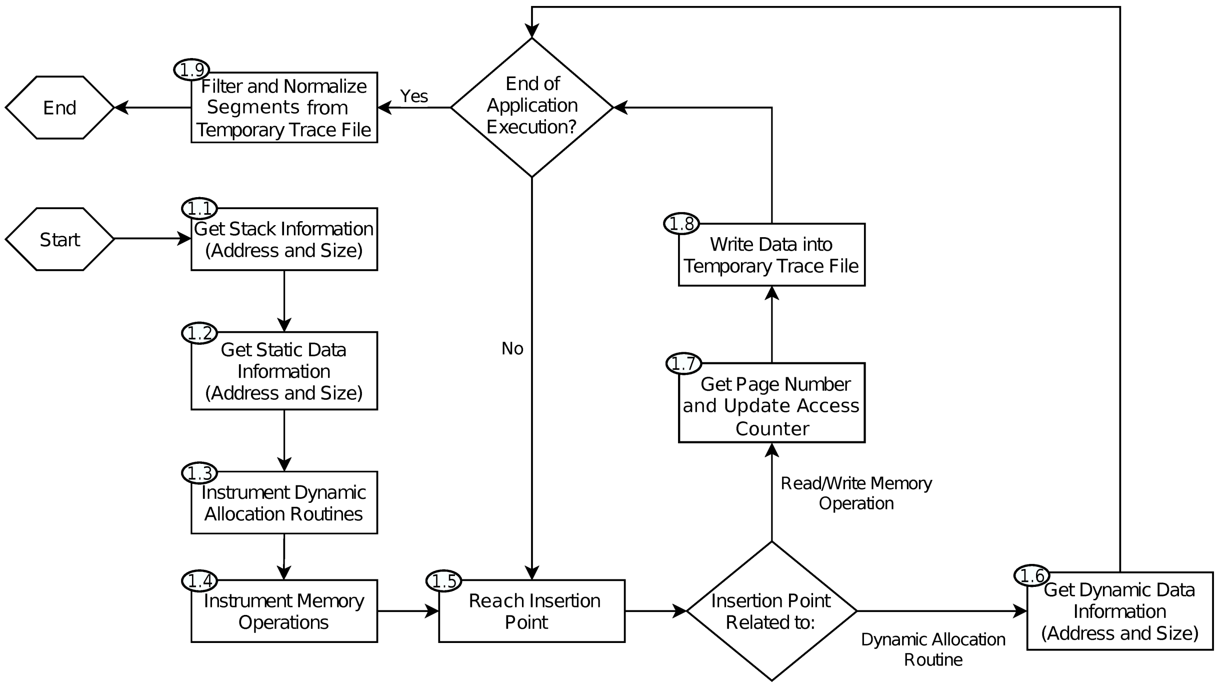
3.2.1 Step 1: *Pintool* Profiling

The first step of our trace-driven methodology consists in running the application of interest on Linux and collecting information about its memory accesses (Step ①). A classical approach to obtain such information without any application source code changes is to use a *binary instrumentation tool* such as *Valgrind* (NETHERCOTE; SEWARD, 2007) or Pin (LUK et al., 2005). Valgrind features a tool named *Lackey*, which outputs the address of every memory access made by the program. The drawback of Lackey is that it only distinguishes instruction loads from memory accesses, thus needing further manipulations on the trace file to get the memory accesses in each memory segment (*static data*, *dynamic data* and *stack*). Pin, on the other hand, is much more flexible, but it requires specific knowledge from the developer to build a *pintool*[1]. In this dissertation, we decided to use Pin due to its flexibility and disregard the instruction memory segment.

Figure 9 shows an overview of the pintool designed to collect the memory accesses of an application and to produce the corresponding trace files. First, the pintool discovers the initial memory address and size of two memory segments: *stack* and *static data* (Activities (1.1) and (1.2), respectively). The initial address and the maximum size of the stack can be easily obtained with the `PIN_GetContextReg()` function available in Pin and the `getrlimit()` Linux system call, respectively. In contrast, there is not a straightforward way of obtaining the initial address and size of the static data segment. The solution adopted in our pintool is to use a standard Linux utility program named `objdump` to disassemble all the object file sections and get information about the *bss* and *data* segments. As opposed to the aforementioned segments, information about the dynamic data segment (heap) is obtained in runtime. To do so, the pintool instruments dynamic allocation routines (e.g., `malloc`, `calloc`, `realloc` and `align_alloc`) and places an insertion point on each routine occurrence (Activity (1.3)). Furthermore, the pintool instruments instructions that read or write memory (Activity (1.4)), thus adding an insertion point

[1] A *pintool* can be thought of as a plugin that can modify the code generation process inside Pin.

Figure 9 – Pintool flowchart.


1.9 Filter and Normalize Segments from Temporary Trace File
End
Yes
End of Application Execution?
Start
1.1 Get Stack Information (Address and Size)
1.8 Write Data into Temporary Trace File
1.2 Get Static Data Information (Address and Size)
No
1.7 Get Page Number and Update Access Counter
1.3 Instrument Dynamic Allocation Routines
Read/Write Memory Operation
1.4 Instrument Memory Operations
1.5 Reach Insertion Point
Insertion Point Related to:
Dynamic Allocation Routine
1.6 Get Dynamic Data Information (Address and Size)

Source: the author.

to each of them. Each insertion point tells Pin where to insert the analysis routine call defined by our pintool.

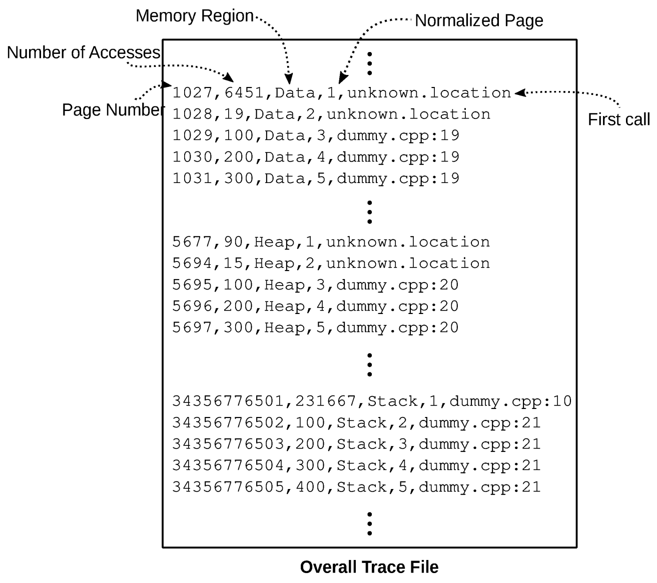
When an insertion point is reached in runtime (Activity (1.5)), the pintool checks if it concerns a dynamic allocation routine call instrumented in Activity (1.3) or a simple instruction that read or write memory. If the insertion point is related to a dynamic allocation call, our analysis routine collects the initial address of the allocated memory and its size (Activity (1.6)). This information is stored inside a hash table structure, which will be essential for Activity (1.9). However, if the insertion point is related to instructions that read or write memory, the analysis routine performs the following tasks: (i) it converts the memory address to a page number and increments the access counter for that page number in a hash table structure (Activity (1.7)); (ii) it writes the page number along with the current value of a *memory access event counter*[2] (initially set to 1) into a temporary trace file (Activity (1.8)); and (iii) it increments the value of the memory access event counter. Once the aforementioned operations are finished, the pintool continues to execute the instructions of the application until it reaches another insertion point. When the application finishes, the pintool outputs a *memory segments overview file* and *three trace files* (Activity (1.9)).

The memory segments overview file contains information about each memory page allocated by the application. An example is shown in Figure 10(a) for the program given in Listing 1. In this file, the following information is provided for each page: its virtual page number, the total number of accesses to that page, the memory segment

[2] A memory access event count is actually considered as a discrete *timestamp*.

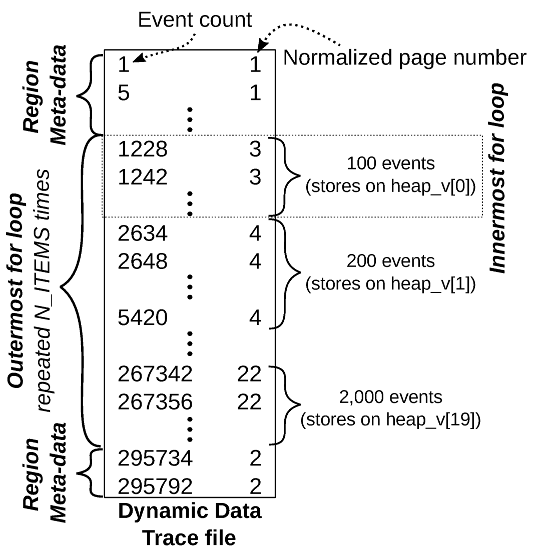
Figure 10 – Memory segments overview file and dynamic data segment trace file obtained from the application presented in Listing 1.

(a) Memory segments overview file.



```
⋮
1027,6451,Data,1,unknown.location
1028,19,Data,2,unknown.location
1029,100,Data,3,dummy.cpp:19
1030,200,Data,4,dummy.cpp:19
1031,300,Data,5,dummy.cpp:19
⋮
5677,90,Heap,1,unknown.location
5694,15,Heap,2,unknown.location
5695,100,Heap,3,dummy.cpp:20
5696,200,Heap,4,dummy.cpp:20
5697,300,Heap,5,dummy.cpp:20
⋮
34356776501,231667,Stack,1,dummy.cpp:10
34356776502,100,Stack,2,dummy.cpp:21
34356776503,200,Stack,3,dummy.cpp:21
34356776504,300,Stack,4,dummy.cpp:21
34356776505,400,Stack,5,dummy.cpp:21
⋮
```

(b) Dynamic data segment (heap) trace file.



```
1       1
5       1
   ⋮
1228    3
1242    3
   ⋮
2634    4
2648    4
   ⋮
5420    4
   ⋮
267342  22
267356  22
   ⋮
295734  2
295792  2
```

Source: the author.

where the page resides, its normalized page number and the location in the source code where the page was accessed for the first time. The normalized page number within its memory segment is obtained by storing all page numbers inside a hash table structure and then assign a consecutive integer number (starting at 1) as value to each of them.

Overall, the memory segments overview file (Figure 10(a)) gives a global view of the memory accesses carried out by the application and helps the user decide which memory segment should be considered in the following steps of the trace-driven methodology. As expected, the number of accesses to pages that hold the three arrays allocated by the program shown in Listing 1 (`static_v`, `heap_v` and `stack_v`) correspond to those write operations executed by the nested loops of the program (Listing 1, lines 17–18). We can also observe that there are few other pages in each memory segment that are also accessed during the execution. The data allocated in these pages correspond to local variables such as loop iterators (stack segment), global or static variables allocated in shared libraries (static data segment) and meta-data managed by the built-in allocator provided by the GNU C Library (dynamic data segment).

In addition to the aforementioned output file, the pintool splits the temporary trace file produced in Activity (1.8) into three trace files, each one containing only the trace events related to a specific memory segment (*static data*, *dynamic data* or *stack*). Figure 10(b) shows some events stored in the dynamic data trace file obtained from the program given in Listing 1. As it can be noticed, all event counts are normalized in this trace file (the first event count in the dynamic data segment is set to 1 and the following ones are normalized with respect to it). The same procedure is done to produce the other two trace files (*static data* and *stack*). We can also observe that two pages holding meta-data of the standard dynamic memory allocator (normalized page numbers 1 and 2) are accessed at the beginning and at the end of the trace file. As expected, a repeated number of the accesses are done in pages that hold the `heap_v` array. These accesses are based on the outer and innermost loops of the program shown in Listing 1.

Considering the program given in Listing 1, our pintool produced trace files of approximately 202 kB (static data), 194 kB (dynamic data) and 2.3 MB (stack). However, these trace files can easily occupy gigabytes of data if more complex programs are profiled, since they include every memory access done by the program. In the next section, we show how we can represent the overall page access pattern of an application in a much more compact way by using heatmaps.

3.3 PAGE ACCESS PATTERN STAGE

The page access pattern stage is composed of two main steps, which are described below.

3.3.1 Step 2: Heatmap

The trace files obtained from the previous step are used in the *page access pattern stage* to produce a *heatmap* structure (Step ②). The *heatmap* is a 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 (SUEMATSU et al., 2014). 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 (or 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 group 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.

Figure 11 – Heatmap of the *dynamic data segment* of the program shown in Listing 1.

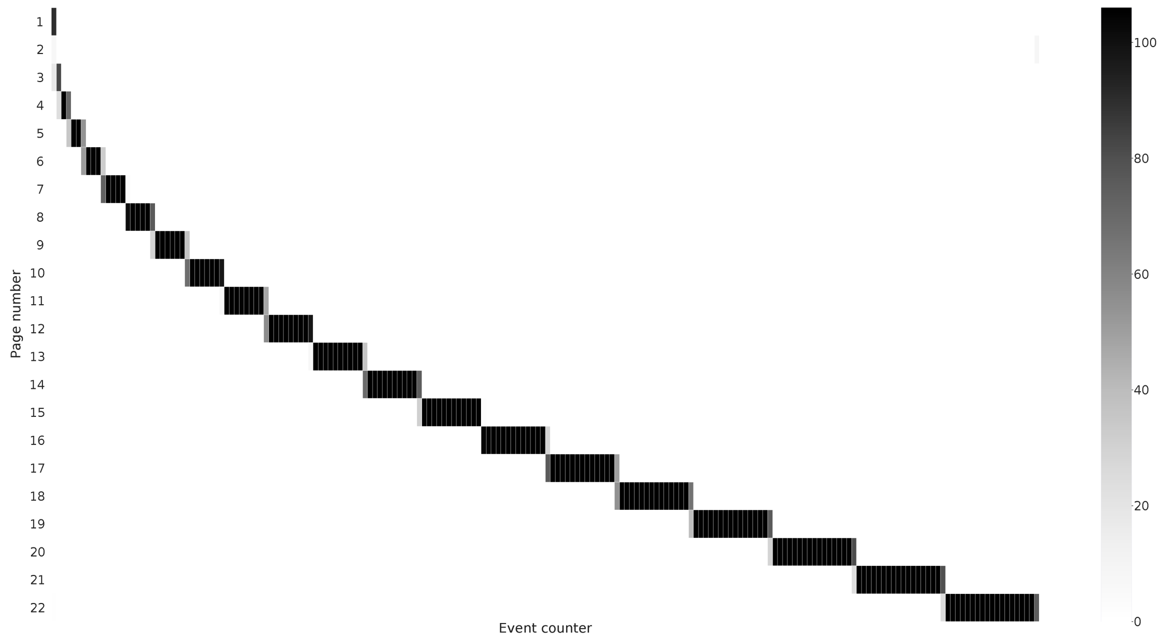


Figure 11 shows the heatmap containing page accesses to the dynamic data segment in which the array `heap_v` of the program shown in Listing 1 resides. The heatmap shows 22 memory pages[3] in the y-axis and 200 discrete time *bins* in the x-axis.[4] As it can be noticed, the heatmap shows exactly the behavior we would expect based on the program shown in Listing 1: memory pages occupied by `heap_v` are accessed in ascending order and the number of page accesses increases as the execution progresses.

3.3.2 Step 3: Proxy Application

The last step concerns the use of a *proxy application* to mimic the page access pattern of the original application on the target lightweight manycore processor (Step ③). The proxy application takes as input parameters a *heatmap* (*hm*), which was obtained in Step ②, 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 memory page accesses) and it uses a probabilistic approach that follows a uniform function 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

$$trials_x = \frac{\text{number of events in the original } heatmap \text{ in } x}{tf} \tag{3.1}$$

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

[3] The `heap_v` array is stored in pages 3–22.
[4] All x-axis labels were omitted to improve readability.

Algorithm 1 – Overview of the proxy application.

Input: A heatmap *hm* containing the page access pattern of the profiled application and a trial factor *tf* to be considered in each time *bin*.
Output: A trace file containing all page accesses generated by the proxy application as well as other statistics.

```
1: procedure MimicPageAccessPattern($hm, tf$)
2:     for $bin \leftarrow 1$ to $hm.nybins$ do
3:         $pages[bin] \leftarrow$ PageAlloc()        ▷ Request a new page and keep its reference
4:     for $bin \leftarrow 1$ to $hm.nxbins$ do                          ▷ For all time bins
5:         $ntrials \leftarrow \frac{hm.nevents[bin]}{tf}$              ▷ Compute the number of trials
6:         for $trial \leftarrow 1$ to $ntrials$ do                  ▷ For all page access trials
7:             $rand \leftarrow$ GetRandomPage($hm.histogram[bin]$)  ▷ Select a random page
8:             $stat \leftarrow$ PageRequest($pages[rand]$)           ▷ Request selected page
9:             Output($bin, trial, rand, stat$)                ▷ Write event and statistics
10:    for $bin \leftarrow 1$ to $hm.nybins$ do                       ▷ Free all allocated pages
11:        PageFree($pages[bin]$)
```

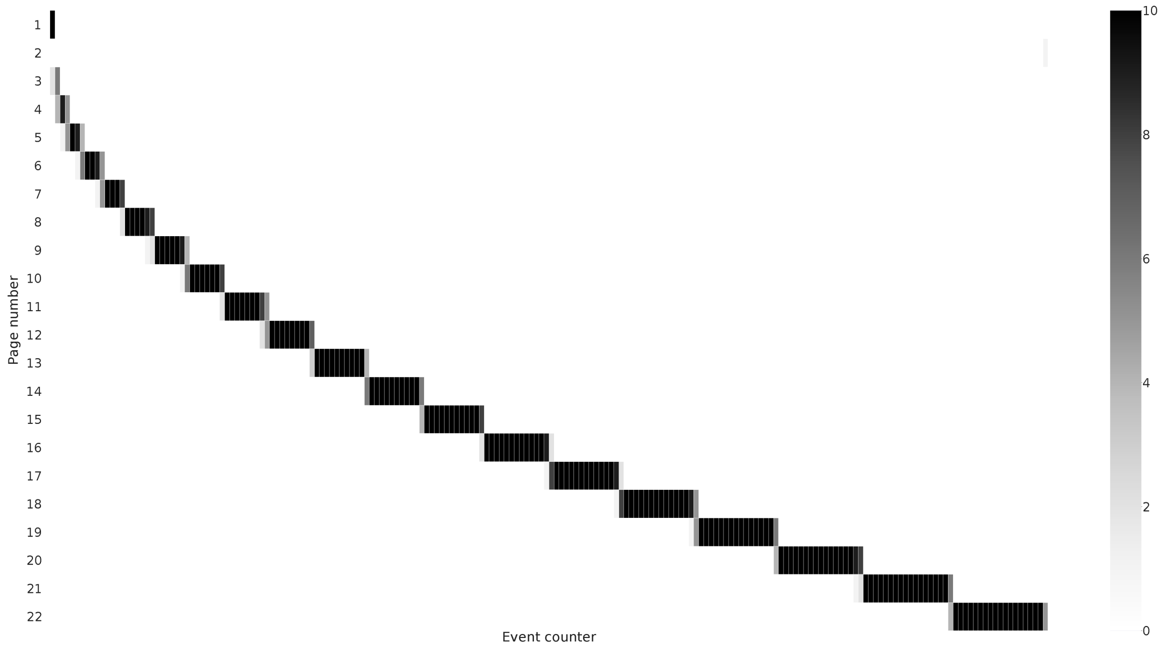
Source: the author.

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 (real) application. However, as we will show later on in Chapter 4, it is possible to reduce considerably the number of trials without losing the overall behavior of the original page access pattern. This allows us to reduce considerably the execution time of the *proxy application* as well as to deal with very large *heatmaps* more efficiently.

Algorithm 1 shows the pseudocode of our proxy application. First, the number of pages is obtained from the *heatmap* and the necessary pages are allocated (lines 2–3) by invoking the OS system call to allocate a memory page (PageAlloc). Each allocation will return a page number from the OS. Then, *ntrials* page access requests are performed (PageRequest) for each time *bin* in the *heatmap* (lines 4–9). For each page access request, a page number is randomly chosen based on the histogram of the time *bin*. All page accesses generated by the proxy application are then stored in an output file (line 9). At this point, a *heatmap* can be generated from the output of the proxy application and it can be compared with the original *heatmap* for validation purposes. Finally, all pages allocated by the proxy application are freed (lines 10–11) by calling the OS system call to free a memory page (PageFree).

Figure 12 shows the *heatmap* produced by the proxy application. In this example, we kept the same 200 discrete time *bins* in the x-axis but we reduced the number of total page accesses by a factor of 10 in each time *bin*. As shown in Figure 12, the overall access pattern is kept, proving that the probabilistic approach used in the proxy application is valid, even with a reduced number of trials per time *bin*. An experimental validation of the proposed approach with more complex applications will be presented in Chapter 4.

Figure 12 – Heatmap of the *dynamic data segment* obtained from the proxy application.



3.4 IMPLEMENTATION DETAILS

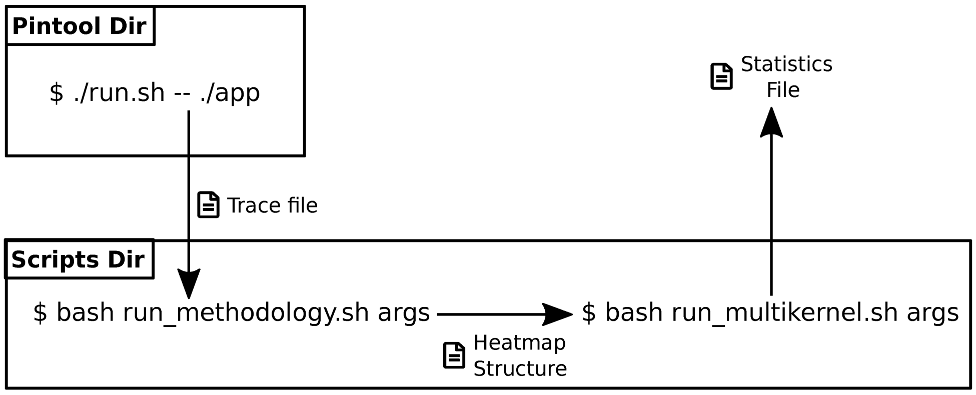
As discussed in the previous sections, our methodology adopts Pin to trace the target application and to produce trace files (Step ①). In contrast, the *heatmap* and the integration with the target distributed OS are done with *bash* scripts. In the following paragraphs we give more details about the scripts and workflow of the proposed methodology.

Figure 13 shows all scripts involved in the production of all the artifacts of our methodology. The first step is to compile the target application using `gcc` with the following flags:

- `-g`: the debugging option will provide means for our pintool to access the proper location of each memory access in the source files; and
- `-no-pie -fnopic`: these flags disable the generation of position-independent executable and code, respectively. Hence, address space randomization is disabled and our code will be loaded at a specific memory location. This characteristic will enable our pintool to compare static data segment addresses with addresses obtained from `objdump` sections.

Once the target application is compiled, the developer shall execute our main script (`run.sh`), which takes the target application (`app`) as input argument as follows: `./run.sh -- ./app`. The provided `./run.sh` script will compile and run our pintool along with the target application. Once the execution finishes, the pintool will output three trace files, one for each memory segment, and the developer shall select one of them to build the *heatmap*.

Figure 13 – Scripts of the trace-driven methodology and their interactions.



Source: the author.

A *bash* script named `run_methodology.sh` is provided to build the *heatmap* from the selected trace file. This script takes as input arguments the trace file of a memory segment, the *heatmap* resolution (number of *bins* in x and y axes) and *heatmap* aesthetics (e.g., image size and font). It calls a Python script that executes the following actions to build the *heatmap*. First, it reads the trace file and creates a `DataFrame` (`df`) using `pandas` library from Python. This `df` consists of a series entries, each one representing a page access event and having an event count and a page number associated to it. Then, a *groupby* operation is performed in the original `df` to produce a new `df` that stores the number of page accesses for each group of pages (y-axis) at each discrete time interval (x-axis). This is exactly the information needed to produce the *heatmap* data structure as well as its graphical visualization.

The proxy application implements the algorithm described in Section 3.3.2 (Algorithm 1) in C programming language. We integrated this application into Nanvix, and it is compiled along with the OS kernel. We opted to export the *heatmap* structures produced by the `run_methodology.sh` to simple C header files that can be included by the proxy application. Each C header file contains all information needed by the proxy application to reproduce the memory access pattern of the original application.

Finally, the `run_multikernel.sh` is responsible for executing the proxy application in Nanvix on the target lightweight manycore processor (Kalray MPPA-256). It takes as input parameters the *heatmap* structure and a trial factor (*tf*) to be used by the proxy application to reproduce the memory access pattern of the original application. Since this script is OS- and platform-dependent, it must be modified in order to carry out experiments with other distributed OSs and/or lightweight manycore processors.

When the execution finishes, the proxy application outputs a file containing all page access events generated during the execution. This file has the same format as the original trace file, so it can be used to produce a *heatmap* and its graphical visualization. Thus, it is possible to compare the *heatmap* produced by the proxy application with the *heatmap* produced from the original trace file. This allows the developer to check if these *heatmaps* are similar or not (i.e., if the proxy application produced a page access pattern similar to the original application). In addition, the proxy application may output other statistics that are specific to a memory management service under study. In Chapter 4, we discuss one possible case study where our trace-driven methodology can be applied as well as other statistics and metrics considered for this case.

4 EXPERIMENTS

In this chapter, we present the experiments performed with our methodology. First, Section 4.1 presents the experimental methodology used in this work. Then, Section 4.2 shows the obtained results.

4.1 METHODOLOGY

In this dissertation, we chose the CAP Bench (SOUZA et al., 2010) to study the effectiveness of our methodology. The benchmark suite is an effort on providing a benchmark suite for both multicores and lightweight manycores. CAP Bench has seven applications from different domains: Features from Accelerated Segment Test (FAST), Friendly Numbers (FN), Gaussian Filter (GF), Integer Sort (IS), K-Means (KM), LU Factorization (LU) and Traveling-Salesman Problem (TSP). Also, these applications are implemented with different parallel patterns (Map, MapReduce, Divide and Conquer, Stencil and Workpool). All the applications are developed in C language and rely upon two runtime libraries: OpenMP (for general-purpose multicores) and the Kalray proprietary API (for Kalray MPPA-256).

We selected five applications from CAP Bench (SOUZA et al., 2010) that feature different memory access patterns. 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 (SOUZA et al., 2010).

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.

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.

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.

LU This program factors a matrix A as the product of two triangular other matrices L and U. The factorization is performed using Gaussian elimination.

IS This application sorts a collection of integer numbers. The chosen algorithm implementation performs sorting based on buckets. Integer numbers are distributed across buckets, which are sorted individually. Then, all buckets are merged to achieve the final result.

4.1.1 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.

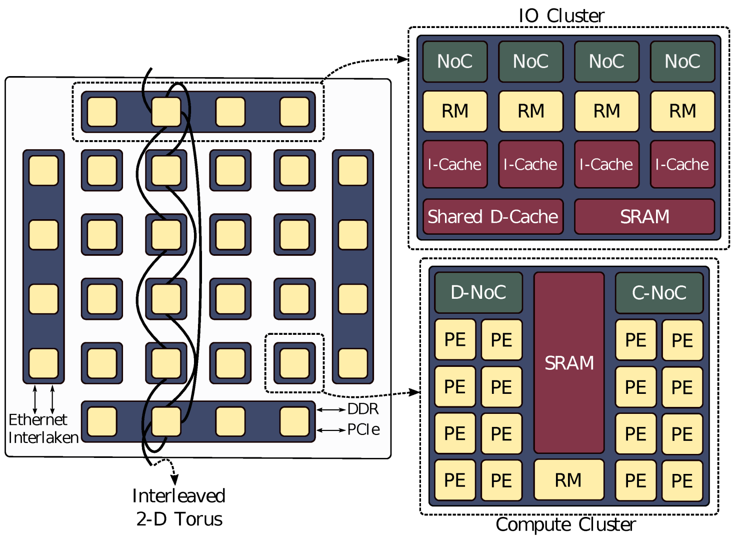
Figure 14 pictures an architectural overview of Kalray MPPA-256 (Bostan architecture), which was the processor adopted in this dissertation. It features 272 general-purpose cores and 16 firmware-cores, called Processing Elements (PEs) and Resource Managers (RMs), respectively, all running at 400 MHz. Cores within the same cluster share some local hardware resources, such as 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 and a shared data cache. 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

Figure 14 – MPPA-256 lightweight manycore processor.



Source: Penna, Francis & Souto (2019)

NoC (C-NoC), which is reserved for small control messages. Kalray MPPA-256 has built-in DMA controllers in their NoC interfaces to enable asynchronous communications.

4.1.2 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 detail.

1. **Methodology validation (Section 4.2.1)**: it aims at validating our trace-driven methodology by comparing the original *heatmaps* 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*, to evaluate if the overall memory access pattern behavior of the original applications are preserved.
2. **Case study (Section 4.2.2)**: it shows 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, to avoid the high latency of fetching pages from remote memories. Originally, the software-managed cache implementation used a simple FIFO page replacement policy, since Kalray MPPA-256 has paramount hardware limitations[1]

[1] 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.

> 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 NFU and Aging inside Nanvix. Based on the obtained results, hardware architects can decide whether 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 (see Section 2.5). In both policies, the page with the lowest reference counter is chosen to be evicted from the software-managed cache.

The reference counter update occurs when a page is brought to or accessed from the software-managed cache. On this basis, the fixed time interval dictates how many of these operations are performed without changes on the reference counters. Only pages that were accessed during this interval have their reference counter updated. Therefore, the interval is extremely important, because it impacts directly the reference counter and, consecutively, the behavior of replacement policies.

We ran CAP Bench applications on Intel Xeon to collect their traces and to build their *heatmaps* (Steps ① and ②). We implemented our *proxy application* (Step ③) as user-level application in Nanvix and ran it on the target lightweight manycore processor (Kalray MPPA-256). The software-managed cache is fully associative and its size in all experiments was 128 kB, which allows us to store 32 pages of 4 kB each. We selected only the *heap* segments of CAP Bench applications during the profiling step (Step ①), since data processed by these applications are allocated dynamically. Finally, our *proxy application* has a deterministic behavior because pages are selected randomly using a standard random generator and a constant seed.[2] Thus, a single execution is enough to obtain the desired results.

4.2 RESULTS

First, Section 4.2.1 presents the validation of our trace-driven methodology. Then, Section 4.2.2 discusses the results obtained in our case study with a software-managed cache implementation.

[2] If the same seed is used throughout executions, the same page numbers will be selected.

4.2.1 Methodology Validation

Figures 15, 16 and 17 presents the *heatmaps* built from the traces collected from the execution of CAP Bench applications (*original*) as well as from our *proxy application*. We divided our *heatmaps* into three classes for presentation purposes: high, medium and low based on the total memory access count of the applications. The total memory access count is directly related to the application and the given number of pages.

We used a different *trial factor* in each class to cope with the total memory access count of applications:

- **High**: $tf = 1000$ and $tf = 10000$;
- **Medium**: $tf = 100$ and $tf = 1000$; and
- **Low**: $tf = 10$ and $tf = 100$.

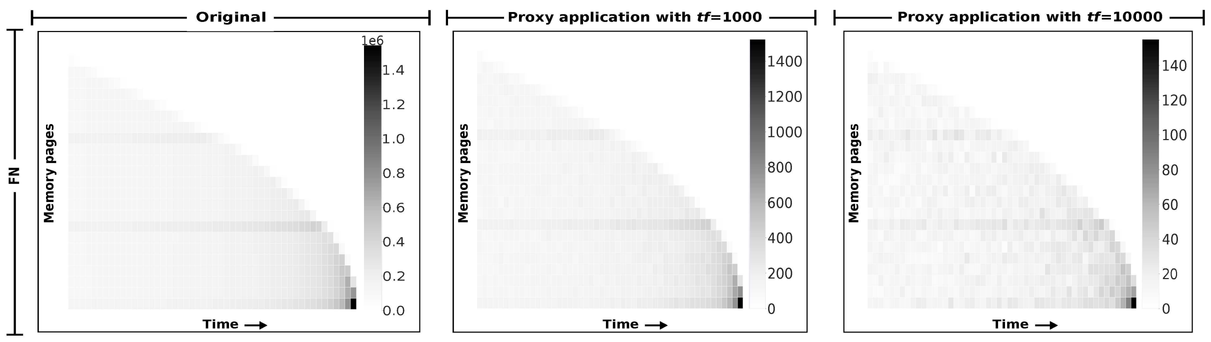
In each class, the first *trial factor* was chosen so as to have a scenario with a moderate yet significant reduction factor. The second one, on the other hand, represents the near maximum possible value for the application. The near maximum value characterizes a *trial factor* that produces dozens to a couple of accesses in each time *bin*. Thus, a higher reduction factor may introduce time *bins* with no memory accesses, which is a paramount information in the *heatmaps*. The x-axis in Figures 15, 16, 17 represents the *temporal behavior* (*timestamps* of memory page accesses), whereas the y-axis represents the *space behavior* (memory pages). Finally, for a better visualization, the resolutions of *heatmaps* were fine-tuned to have 24 *bins* in the y-axis and 50 *bins* in the x-axis. For some applications, we used dotted regions to represent a region where page accesses were performed to guide our discussion further and provide more visibility.

Figures 15(a), 15(b) and 17(a) show FN, KM and LU applications, respectively. As it can be noticed, the proposed methodology was able to mimic properly the behavior of the original applications but some differences appear as the *trial factor* increases. Although, some of LU application page accesses are not visible, they exist inside the dotted region. Conversely, Figures 16 and 17(b) show a quite consistent page access behavior throughout *trial factors* in comparison with other applications. Overall, the *proxy application* was able to mimic the page access pattern of the original applications, even with a high *trial factor*.

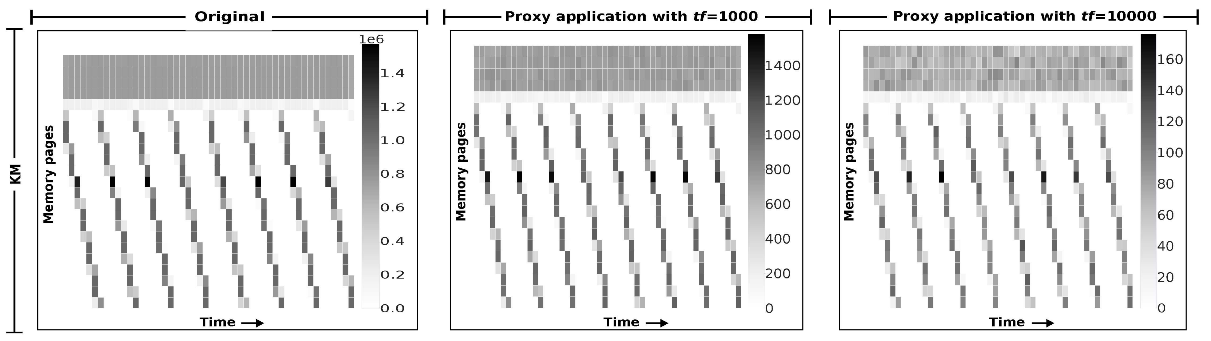
However, there are some causes that may reduce our methodology precision. GF application has highly accessed pages at the top range and there are pages inside the dotted regions that are not frequently accessed. As the *trial factor* increases, some of these page accesses are lost or the number of accesses are significantly reduced. A similar behavior happens with IS. Inside the dotted region, pages of this application have a low number of accesses and a more disperse behavior. A high *trial factor* removes some information and pages that were normally accessed may be removed from our *proxy application heatmap*. This happens due to the probabilistic approach of our methodology,

Figure 15 – High access class.

(a) FN application.



(b) KM application.



which tends to keep information about memory pages that were accessed more frequently. A similar behavior can be seen in a *heatmap* that has, approximately, the same number of accesses in each cell. For instance, Figure 15(a) has some loss of information with $tf = 10000$, because our probabilistic approach sees each cell with the same probability of occurrence. Thus, in this context, there are not enough *trials* to keep all cells with the same number of accesses. In addition, with different *trial factors*, there are pages that are shown with some number of accesses even though the original *heatmap* does not present it. This is due to these pages having a very low number of accesses and, when the *trial factor* increases, they begin to appear.

Figure 16 – Medium access class: GF application.

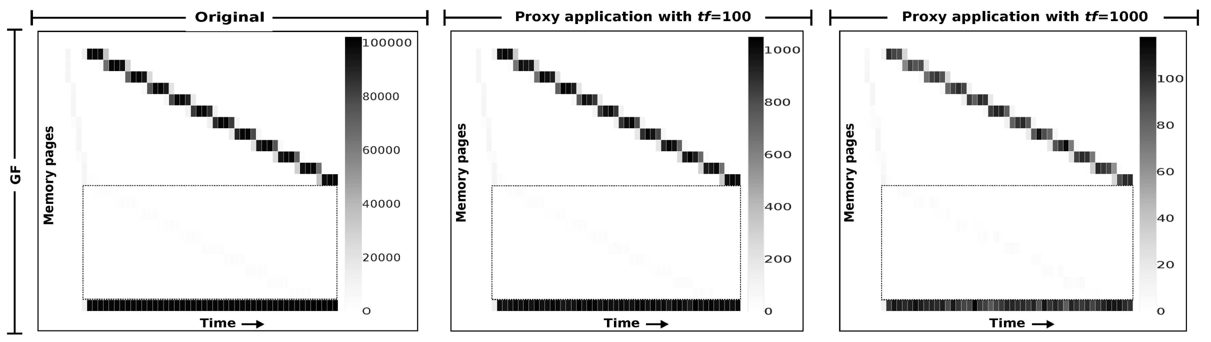
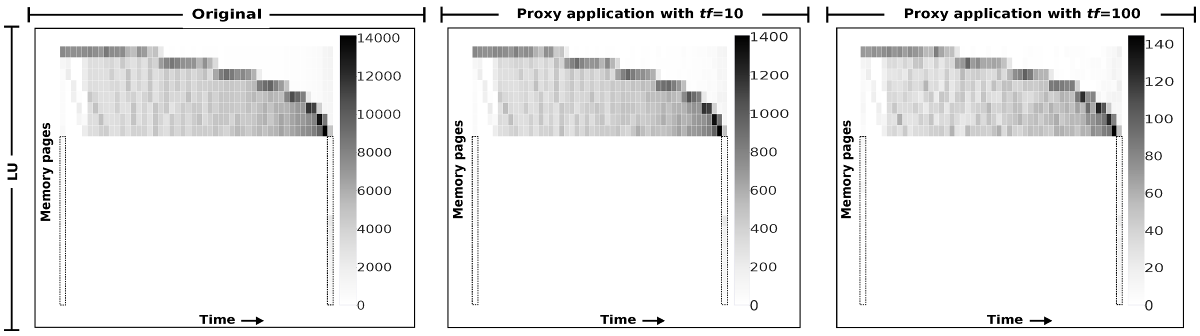


Figure 17 – Low access class.

(a) LU application.



(b) IS application.

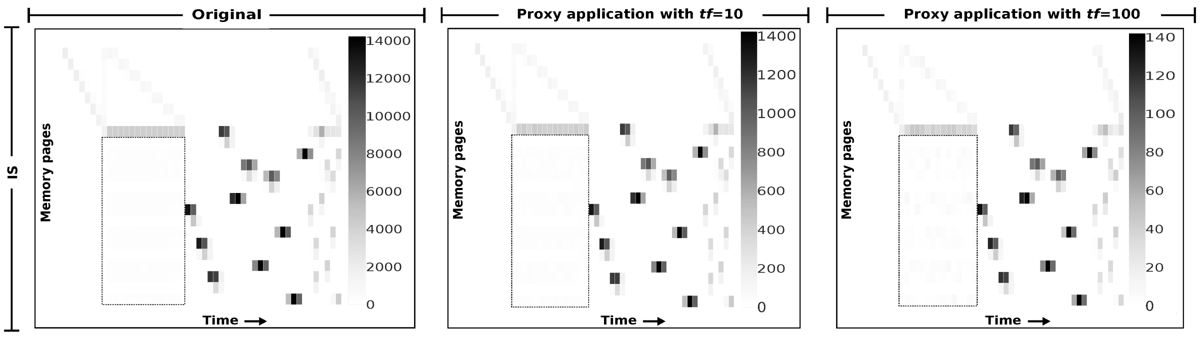


Table 1 shows the Root-mean Square Deviation (RMSD) of the results obtained from the *proxy application* with different *trial factors* (the lower the RMSD value, the higher the similarity between the access pattern of the original application and the one produced by the *proxy application*). The very 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, all applications showed similar RMSD values, where IS and KM showed the highest and lowest difference, respectively. The main reason for these differences is two-fold: (i) KM presents more pages with different number of accesses and less low accessed pages, which provides a more adequate behavior for our probabilistic approach to work with; and (ii) IS has a set of pages inside the dotted region that are accessed very few times, thus, due to the probabilistic approach, the probability of accessing a page in this set is 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

Table 1 – RMSD of the results obtained from the proxy application.

Application	**RMSD** ($tf = max/10$)	**RMSD** ($tf = max$)
FN	0.364	1.251
KM	0.250	0.778
GF	0.359	1.272
LU	0.339	1.085
IS	0.495	1.533

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 2 – Sizes of trace files and *heatmaps*.

Application	**Trace**	***Heatmap***
FN	1.5 GB	6.9 kB
KM	3.6 GB	5.8 kB
GF	98 MB	3.8 kB
LU	15 MB	4.2 kB
IS	5.3 MB	3.8 kB

4.2.2 Case Study: Software-managed Cache

Our case study is divided into three main results. First, Section 4.2.2.1 discusses the impact of the *trial factors* on the software-managed cache behavior. Second, Section 4.2.2.2 presents a comparison between software-managed cache update time intervals. Finally, based on these results, Section 4.2.2.3 shows the impact of replacement policies on the software-managed cache performance.

The hit ratio was computed periodically with each period defined by a fixed quantity of memory access events. The number of accesses for each period is described as follows:

$$p = \frac{\text{total number of accesses}}{\text{number of periods}} \tag{4.1}$$

The number of periods can be defined arbitrarily to achieve better visualization. Thus, each point in our results represents the software-managed cache hit ratio computed as follows:

$$hit\_ratio = \frac{\text{number of cache hits in } p}{p} \tag{4.2}$$

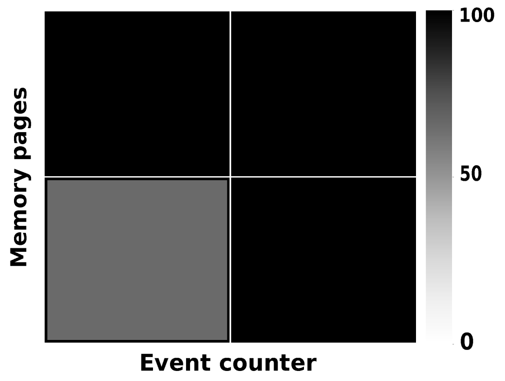
We varied the working size of each application to fit, approximately, 32, 48 and 64 pages. Also, we fine-tuned the input *heatmaps* used by the *proxy application* as follows:

maximum resolution in y-axis (*i.e.*, 1 page per *bin*) and 500 *bins* x-axis. Our empirical tests with our applications showed that this configuration was enough to obtain precise results.

Finally, we use a *software-managed cache heatmap* to provide a better explanation for our claims in this section. Similarly to the *heatmap* produced by the *proxy application*, the x-axis represents a *temporal behavior* and the y-axis represents a *spacial behavior*. The *resolution* of the *heatmap* follows the one produced by the *proxy application* to provide the same *bins* in the x and y axes. Each cell represents how much time a page stays inside the software-managed cache in its respective time *bin* (the darker the color of the cell of the *heatmap*, longer a page stays inside the software-managed cache).

Figure 18 illustrates a conceptual software-managed cache *heatmap*. The first time *bin* presents a page that stays inside the software-managed cache until the end of the *bin*, whereas the other page stays only a part of the total time. In contrast, the second time *bin* shows both pages inside the software-managed cache almost all the time until the end of the *bin*. The *heatmap* is only a guideline for our discussion, hence there is no definitive information about more specific software-managed cache details, such as which page was replaced in a specific time *bin*. In addition, we will select certain parts of the software-managed cache *heatmap* to focus our discussion. Each part may have a different *resolution*, still it remains restrained by the full *proxy application heatmap resolution*.

Figure 18 – Conceptual software-managed cache pages *heatmap*.



4.2.2.1 Trial Factor

In this result set, we discuss how the *trial factor* impacts our software-managed cache behavior. Thus, we: (i) used all five applications; (ii) varied the working set to approximately 32, 48 and 64 pages for each application; (iii) varied the *trial factor* based on Section 4.2.1; and (iv) fixed an update time interval in 50 operations[3] for Aging and NFU replacement policies.

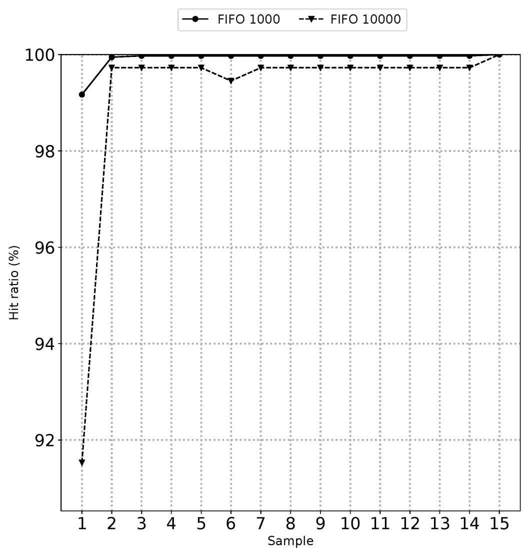
Figure 19 presents the impact of different *trial factors* on the software-managed cache behavior. Mainly, there are four main behaviors found on our results. Therefore, we

[3] There are 50 operations to bring or access a page before the next reference counter update.

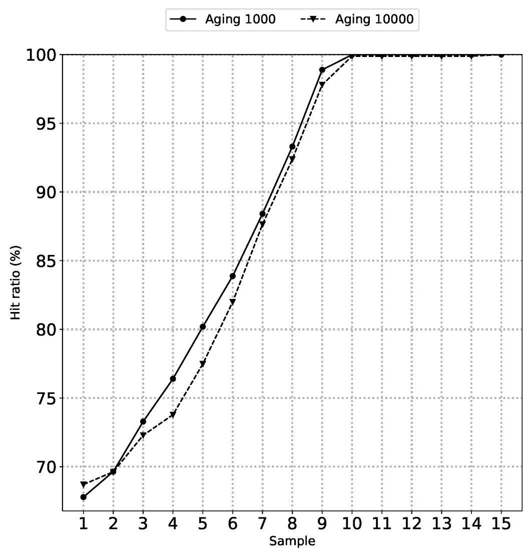
only present results related to these behaviors to provide a more direct and uncluttered discussion about *trial factor* variation.

Figure 19 – FN, GF and IS *trial factor* evaluation.

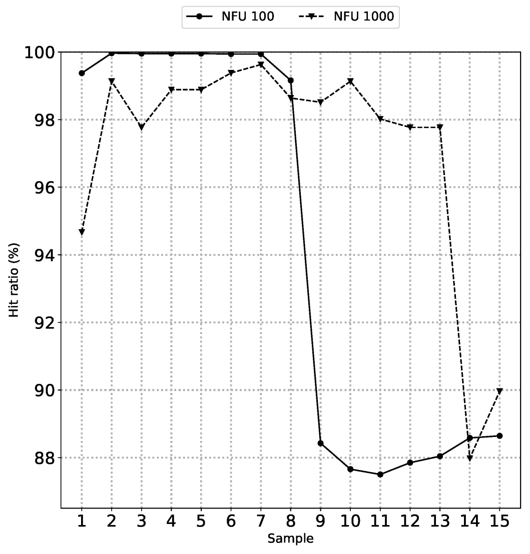
(a) FN with 32 pages and FIFO algorithm.



(b) FN with 48 pages and Aging algorithm.



(c) GF with 64 pages and NFU algorithm.



(d) IS with 64 pages and Aging algorithm.

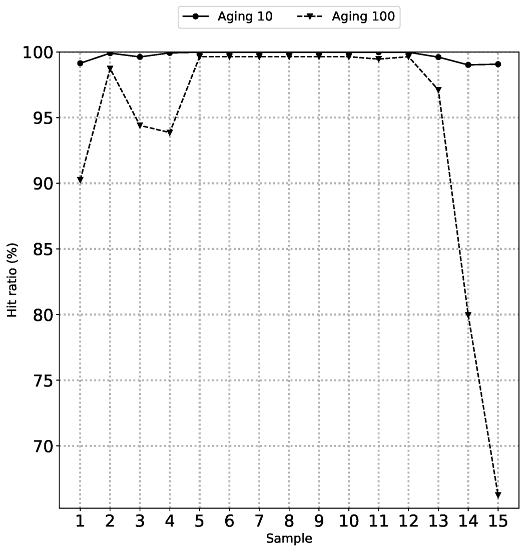
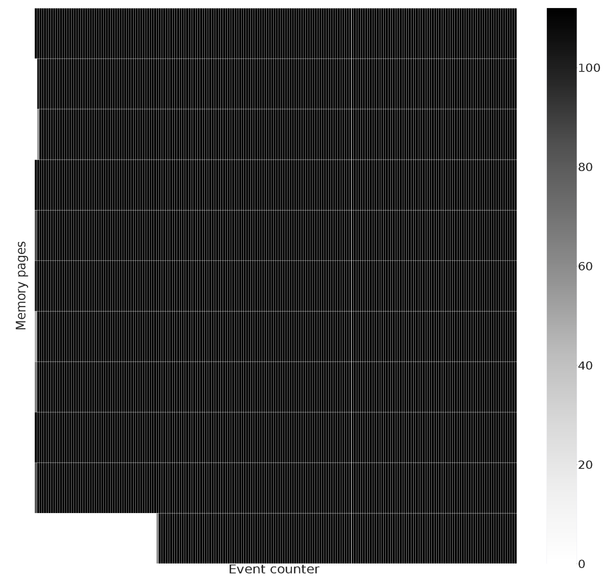


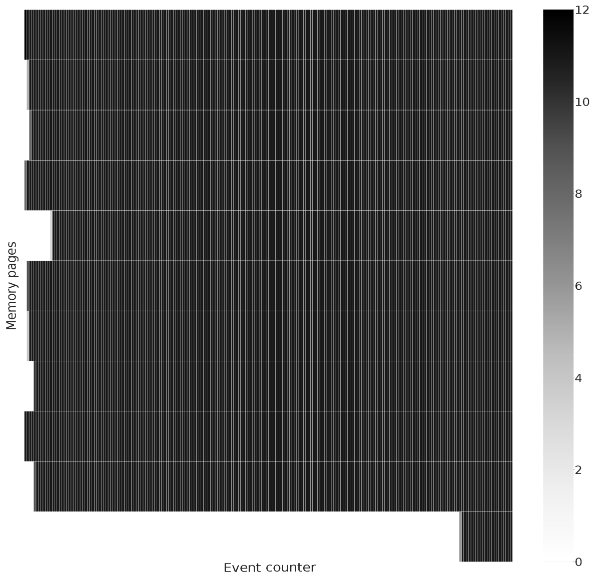
Figure 19(a) shows the software-managed cache behavior for FN application with 32 pages and FIFO algorithm. The behavior has a steep climb on hit ratio and a great difference between *trial factors* at the beginning. The reasoning for this is because FN application has almost the same number of accesses for each page, and our methodology selects these pages with, approximately, the same probability. Therefore, a high *trial factor* decreases the number of pages selected by our methodology in the first *bins*, which increases the chance of misses in the near future. To illustrate our discussion, Figure 20 shows a small part of FN application software-managed cache *heatmap*, which addresses only its last pages and first time *bins*. With a near maximum *trial factor*, the first time *bins* show several pages that could be brought sooner to the software-managed cache and increase hit ratio.

Figure 20 – FN cache *heatmap* with 32 pages and FIFO algorithm.

(a) FN with a moderate *trial factor*.



(b) FN with near maximum *trial factor*.

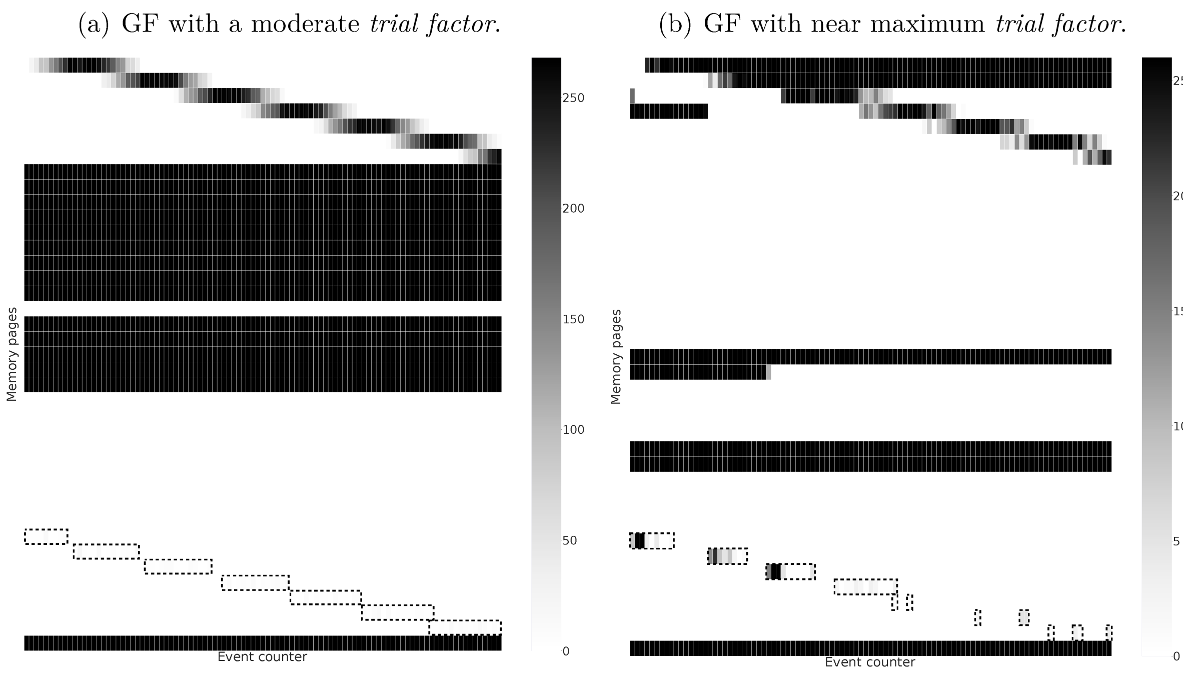


However, even with this impact on hit ratio, our methodology can recover after some number of accesses, because FN has an overall consistent behavior and, even with fewer accesses for each time *bin*, not selected pages by our *proxy application* appear as execution progresses. Thus, the impact on the hit ratio is dampened. Moreover, the sixth and fifteenth samples present a different behavior from the others. The former behavior is due to a new page introduced by our *proxy application*, whereas the latter is because there are only a few pages being heavily accessed in the end of application execution, which increases the hit ratio of the respective sample. Figure 20 demonstrates the reasoning behind the sixth sample. The last page is brought later with maximum *trial factor*, thus there is an impact on hit ratio.

Overall, the near maximum *trial factor* reduces lightly the maximum hit ratio in comparison with the moderate *trial factor*. Most applications present this behavior, and it is expected, because a lower *trial factor* brings a higher number of accesses for each time *bin*, which brings a more precise representation of the original application. Figure 19(b) illustrates this behavior further, where FN application with 48 pages and Aging algorithm has a similar difference. In both aforementioned results, the difference between *trial factors* is considerably small.

Furthermore, Figure 19(c) shows GF application with 64 pages and NFU algorithm, where the near maximum *trial factor* gives a better hit ratio after the seventh sample. This behavior is specific for the NFU algorithm, where highly accessed pages are never removed from the software-managed cache. As can be seen in Figure 16, GF has several highly accessed pages, but it has little accessed ones as well. A high *trial factor* decreases the total number of accesses, reduces pages total reference counter and enables pages that are highly accessed to be removed sooner by the little accessed ones. Figure 21 illustrates part of the GF application software-managed cache *heatmap*, which addresses its last pages and last time *bins*. The moderate *trial factor* increases significantly the top pages reference counter. Hence, bottom pages are removed frequently. In contrast, the near maximum *trial factor* decreases the reference counter gap and keeps bottom pages longer inside the software-managed cache. In addition, the gap begins to increase at the end of application execution, because pages reference counter are always increasing.

Figure 21 – GF cache *heatmap* with 64 pages and NFU algorithm.



(a) GF with a moderate *trial factor*. (b) GF with near maximum *trial factor*.

Figure 19(d) presents IS application with 64 pages and Aging algorithm, where a higher *trial factor* gives a significant lower hit ratio. The reasoning for this behavior is that a high *trial factor* with a high enough update time interval degrades software-managed cache performance. A higher update time interval makes pages take longer to change their reference counter. Hence, when a page has to be removed from the software-managed cache, a page that will be accessed in the near future may be chosen, because their reference counter was not updated thus far. Moreover, due to the similar number of accesses between pages, some of them may have similar or even equal reference counters. Therefore, pages may be removed from the software-managed cache just to be brought shortly after. A higher *trial factor* gives more space for this behavior to happen, because there is less *trials* for each time *bin* and the reference counter update may occur some *bins* later. Also, an update time interval higher than the quantity of *trials* in a time *bin* implies that the reference counter is updated in another *bin*, which can cause a significant impact on page replacement. Section 4.2.2.2 provides a more in-depth discussion about update time intervals.

Our experiments showed that, even with an overall similar hit ratio between several applications, there are two cases that should be paid attention to when we have a high *trial factor*: (i) applications with NFU algorithm and highly accessed pages in the same time bins as low accessed ones may change drastically the hit ratio; and (ii) a high update time interval may be undesirable for high *trial factors*. However, these unexpected behaviors can be solved by simple tweaks, such as a lower *trial factor* and update time interval.

4.2.2.2 Update Time Interval

In this section, we aim to discuss more about the reference counter update time interval. To this end, we: (i) used all five applications; (ii) varied the working set to approximately 32, 48 and 64 pages for each application; (iii) varied the update time interval to 10, 50 and 100 operations; and (iv) fixed the moderate *trial factor* based on Section 4.2.1. To provide a more clean discussion, we demonstrate specific behaviors that are repeated throughout our obtained results.

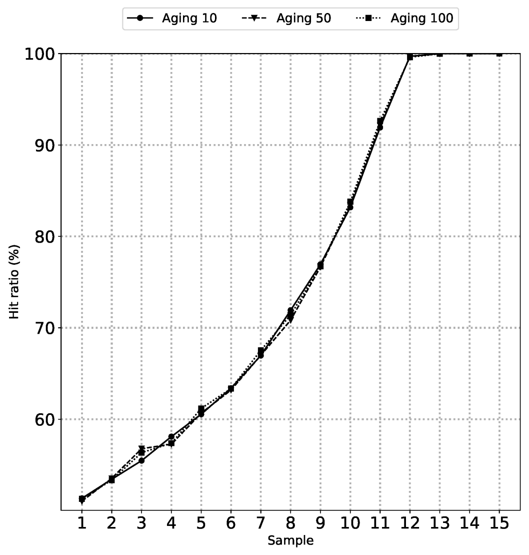
Figures 22 and 23 presents the update time interval evaluation for all applications with 64 pages and Aging algorithm. Overall, for the Aging algorithm, there is no significant impact on our software-managed cache hit ratio. Figure 22 shows the behavior for high and medium class applications, which has a very similar hit ratio between update time intervals. The reasoning for this is two-fold: (i) there is a high quantity of *trials* for each time *bin*, which provides a more precise reference counter updates for our algorithm to work with even with a high update time interval; and (ii) the application page access behavior is consistent, and each update time interval still enables the Aging algorithm to consider pages time span in a similar manner. In addition, a high quantity of *trials* provides even more accesses for the same page if its accessed contiguously between time *bins*. Thus, the impact of a high update time interval is further mitigated.

Conversely, Figure 23 demonstrates the hit ratio for low class applications. At the beginning and end of each application, there is a drop in hit ratio due to sudden appearance of new or long time not accessed pages. In this context, IS and LU applications have this difference in hit ratio, because a high update time interval makes Aging disregard page access time span. Moreover, LU has a greater impact on hit ratio, because these pages in the beginning and end of the execution are shortly accessed pages. A shortly accessed page does not have a long contiguous access, which decreases significantly the total number of accesses for that page and increases the impact of a high update time interval.

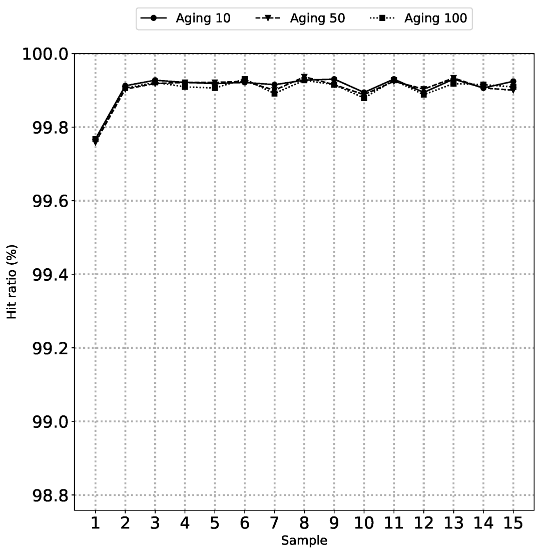
Therefore, the worst update time interval for the Aging algorithm is the higher one. This is because a higher interval hinders the algorithm performance on applications

Figure 22 – FN, KM and GF update time interval evaluation with 64 pages and Aging.

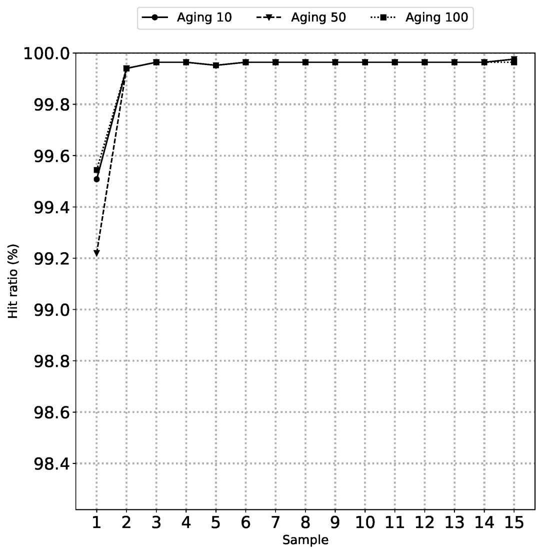
(a) FN application.



(b) KM application.



(c) GF application.

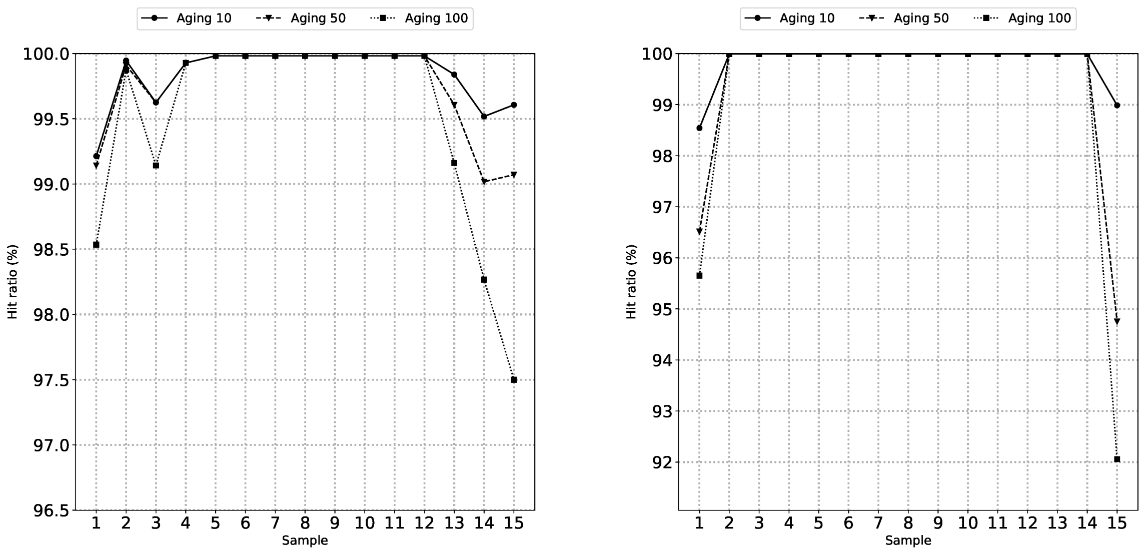


with inconsistent page accesses. It makes pages to be removed even before there is a update on the reference counter or update reference counters in bulks, which may provide the same counter for several pages. Also, as mentioned on Section 4.2.2.1, a higher interval may bring undesired behavior if there is not enough *trials* in a time *bin*.

On the other hand, Figures 24 and 25 demonstrate the update time interval evaluation for all applications with 64 pages and NFU algorithm. The update time interval variation on NFU algorithm shows the most significant impact on hit ratio and overall behavior. As can be noticed in Figures 24(a) and 24(c), it seems that a lower update time interval is the best one, but our remaining results show otherwise. NFU does not

Figure 23 – IS and LU update time interval evaluation with 64 pages and Aging.

(a) IS application. (b) LU application.



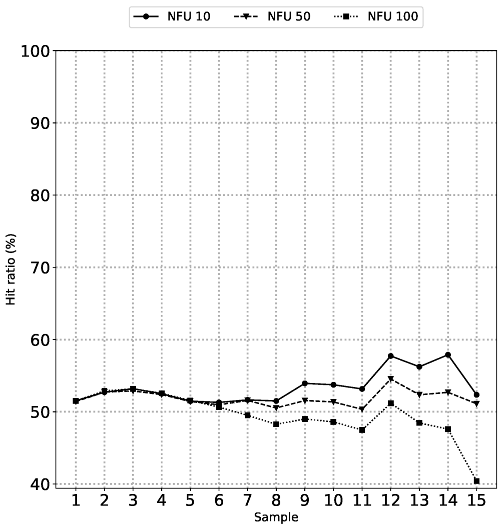
have a consistent enough behavior to select only one update time interval, because a page with enough accesses may be kept inside the software-managed cache even if it is not accessed anymore. Thus, the update interval, which modifies when the reference counter is updated, can also change drastically the overall behavior of the algorithm.

Figure 24(a) shows the FN application hit ratio, where a high update time interval significantly impacts the hit ratio. This is due to the fact that FN has several accesses in all working set pages in the beginning of execution. NFU increases the reference counter of some, fills the software-managed cache and makes other pages dispute between themselves a slot inside it. In this context, the update time interval may change the pages that are inside the software-managed cache. For instance, there is a software-managed cache with four slots and a specific update time interval. After execution, there are four pages inside the software-managed cache. Due to a different design choice, the update time interval is modified and another execution is performed. After the second execution, there may be four different pages inside the software-managed cache. The reasoning for this is because the update time interval changes how the reference counters are updated and, thus, which pages persist inside the software-managed cache.

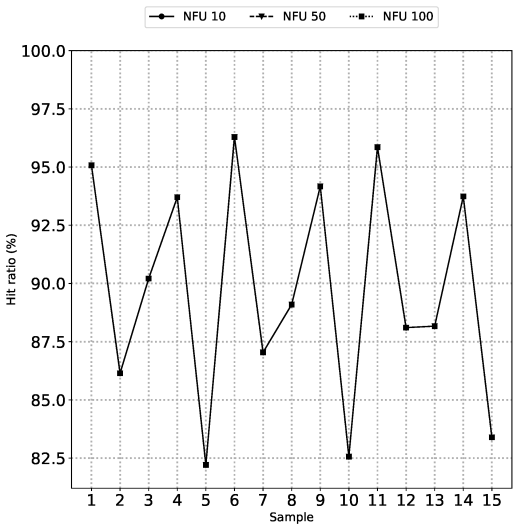
In contrast, Figure 24(b) presents KM application with identical hit ratios for all update time intervals. The main reasoning for this is the page access behavior of the application depicted by Figure 15(b). All pages above the FIFO pattern region fill the software-managed cache and achieve a high reference counter. Notwithstanding, pages bellow this region have to dispute a single slot inside the software-managed cache between them, because when the lowest reference counter is removed, the next page more likely to be removed is the newest one. Hence, the hit ratio is decreased. To further base

Figure 24 – FN, KM and GF update time interval evaluation with 64 pages and NFU.

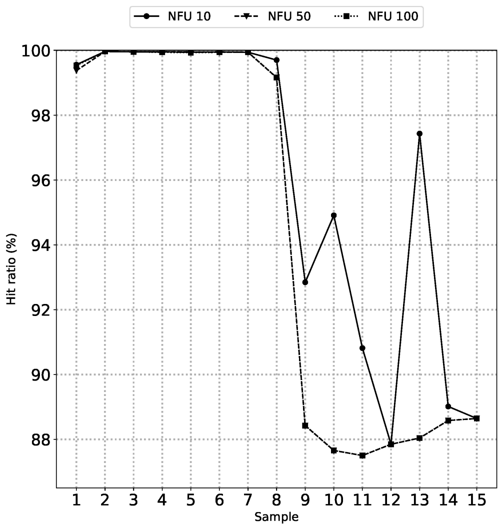
(a) FN application.



(b) KM application.



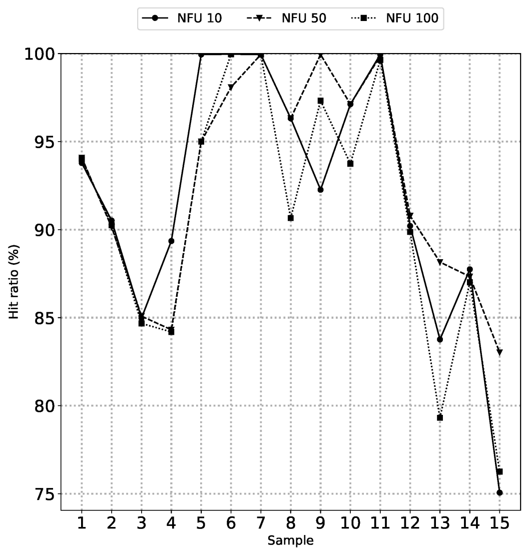
(c) GF application.



our discussion, Figure 26 shows a slice the of KM application software-managed cache *heatmap* with a working set of 64 pages, NFU algorithm and a fixed 50 update intervals. In this *heatmap*, only the middle pages and time *bins* were selected. Due to their low reference counter, bottom range pages (FIFO pattern region) are removed from the software-managed cache as soon as they arrive just to be brought again in the near future. In this context, different update time intervals do not interfere, because: (i) a great part of heavily used pages in our application fit the software-managed cache; and (ii) the reference counter is only used when a page has to be removed from the software-managed cache. At that time, several pages have a high enough reference counter to make the algorithm only use one slot inside the software-managed cache.

Figure 25 – IS and LU update time interval evaluation with 64 pages and NFU.

(a) IS application.



(b) LU application.

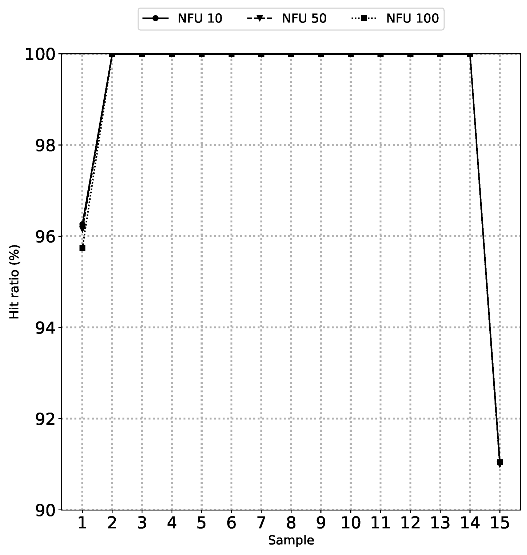
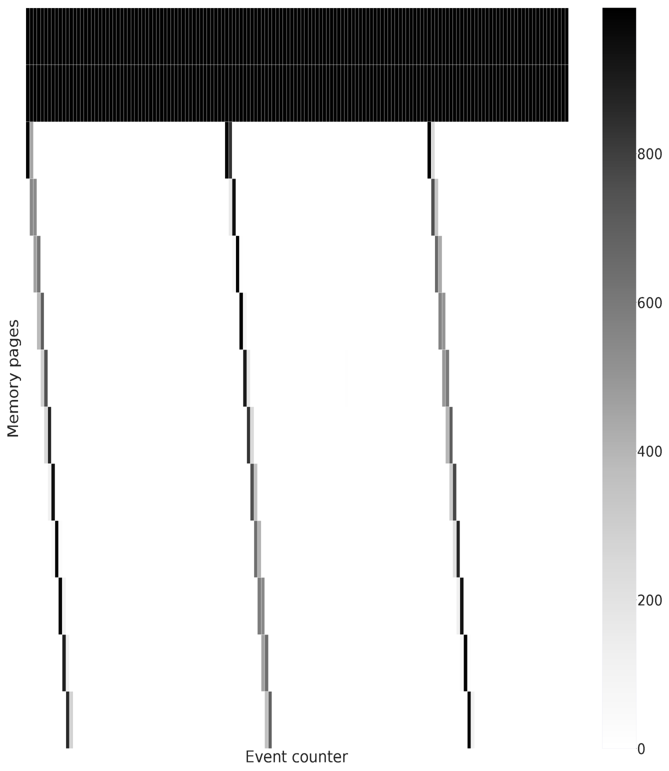


Figure 26 – KM cache *heatmap* with 64 pages and NFU algorithm.

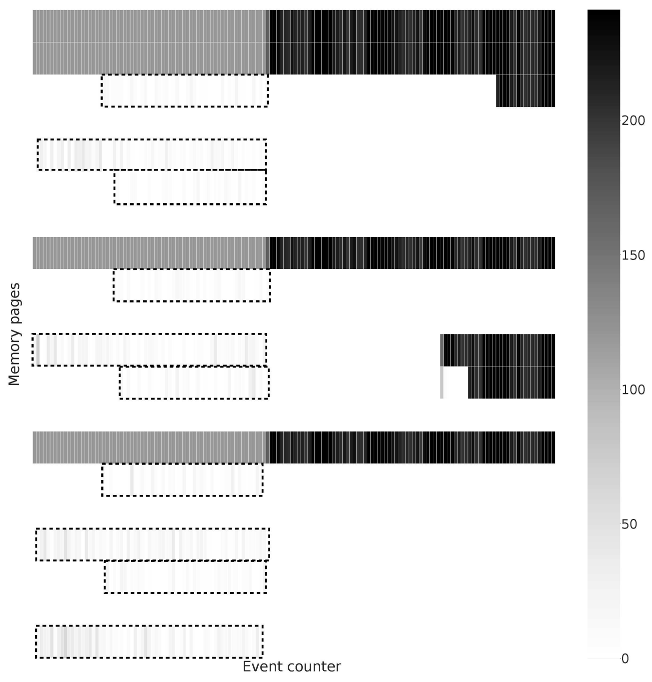


Therefore, our software-managed cache has a similar page access behavior even with a considerably high update time interval. In addition, note that if the update time interval is high enough, the reference counter may be updated after a long time, which may affect the overall hit ratio of the application. Similarly, Figure 25(b) illustrates LU application with almost no variation between hit ratios with different update time intervals. This application has the same aforementioned behavior with the pages inside the dotted regions in Figure 17(a).

Figure 24(c) shows GF application hit ratio, where only the lower update time interval gives some samples with a better hit ratio. GF application has high and low pages being accessed in the same pattern, where they are highly and low accessed in the same time *bin*, respectively. Usually, in the NFU algorithm, the higher the number of accesses on a page, the higher its reference counter. However, a different update time interval may impact this behavior. A high update time interval prevents a high number of updates in the time *bin* and may degrade software-managed cache performance, because the algorithm will have little information about pages to work with. On the other hand, a lower update time interval brings more updates in a time *bin* and provides a higher reference counter to all pages inside that *bin*. Thus, GF has some pages that benefited from this characteristic and, due to application behavior, there was peaks of hit just to be lowered again.

Figure 25(a) presents IS application with a quite diverse hit ratio. The reasoning for this is because NFU algorithm has to deal with a disperse page access behavior and, thus, predicts poorly next accessed pages. To approach our discussion in more detail, Figure 27 illustrates a part of the IS application software-managed cache *heatmap* with 64 pages and NFU algorithm. Only middle pages and first time *bins* were selected. Top range pages receive high reference counters, whereas some bottom range ones need to dispute between them to get a space inside the software-managed cache. In addition, the bottom range pages are removed from the software-managed cache to just be brought in the near future. This behavior decreases significantly the hit ratio.

Figure 27 – IS cache *heatmap* with 64 pages and NFU algorithm.



It is a challenge to select an update time interval for NFU, because the same update time interval, in comparison with other ones, increases hit ratio in some cases and decreases in others. It is necessary to test update time intervals to select which suits best each application. Thus, we argue that, in NFU algorithm, there is no overall best update time interval.

4.2.2.3 Replacement Policies Performance

In this section, we aim to discuss the performance of each algorithm in our software-managed cache. To this end, we: (i) used all five applications; (ii) varied the working set to approximately 32, 48 and 64 pages for each application; (iii) used FIFO, NFU and Aging algorithms; (iv) fixed the update time interval to 10 operations for Aging and NFU algorithms,[4] (v) fixed the moderate *trial factor* based on Section 4.2.1. Our results show the variation of the software-managed cache hit ratio obtained from our *proxy application*.

As a proof of concept, Figure 28 presents four applications with a working set of 32 pages. All these pages almost fits the software-managed cache, which provides a near maximum hit ratio. All applications show a small decline on hit ratio at the beginning, because of the software-managed cache warmup. However, a different behavior is observed near the end of the execution of IS, where the hit ratio is slightly reduced. Figures 29(b) and 29(c) illustrate further this case. The former shows the new page at the middle range being brought inside the software-managed cache, whereas the latter presents pages appearing at the end of execution.

Figure 30 illustrates the hit ratio for FN application with 48 and 64 pages. As shown in Figure 15(a), 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 software-managed cache. The best results were achieved with FIFO and Aging. FIFO algorithm deals extremely well with the FN memory access behavior, because it follows the same pattern. Conversely, Aging removes efficiently pages that were not accessed anymore and follows a similar FIFO behavior.

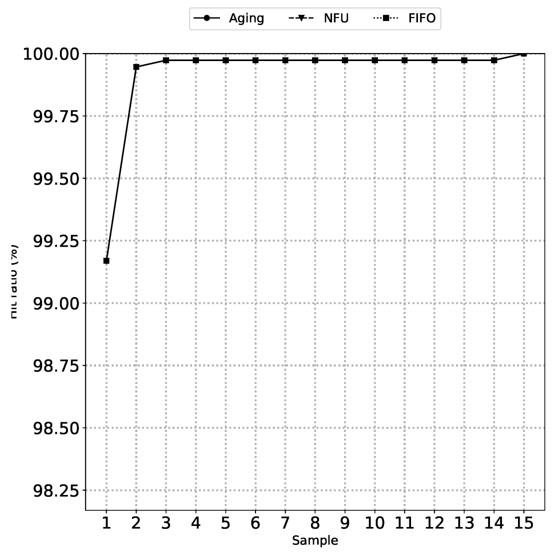
Furthermore, NFU algorithm does not perform well with FN application. This is due to all working set pages being accessed at the first time bins. The pages stored inside the software-managed cache increases drastically their reference counter, and they are not removed anymore. Thus, pages with a lower reference counter suffer drastically.

Figure 31 shows that Aging is more beneficial for KM. This application has two well-defined page access behaviors: pages on top and bottom ranges in Figure 15(b) are

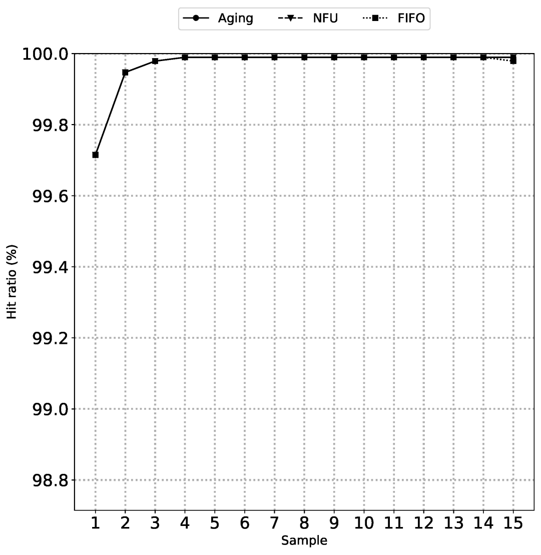
[4] We selected the same update time interval for both algorithms to provide a more fair comparison.

Figure 28 – FN, KM, GF and IS performance evaluation with 32 pages.

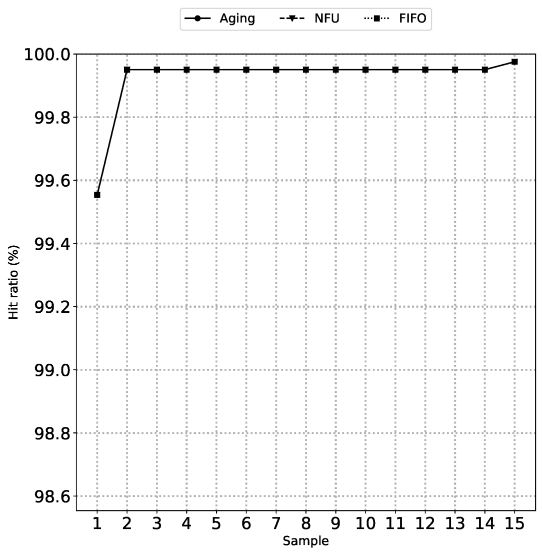
(a) FN application.



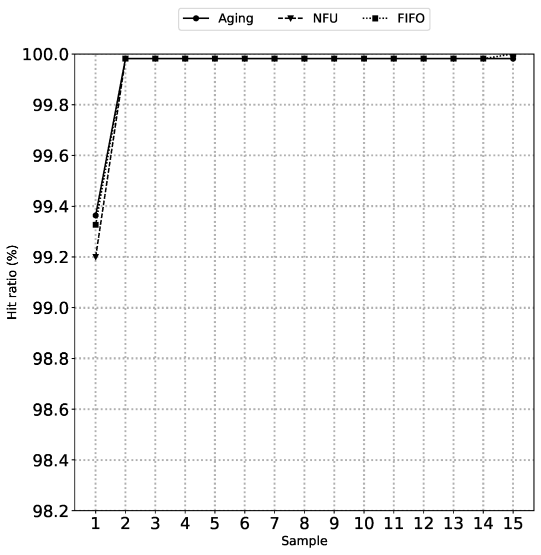
(b) KM application.



(c) GF application.



(d) LU application.

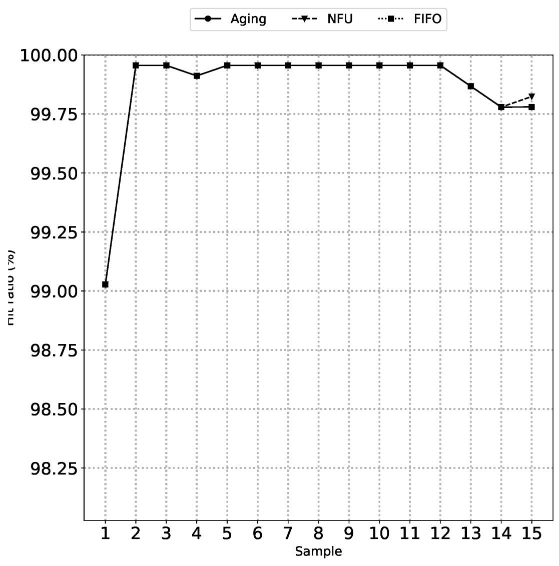


always accessed throughout the execution whereas middle range pages are accessed in a FIFO ordering fashion. The FIFO and Aging page replacement policies are able to correctly keep middle range pages in software-managed cache. These policies showed a stable software-managed cache hit ratio above 99% on average.

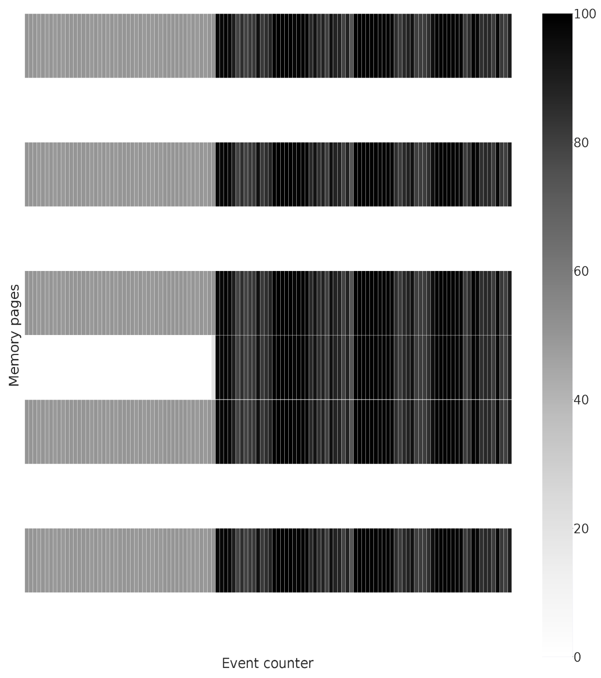
On the other hand, NFU presented a high hit ratio variation with several spikes. The main reason is that NFU constantly removes middle range pages because they usually have lower reference counters. This problem is solved by Aging, since it is aware of the time span of page use. Figure 32 presents the main reasoning for our claim. This figure shows a portion of KM application software-managed cache *heatmap* with 48 pages and

Figure 29 – IS application performance evaluation and software-managed cache *heatmap* with 32 pages.

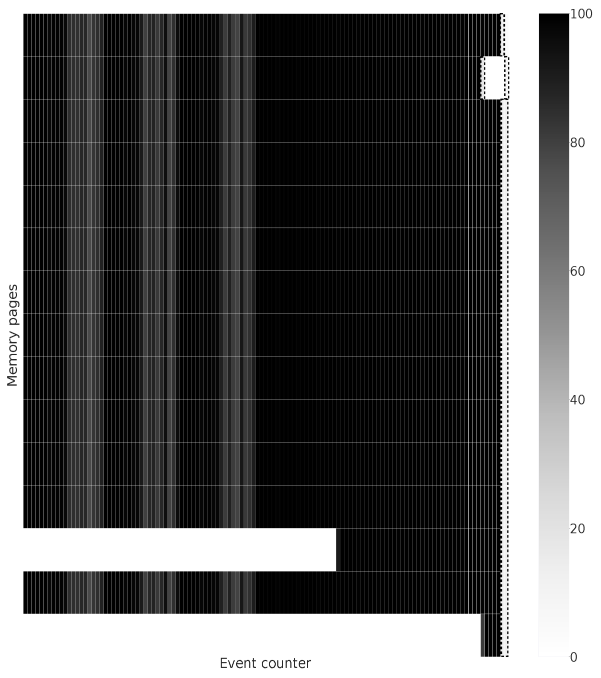
(a) IS application.



(b) IS application fourth sample software-managed cache *heatmap*.



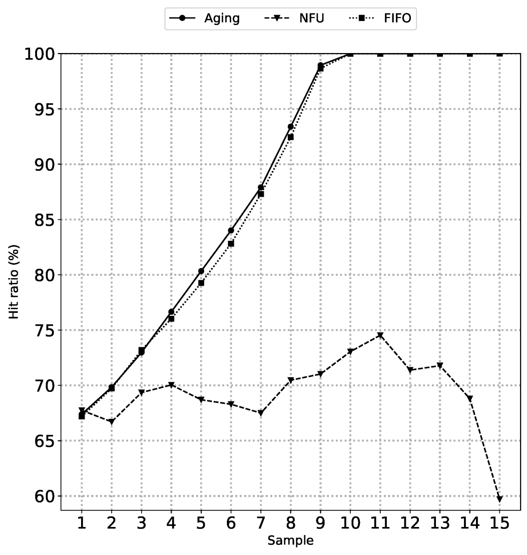
(c) IS application thirteenth sample software-managed cache *heatmap*.



NFU algorithm. There are several FIFO access patterns in KM, but our *heatmap* only shows three of them. As can be noticed, they illustrate pages being removed and brought immediately back to the software-managed cache. In addition, these patterns repeat throughout the application and are not far apart from each other, but the algorithm does not tend to keep pages from them inside the software-managed cache. Thus, KM performance suffers because of this behavior.

Figure 30 – FN performance evaluation with 48 and 64 pages.

(a) FN application with 48 pages.



(b) FN application with 64 pages.

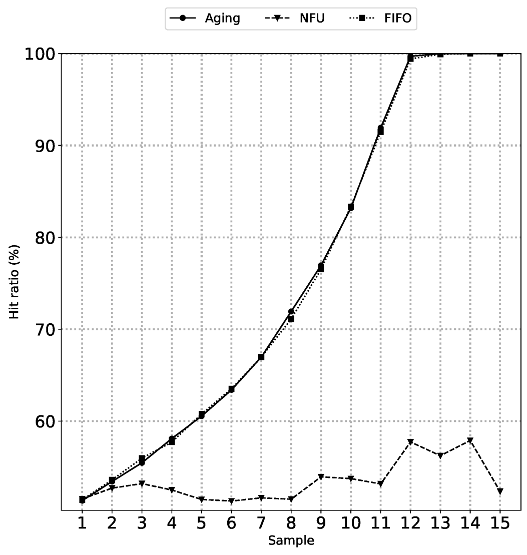
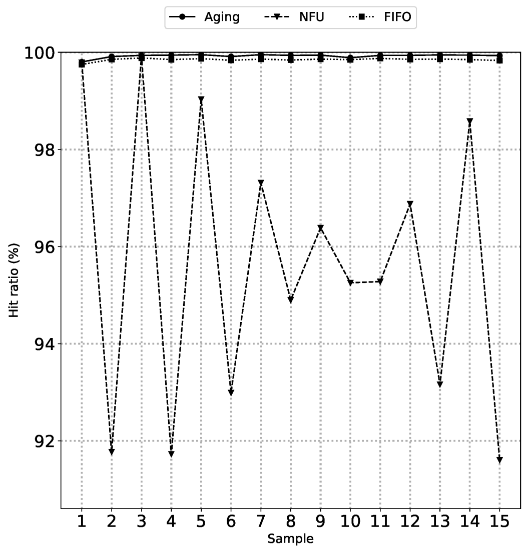


Figure 31 – KM performance evaluation with 48 and 64 pages.

(a) KM application with 48 pages.



(b) KM application with 64 pages.

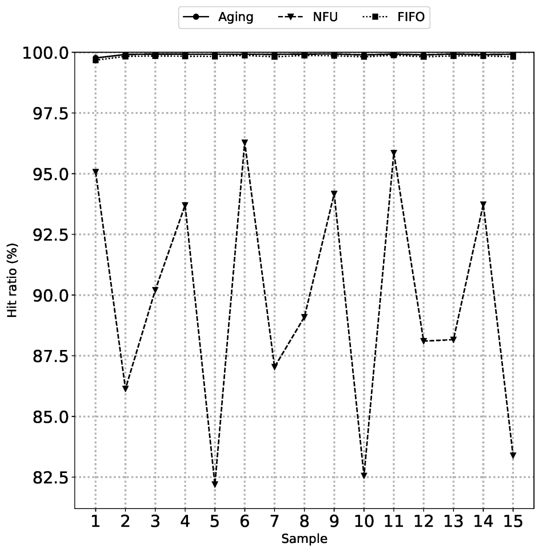
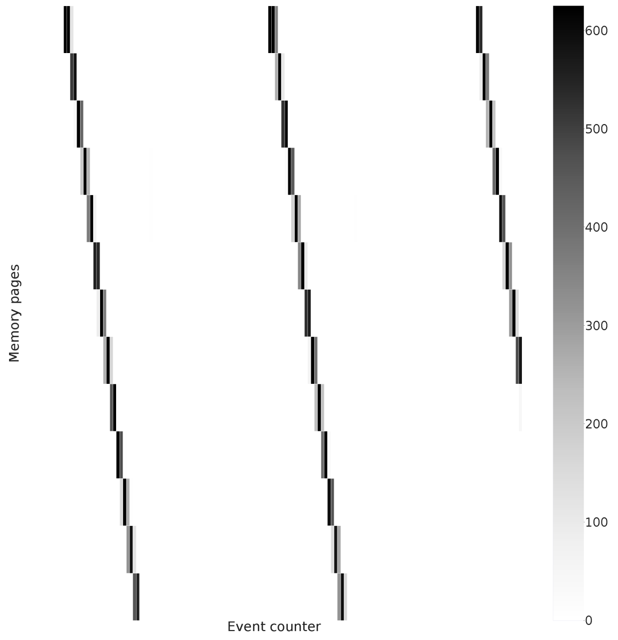


Figure 33 presents GF application performance with 48 and 64 pages. As shown in Figure 16, this application has one page that is constantly accessed throughout the execution whereas the access pattern of all other pages follows the FIFO rule. We observed that FIFO and Aging page replacement policies were able to keep a hit ratio very close to 100%. Both algorithms deal well with a FIFO page access behavior. However, NFU suffers because it maintains highly accessed pages indefinitely inside the software-managed cache

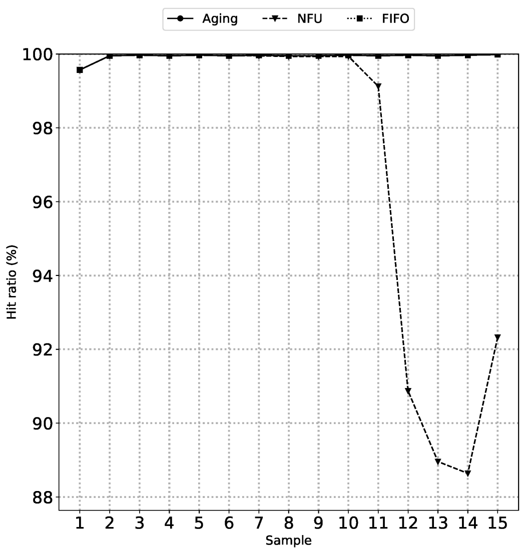
Figure 32 – KM application software-managed cache *heatmap* with 48 pages and NFU algorithm.



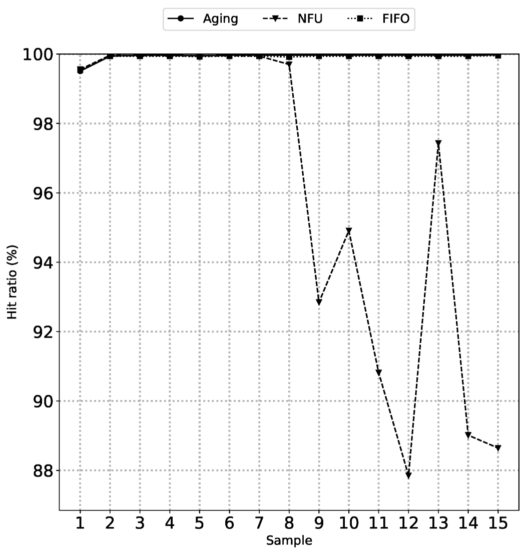
and the middle pages,[5] which have low accesses, are removed as soon as possible just to be brought to the software-managed cache again. To illustrate this behavior, Figure 34 shows a part of GF application software-managed cache *heatmap* with 48 pages and NFU algorithm. It was selected pages from the middle range up to the bottom range and the last time *bins*. As can be noticed, low range pages (dotted regions) are brought to the software-managed cache to be immediately removed for top range pages (highly accessed).

Figure 33 – GF performance evaluation with 48 and 64 pages.

(a) GF application with 48 pages.

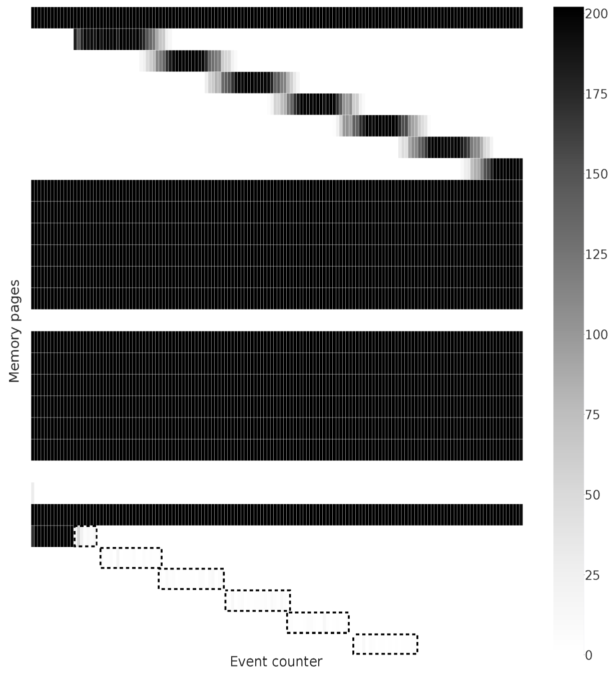


(b) GF application with 64 pages.



5 Represented by the dotted region in Figure 16.

Figure 34 – GF software-managed cache *heatmap* with 48 pages and NFU algorithm.



Furthermore, Figure 35 illustrates the hit ratio of IS application with 48 and 64 pages. As can be seen in Figure 17(b), This application has a really disperse page access behavior, where pages are accessed at distant time *bins*. Even with this behavior, FIFO and Aging algorithm deal with the application reaching almost 100% hit ratio. The reasoning for a lower hit ratio in comparison with aforementioned applications for these two algorithms is two-fold: (i) first accessed pages are highly disperse for our algorithms to predict well and achieve a higher hit ratio; and (ii) at the end of execution, IS application access several new pages or pages that were not accessed in a long time.

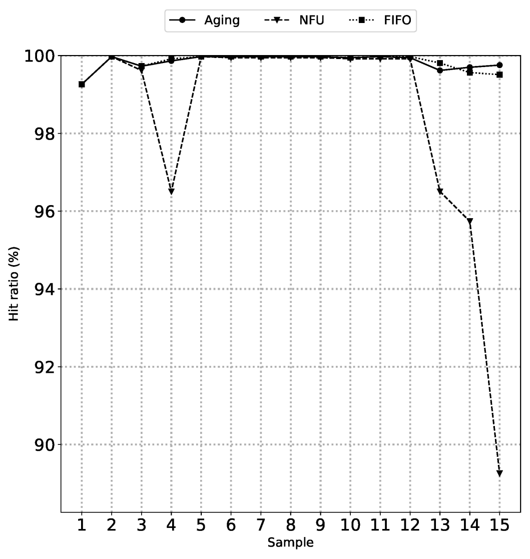
Conversely, NFU does not deal well with the disperse behavior. The algorithm keeps a high reference counter for the first accessed pages and has to manage the other pages. Fortunately, the middle pages fit the software-managed cache and were accessed at the beginning, which increases performance. However, at the end, not accessed in a long time and new pages arise. Hence, pages with low reference counter have to dispute between them to get a slot inside the software-managed cache, which decreases significantly the hit ratio.

Figure 36 presents the hit ratio of LU application with 48 and 64 pages. As can be seen in Figure 17(a), the application access behavior is consistent on top pages and there are accesses on middle and bottom pages (dotted region). At the beginning and end of execution, all pages follow a FIFO access pattern and, at these moments, middle and bottom pages overload the software-managed cache, which has no space to settle them. Thus, this behavior decreases hit ratio.

Overall, FIFO and Aging present a comparable hit ratio above 98%. However, the former algorithm has the advantage, because of the FIFO pattern. To better demonstrate

Figure 35 – IS performance evaluation with 48 and 64 pages.

(a) IS application with 48 pages.



(b) IS application with 64 pages.

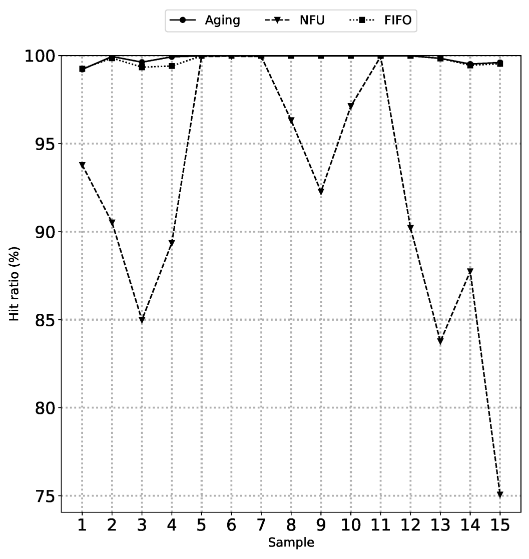
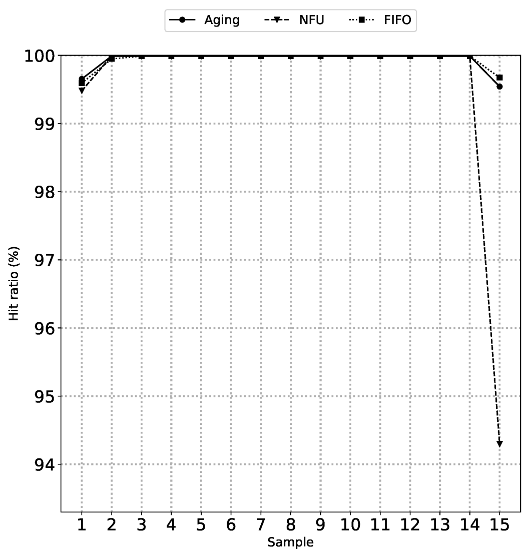
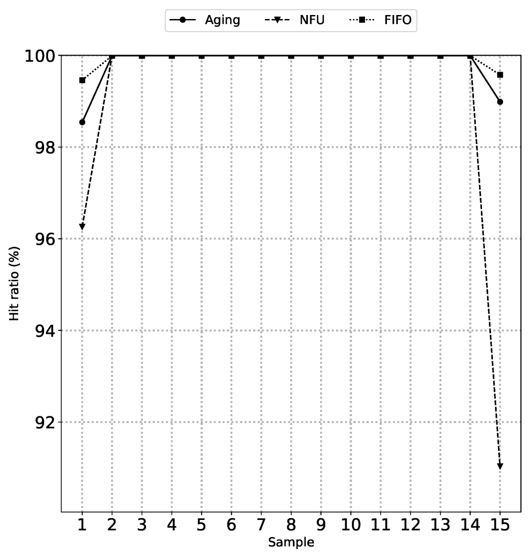


Figure 36 – LU performance evaluation with 48 and 64 pages.

(a) LU application with 48 pages.



(b) LU application with 64 pages.

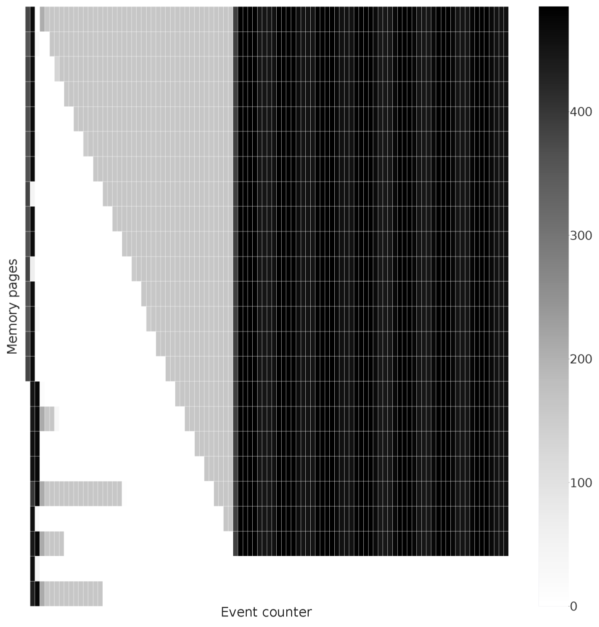


this discussion, Figures 37 and 38 present the slice of the execution of the LU application software-managed cache *heatmap* with these two algorithms and 64 pages. Figure 37 shows that FIFO keeps the top pages for more time, which is ideal, because they are accessed throughout the execution, whereas Aging removes them sooner just to bring them to the software-managed cache again.

In addition, Figure 38 illustrates the end of application execution. With Aging, several pages from the application FIFO pattern region are removed in favor of the top pages. Conversely, FIFO deals with this dispute in a more efficient manner, which makes the pattern region pages more consistent inside the software-managed cache.

Figure 37 – LU application software-managed cache *heatmap* at the beginning of execution.

(a) LU application with 64 pages and FIFO algorithm.



(b) LU application with 64 pages and Aging algorithm.

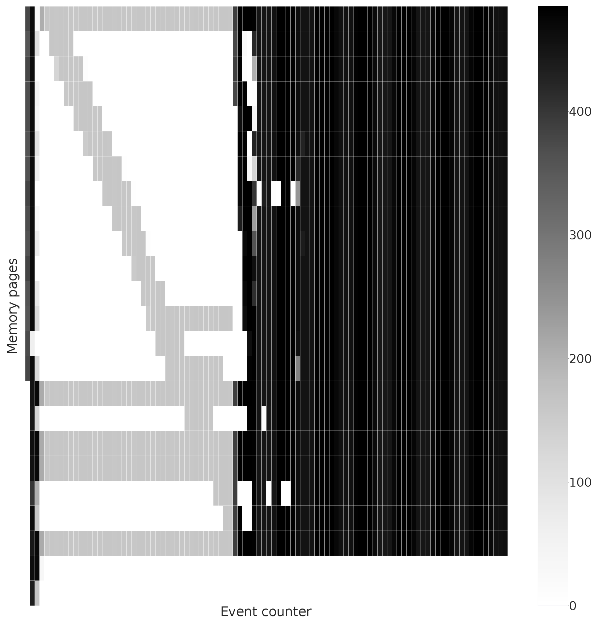
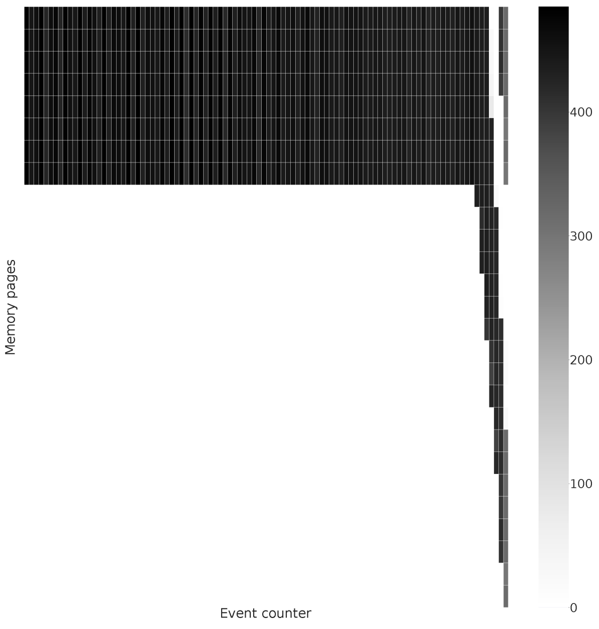
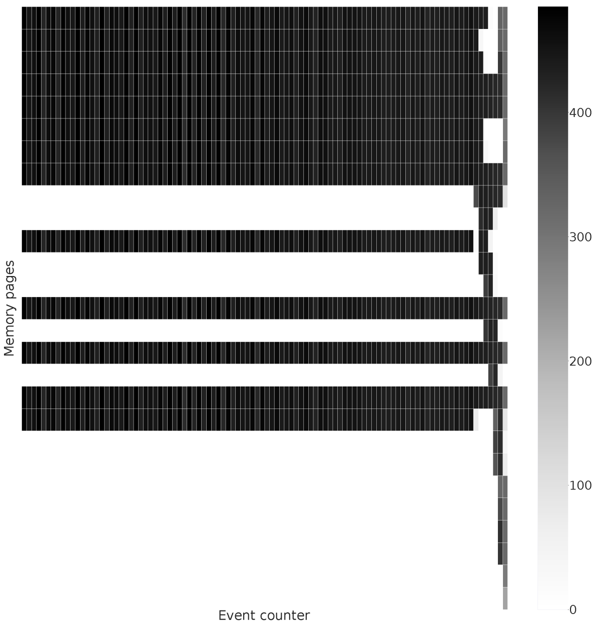


Figure 38 – LU application software-managed cache *heatmap* at the end of execution.

(a) LU application with 64 pages and FIFO algorithm.



(b) LU application with 64 pages and Aging algorithm.



The NFU algorithm presents the lowest hit ratio by a large margin in comparison with other algorithms. Due to several accesses on top pages of the first time *bins*, NFU increases significantly their reference counter. In contrast, other pages that also belong to the FIFO access pattern are low frequently accessed. Hence, pages from the FIFO access pattern dispute between themselves a slot inside the software-managed cache. This dispute significantly decreases the hit ratio.

5 RELATED WORK

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. Traces can also be used to allow more realistic simulations of architectures and applications, providing means to evaluate several possible scenarios and configurations in a feasible time. In this chapter, we discuss related researches that focus on the use of trace-driven approaches to help evaluate and/or optimize software and hardware solutions.

To the best of knowledge, the closest work to ours was proposed by M. M. Rahman K. Nasartschuk & Dueck (2016), 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. The quantity of operations is defined by a given number, and its type[1] is defined based on ratios. Thus, the synthesizer randomly selects a type for an operation based on these ratios and write them into a synthetic trace file. 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. (2015) 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. (2015) 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 and NoC simulators. Moeng, Cho & Melhem (2011) 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 reg-

[1] The type can be classified into allocation, read and store

Table 3 – Summary of related work.

Related Work	**Input Data Type**	**Data Reduction Scheme**	**Context**	**Platform**
M. M. Rahman K. Nasartschuk & Dueck (2016)	Trace	Synthesizer	Memory Management	Simulator
Diener et al. (2015)	*Heatmaps*	None	Shared Memory	Multicore
Moeng, Cho & Melhem (2011)	Trace	None	Cache	Simulator
Keramidas, Strikos & Kaxiras (2011)	Trace	None	Cache	Simulator
Dumas et al. (2017)	Trace	None	Cache	Simulator
Chu & Kise (2018)	Trace	None	NoC	Simulator
Niwa et al. (2018)	Trace	None	NoC	Simulator
Our proposal	**Trace**	***Heatmap***	**Memory Management**	**Lightweight manycore**

Source: the author.

istered in the traces. Similarly, Keramidas, Strikos & Kaxiras (2011) used both CPU and GPU processors to speedup cache simulation. Dumas et al. (2017) 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. NoC is another interesting feature for manycores and can be configured in several ways on a design stage to achieve the best performance. However, simulation for this feature can be very challenging due to a high synchronization cost. Chu & Kise (2018) proposed a trace-driven NoC simulation for FPGA focused on reducing synchronization costs. In the same context, Niwa et al. (2018) focused on predicting the performance of the system based on NoC configurations. More precisely, the work aimed to estimate the number of clock cycles without the need to perform a full system simulation. In contrast to these works, we were interested in evaluating memory management optimizations in a distributed OS running on a lightweight manycore. To do so, we focused on carrying out experiments for a software-managed cache case-study on a baremetal hardware without any cache-level simulation.

Table 3 outlines the aforementioned related work into 4 categories: (i) how they collect information to be used for several evaluations; (ii) which reduction scheme is used to deal with large traces; (iii) the target context where each work is applied; and (iv) what is the target platform. As it can be noticed, these researches provide interesting approaches to: aid the design and optimizations of interesting features for processors, such as caches and NoC; and use *heatmaps* to store and reproduce information. However, they do not focus on distributed OS or baremetal lightweight manycores, and only one of them focus on the memory management service. Therefore, with this dissertation, we focus on a trace-driven approach based on *heatmaps* that helps developers to evaluate and optimize features of a memory management service of distributed OSs for lightweight manycore processors.

6 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 their design and implementation, opening new opportunities for optimizations. A possible approach to study the impacts and trade-offs of different optimizations 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 dissertation, 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. We integrated our methodology in Nanvix and validated it using five applications from CAP Bench. 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. Our results show a very low difference between our *proxy application* and original *heatmaps* with a RMSD below 0.5 and 1.5 for a moderate and maximum *trial factor*, respectively. Furthermore, we carried out a case study using a software-managed cache implementation available in Nanvix. Our methodology enabled us to evaluate several configurations and 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.

BIBLIOGRAPHY

ASMUSSEN, N. et al. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. **ACM SIGARCH Computer Architecture News**, ACM, New York, USA, n. 2, p. 189–203, 2016.

BARTOCCI, E. et al. Introduction to Runtime Verification. In: **Lectures on Runtime Verification. Lecture Notes in Computer Science, vol 10457**. Cham: Springer, 2018. p. 1–33. ISBN 978-3-319-75631-8.

BAUMANN, A. et al. The multikernel: a New OS Architecture for Scalable Multicore Systems. In: **ACM SIGOPS Symp. on Operating Systems Principles (SOSP)**. New York, USA: ACM, 2009. p. 29. ISBN 9781605587523.

BORKAR, S. Thousand core chips. In: **Design Automation Conf. (DAC)**. New York, New York, USA: ACM Press, 2007. p. 746. ISBN 9781595936271.

BOSE, P. **Power Wall**. Boston, MA: Springer US, 2011. 1593–1608 p. ISBN 978-0-387-09766-4. Disponível em: https://doi.org/10.1007/978-0-387-09766-4_499.

BOYD-WICKIZER, S. et al. Corey: An Operating System for Many Cores. In: **USENIX Conf. on Operating Systems Design and Implementation (OSDI)**. San Diego, USA: USENIX Association, 2008. p. 43–57.

BROOKS, D. et al. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. **IEEE Micro**, v. 20, n. 6, p. 26–44, 2000. ISSN 0272-1732.

CASTRO, M. et al. Energy Efficient Seismic Wave Propagation Simulation on a Low-power Manycore Processor. In: **International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. Paris, France: IEEE Computer Society, 2014. p. 57–64.

CASTRO, M. et al. Seismic wave propagation simulations on low-power and performance-centric manycores. **Parallel Computing**, v. 54, p. 108–120, 2016. ISSN 01678191. Disponível em: http://www.sciencedirect.com/science/article/pii/S0167819116000417.

CHU, T. V.; KISE, K. An effective architecture for trace-driven emulation of networks-on-chip on FPGAs. **Proceedings - 2018 International Conference on Field-Programmable Logic and Applications, FPL 2018**, p. 419–426, 2018.

DIENER, M. et al. Communication-aware process and thread mapping using online communication detection. **Parallel Computing**, v. 43, p. 43–63, 2015. ISSN 0167-8191.

DINECHIN, B. D. de et al. A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. In: **International Conference on Computational Science (ICCS)**. Barcelona, Spain: Elsevier, 2013. v. 18, p. 1654–1663.

DUMAS, J. et al. Trace-driven exploration of sharing set managementstrategies for cache coherence in manycores. In: **International New Circuits and Systems Conference (NEWCAS)**. Strasbourg, France: IEEE Computer Society, 2017. ISBN 9781509049912.

FRANCESQUINI, E. et al. On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms. **Journal of Parallel and Distributed Computing (JPDC)**, Academic Press, Orlando, USA, v. 76, n. C, p. 32–48, 2015. ISSN 0743-7315.

FU, H. et al. The Sunway TaihuLight supercomputer: system and applications. **SCIENCE CHINA Information Sciences**, v. 59, n. 7, p. 072001:1—-072001:16, 2016.

GSCHWANDTNER, P. et al. On the potential of significance-driven execution for energy-aware HPC. **Computer Science - Research and Development**, v. 30, n. 2, p. 197–206, 2015. ISSN 18652042.

HOLLIS, S. J.; MA, E.; MARCULESCU, R. nOS: A Nano-Sized Distributed Operating System for Many-Core Embedded Systems. In: **International Conf. on Computer Design (ICCD)**. Scottsdale, USA: IEEE, 2016. p. 177–184. ISBN 978-1-5090-5142-7.

JAHIC, J. et al. A Framework for Non-intrusive Trace-driven Simulation of Manycore Architectures with Dynamic Tracing Configuration. In: **Colombo C., Leucker M. (eds) Runtime Verification. RV 2018. Lecture Notes in Computer Science, vol 11237**. Cham: Springer, Cham, 2018. p. 458–468.

KERAMIDAS, G.; STRIKOS, N.; KAXIRAS, S. Multicore cache simulations using heterogeneous computing on general purpose and graphics processors. **Proceedings - 2011 14th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD 2011**, p. 270–273, 2011.

KLUGE, F.; GERDES, M.; UNGERER, T. An Operating System for Safety-Critical Applications on Manycore Processors. In: **International Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing**. Reno, USA: IEEE, 2014. p. 238–245. ISBN 978-1-4799-4430-9.

LUK, C. K. et al. Pin: Building customized program analysis tools with dynamic instrumentation. **ACM SIGPLAN Notices**, v. 40, n. 6, p. 190–200, 2005. ISSN 03621340.

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

MOENG, M.; CHO, S.; MELHEM, R. Scalable multi-cache simulation using GPUs. **IEEE International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems - Proceedings**, p. 159–167, 2011.

NETHERCOTE, N.; SEWARD, J. Valgrind: A Program Supervision Framework. **Electronic Notes in Theoretical Computer Science**, v. 89, n. 2, p. 44–66, 2003. ISSN 1571-0661. Disponível em: https://www.sciencedirect.com/science/article/pii/S1571066104810429.

NETHERCOTE, N.; SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: **ACM SIGPLAN Notices**. New York, NY, USA: Association for Computing Machinery, 2007. v. 42, n. 6, p. 89–100. ISBN 9781595936332. ISSN 15232867.

NIGHTINGALE, E. B. et al. Helios: Heterogeneous multiprocessing with satellite kernels. **SOSP'09 - Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles**, p. 221–234, 2009.

NIWA, N. et al. An trace-driven performance prediction method for exploring noc design optimization. **Proceedings - 2018 6th International Symposium on Computing and Networking Workshops, CANDARW 2018**, IEEE, p. 182–185, 2018.

OLOFSSON, A.; NORDSTROM, T.; UL-ABDIN, Z. Kickstarting high-performance energy-efficient manycore architectures with Epiphany. **Conference Record - Asilomar Conference on Signals, Systems and Computers**, IEEE, v. 2015, p. 1719–1726, 12 2014.

PENNA, P. H.; FRANCIS, D.; SOUTO, J. The Hardware Abstraction Layer of Nanvix for the Kalray MPPA-256 Lightweight Manycore Processor. **Conférence d'Informatique en Parallélisme, Architecture et Système**, p. 1–11, 2019. Disponível em: https://hal.archives-ouvertes.fr/hal-02151274.

PENNA, P. H. et al. On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. **Brazilian Symposium on Computing System Engineering, SBESC**, v. 2019-Novem, 2019. ISSN 23247894.

PENNA, P. H. et al. Inter-Kernel Communication Facility of a Distributed Operating System for NoC-Based Lightweight Manycores. **Journal of Parallel and Distributed Computing**, Elsevier, v. 154, p. 1–15, 2021. Disponível em: https://hal.archives-ouvertes.fr/hal-03207388.

PENNA, P. H. et al. RMem: An OS Service for Transparent Remote Memory Access in Lightweight Manycores. In: **25th International Workshop on Programmability and Architectures for Heterogeneous Multicores15th International Workshop on Programmability and Architectures for Heterogeneous Multicores**. Valencia, Spain: [s.n.], 2019. (High-Performance and Embedded Architectures and Compilers Workshops (HiPEAC Workshops)), p. 1–16.

SOUZA, M. A. et al. CAP Bench: A Benchmark Suite for Performance and Energy Evaluation of Low-Power Many-Core Processors. **Concurrency Computation Practice and Experience**, v. 22, n. 6, p. 685–701, 2010. ISSN 15320626.

SUEMATSU, H. et al. A Heatmap-Based Time-Varying Multi-variate Data Visualization Unifying Numeric and Categorical Variables. In: **International Conference on Information Visualisation (IV)**. Paris, France: IEEE, 2014. p. 84–87.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. Fourth. Boston, MA: Pearson, 2015. 1137 p. ISBN 978-0-13-359162-0.

UHLIG, R. A.; MUDGE, T. N. **Trace-Driven Memory Simulation: A Survey**. New York, NY, USA, 1997. v. 29, n. 2, 128–170 p.

VARGHESE, A. et al. Programming the Adapteva Epiphany 64-Core Network-on-Chip Coprocessor. In: **International Parallel Distributed Processing Symposium Workshops (IPDPSW)**. Phoenix, USA: IEEE Computer Society, 2014. p. 984–992.

VILLA, O. et al. Scaling the Power Wall: A Path to Exascale. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)**. New Orleans, USA: ACM, 2014. p. 830–841.

WENTZLAFF, D.; AGARWAL, A. Factored operating systems (fos): the case for a scalable operating system for multicores. **ACM SIGOPS Operating Systems Review**, ACM, v. 43, n. 2, p. 76, 2009. ISSN 01635980.