VIRTUAL MEMORY ON A MANY-CORE NOC

ADRIAN CIARAN MCMENAMIN

PhD
University of York
Computer Science
June 2019

ABSTRACT

Many-core devices are likely to become increasingly common in real-time and embedded systems as computational demands grow and as expectations for higher performance can generally only be met by by increasing core numbers rather than relying on higher clock speeds.

Network-on-chip devices, where multiple cores share a single slice of silicon and employ packetised communications, are a widely-deployed many-core option for system designers. As NoCs are expected to run larger and more complex programs, the small amount of fast, on-chip memory available to each core is unlikely to be sufficient for all but the simplest of tasks, and it is necessary to find an efficient, effective, and time-bounded, means of accessing resources stored in off-chip memory, such as DRAM or Flash storage.

The abstraction of paged virtual memory is a familiar technique to manage similar tasks in general computing but has often been shunned by real-time developers because of concern about time predictability. We show it can be a poor choice for a many-core NoC system as, unmodified, it typically uses page sizes optimised for interaction with spinning disks and not solid state media, and transports significant volumes of subsequently unused data across already congested links.

In this work we outline and simulate an efficient partial paging algorithm where only those memory resources that are locally accessed are transported between global and local storage. We further show that smaller page sizes add to efficiency. We examine the factors that lead to timing delays in such systems, and show we can predict worst case execution times at even safety-critical thresholds by using statistical methods from extreme value theory. We also show these results are applicable to systems with a variety of connections to memory.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACKNOWLEDGMENTS

Enormous thanks go to my supervisor Professor Neil Audsley. Similarly I want to thank Professor Andy Wellings and all the other staff of the Computer Science Department at York, as well as my external examiner Professor Peter Pietzuch.

It was an enormous privilege to be encouraged at the start of this work by Peter J. Denning, Distinguished Professor at the Naval Postgraduate School, Monterey, California. Emeritus Professor Paul A. Rubin of Michigan State University also deserves thanks for encouragement over the years of this research. There are many others too who have helped along the way.

My mother Collette McMenamin has been a constant source of support and my daughters Eibhlin and Aine have put up with their father spending more time with this than with them with good grace.

But most thanks of all most go to my partner Lorraine Marshall. Truly none of this would have possible without her love and support and so this work is dedicated to her.

DECLARATION

I declare that all work contained within this thesis is a result of my own investigations, except where explicit attribution has been given. The content of some of the chapters has already been published in:

- Adrian McMenamin and Neil C. Audley: Partial Paging for Real-Time NoC Systems, *11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2015

This work has not previously been presented for an award at this, or any other, University.

1 INTRODUCTION

1.1 THE RISE OF THE REAL-TIME MACHINES

Real-time computing devices are those for which a timely, as well as a computationally correct, outcome is a necessity for success[1] and they are now essential tools of everyday life: for instance, the World Bank reported that, in 2016, there were 101.6 mobile telephony subscriptions - mobile phones probably being the most pervasive of real time computing devices - for every 100 people globally, with 122 per 100 people in the United Kingdom[2]. Analysts predict a continued decline in sales of general computing devices, leaving hardware and software manufacturers alike looking to mobile (real time) devices, as well as the related software, to support industry profits and growth (Figure 1.1).

Consumer expectations for these devices are high and rising: a demand for what have been called "mobile supercomputers" [13]. Raw speed, though, is not the only demand as long battery life and silent operation are also often essential.

The rise of the real-time machines is not limited to mobile phones. The "internet of things", with devices connected to data-gathering sensors and responding in real time to events, is expected to grow extremely rapidly (cf., Figure 1.2). Aerospace and automotive manufacturers are increasingly reliant on embedded computing for tasks which range from the safety critical, such as management of flight control surfaces, to the merely business critical, such as in-flight entertainment. Making efficient, including cost-efficient, and effective use of hardware and software is an essential matter.

As the complexity and range of embedded computing tasks increases designs based on isolated systems monitoring and controlling one function are unlikely to be cost-effective: though up to now this has been the typical way in which embedded devices have been employed. In future it is likely computing resources will need to be shared in all but those cases where total physical isolation is regarded as essential to security.

In this thesis we explore the ways in which real-time devices, particularly those based on network-on-chip designs, can effectively deploy the virtual memory abstraction[3]. Virtual memory has been central to general computing's approach to sharing memory resources and

1 We give a fuller definition of what we mean by "real time" in Section 1.2
2 http://data.worldbank.org/indicator/IT.CEL.SETS.P2 - accessed 3 October 2017.
3 We discuss virtual memory more fully in Chapter 3.

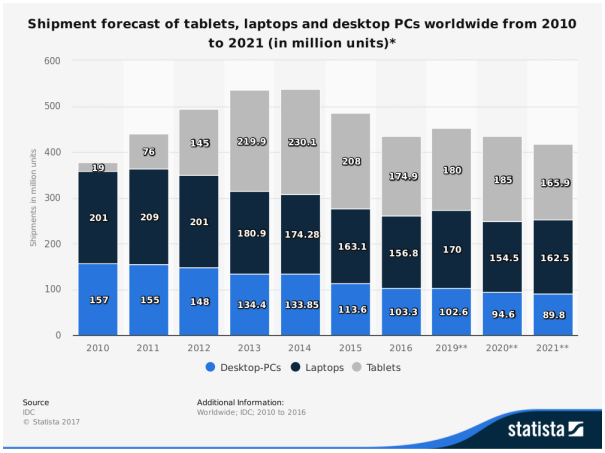

Figure 1.1: Forecast of long-term decline in sales of general computing devices (generated via Statista, 9 November 2017)

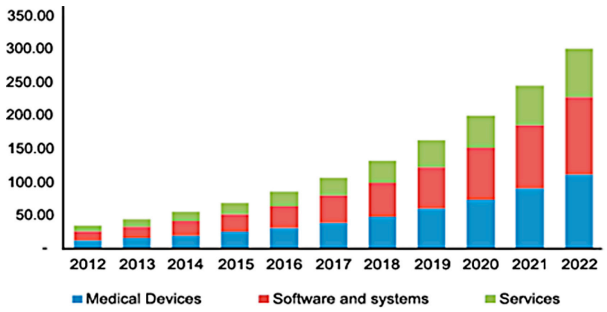

Figure 1.2: North American Internet of Things in healthcare, projected market growth by component (billions of US dollars) 2012 - 2022 (via [59])

enabling multi-computing for decades but its use has generally been avoided by real-time developers. We will explore a novel scheme for the use of virtual memory which delivers virtual memory's well-understood advantages while also addressing the disadvantages that have deterred deployment in real-time and embedded domains.

In the rest of this chapter we outline why the many-core designs seen in network-on-chip systems are increasingly used for embedded and real-time systems before outlining a research hypothesis and then a plan for the rest of this thesis. In 1.2 we define what we mean by real-time computing. In 1.3 we set out the context of increased demands on real-time computing devices and in 1.4 we present our research hypothesis. In 1.5 we describe the structure of the rest of this thesis.

1.2 REAL TIME COMPUTING AND NETWORK-ON-CHIP SYSTEMS

As described above, a real-time system is one where the success of the computational process does not just depend on the logical outcome but also on it being completed in a timely manner. Three broad categories of real-time systems exist [29]:

- *Hard real time*, where the whole system is rendered a failure if the computation fails to complete inside the deadline. Such computations might typically be found in avionics, where a failure to complete the computation in time would threaten operator safety.
- *Firm real time*, where an individual computation is rendered a failure if not completed inside a deadline but where the application as a whole may continue. In such systems a limited number of failures to complete processing by the deadline are acceptable even though processing after the deadline is of no value. Such failures are likely to result in a decline in the quality of service. A typical firm real-time system would be a video decoder and playback device where delayed frames cannot be shown out of order, or allowed to delay playback, and are skipped at the cost of picture quality.
- *Soft real time*, where computational results that have missed deadlines diminish quality of service but may still be used. Typically these can be found in online transactional systems.

Applications that "run on NoCs are typically composed of periodic or sporadic tasks that communicate their results by sending messages over the network" while "[a]pplications that are likely to benefit from networks on chip are complex embedded applications with many communicating subsystems ... [a] significant proportion of embedded applications have real-time requirements" [111]. Here we will simulate this domain of complex communicating

subsystems by running multiple copies of benchmarks designed for real-time embedded systems (with communication considered in terms of traffic to memory) and we will gather data on timing performance to draw conclusions about real-time constraints and behaviour.

An embedded computing system is one which forms a part of a larger machine. Often embedded software is concerned with only one aspect of the machine's function, in other cases it may perform multiple tasks. In this thesis we generally consider the case where an embedded software process performs one task but shares hardware resources with many other pieces of embedded software. This model is a growing domain for embedded systems as computing systems relying on information from sensors replace mechanical and electro-mechanical machines and systems.

Designers of real-time computing devices need to deliver systems which are both predictable and efficient. For real-time systems a fast *average* execution time is of no particular use if the *worst-case* execution time (WCET) - the maximum time any invoked process could require to complete [29] - is many orders of magnitude greater than this average as the reliability or even safety of the system depends on this worst case. Additionally a large gap between the average timing and the worst case also implies a system that will often be idle and so represent a poor return on investment.

1.3 INCREASINGLY COMPLEX DEMANDS ON REAL-TIME AND EMBEDDED DEVICES

Real-time systems are often deployed in an embedded context. We can expect a further, and rapid, growth in embedded and real-time systems deployments, as personalised or wearable devices and medical aids, and through self-driving vehicles and in commercial aeroplanes [32]. These systems will have to deal with tasks of increased computational intensity placing heavy demand on CPUs and memory [75].

Consumer expectations are of ever-faster devices, but it remains difficult to squeeze anything like theoretical performance out of today's processors. For while "Moore's Law" [115] of exponential increases in transistor numbers per integrated circuit (IC) seems likely to remain a valid rule of thumb for some while longer, the phenomenon of "Dennard Scaling" [26] which kept the power densities of ICs under control even as they were driven at faster and faster speeds, has ended. Programmers no longer get what has been called a "free lunch" of faster machines without the complication of handling multiple cores running in parallel.

To drive the performance increases which in turn keep the market healthy, system designers and programmers have to deal with systems with multiple processing units. And

these processors need to be physically close to ensure speed: electromagnetic signals take a nanosecond to travel 30 centimetres and so even the wire lengths we might expect to see inside a commodity device are not negligible for processors running at high frequencies. But successfully operating computing devices which use many processors placed closely together on a single piece of silicon necessitates being able to master systems with steep memory hierarchies: for while on-chip memory can be supplied it is at a significant cost per bit and at the expense of processor capacity or capability.

The growing difference between the performance of ever-faster microprocessor circuits and memory access times has created the so-called *memory gap*. The gap is such that, as a direct result, the Large Hadron Collider 'beauty' (LHCb) experiment is estimated to lose 50% of its computing cycles to stalls caused by memory misses [4]. In general computing this performance problem has been tackled through the use of caches [105], but in real-time systems caches are a major, if not potentially *the* major, source of timing unpredictability, particularly for multi-core systems [45]: for that reason they are generally avoided in real-time systems and the question of how to build a multi- or many-core real time system that can deliver predictable and efficient timing outcomes remains an open one [64, 165].

Modern many-core compute devices, whether based on network-on-chip (NoC) or graphics processing units (GPU), do have small amounts of low-latency local memory on chip which can come close to matching the performance of the core's register file [159]. If memory accesses could be limited to just this area then a real-time system would be easier to manage: "The simplest way [to provide memory], and one to be used if possible, is to provide in the system enough memory for each process to have its own unique area." [9]

But such a low-latency nirvana is almost certainly out of reach: the last thirty years have seen computing devices handle more complex inputs (such as structured documents) and outputs (such as full colour graphics), all of which require more code and storage space. It is also not just the complexity of the tasks being undertaken by computing devices that is driving up program size, but also shifts in programming languages being used to allow programmers easier access to abstractions [97].

Our proposal is founded on the need of today's programmers to access a range of memory resources much greater than that than can be stored simultaneously in locally available low-latency storage. Because of this systems designers and implementors have to have some mechanism that allows them to replace code and data in high-speed memory at a given time with other code or data (i.e., a *replacement policy*).

We advocate the use of virtual memory not just because it is a well-understood abstraction that can support an algorithmic replacement policy and a large address space, but also because it brings other advantages in terms of security and resource sharing (we discuss this further in Section 3.1). In particular we consider paged virtual memory (where the address space

is divided into equal sized blocks called *pages*), both because this is the most commonly hardware-supported form of virtual memory but also because it avoids external fragmentation of a limited memory space that the alternative approach of segmentation might risk (this is discussed further in Section 3.1.3).

The model we consider in Chapters 4 and 5 is of each core only running a single thread of execution. With each NoC core only having a very limited local memory resource multiple threads of execution (if using different virtual memory address mappings) would further increase competition and possibly add to system delay as more threads are blocked waiting for memory.

1.4 RESEARCH HYPOTHESIS

Using virtual memory risks very poor performance because of the phenomenon of *thrashing*, where computing time is lost to the costs of the need to share the small amount of available memory across multiple programs [48]. Further real-time programmers have typically avoided its use because analysis is complex as performance depends on the past string of memory references.

Thrashing is not a new challenge: in the 1960s an earlier generation of programmers, experimenting with time-sharing computers and using virtual memory as a way to increase the range of addressable memory while executing programs generated by the first generation of high-level languages experienced severe performance difficulties as a result of thrashing. The solutions we propose here draw on the lessons learned at that time, but where an earlier generation of researchers sought a better *page replacement algorithm* as the answer to thrashing, our conclusion is that fast memory on many-core systems is so limited that we must also rely on a better *page loading algorithm*.

We propose to enable virtual memory [50] for such systems, using a novel form of paging which, instead of transferring whole pages across the memory hierarchy, concentrates on a minimal transfer of memory inside a paging scheme [110] - this is what we call we call *partial paging*. In this scheme hard faults (i.e. attempts to access a memory address in a page that is not present in local memory) still impose a significant penalty of lost cycles, but the cost of paging in and out of memory is amortised across multiple memory reads and writes as instead of a whole page being loaded on demand only a part (16 bytes aligned on the 16 byte boundary of the faulting address) is loaded. This avoids unnecessary transfer and limits congestion in the memory interconnect, which we find to be a significant determinant of program performance. Accessing other addresses in the page may raise a “small fault”

causing the loading of a further part of the page. This system necessitates that each access to a page is checked to see if the part of a page being sought is present and we suggest a bitmap is used to record which parts of a page have been loaded.

One of the lessons of the 1960s was that maintaining a program's *working set* in low latency memory was the most effective way of limiting paging and thrashing [47]. Much research at that time and into the 1970s and later concentrated on effective ways to maintain the working set of pages in fast memory. Innovation continued into the 1990s (and beyond), for instance with the 2Q algorithm in Linux [88]. Here, though, we establish that better performance in embedded and real-time many-core systems is unlikely to come from an improved page replacement algorithm but from a reorientation towards better management of the internals of the pages.

We cannot eliminate the complexity caused by the dependence of performance on past memory references, but instead we will argue that in many-core systems static analysis of timing is, in-effect, ruled out by the system's overall complexity and that we are driven, in any case, to use statistical methods to determine worst-case timings. We believe we can show that partial paging reduces the statistical complexity of the system and this lowered complexity also delivers better WCETs. In this context entropy means (in analogue to its meaning in thermodynamics) the range of available macrostates of the system, and we will show that increased entropy (specifically in the connection between cores and the global memory pool) increases the range of expected completion times for applications and hence increases WCETs we can expect. We discuss entropy in more detail in Section 4.9.

Fifty years of research and engineering have ensured that general computing page replacement policies, which often mix different approaches, are robust in the context of general computing devices [109] but for real-time computing paging has often been avoided in the hope of increasing timing certainty. With the decreasing cost of memory, much recent research in general and high-performance computing has concentrated on using large page sizes, not least to minimise TLB misses [18] but in our case we return to earlier research findings which emphasise the efficiency of small pages on small systems.

In this thesis, therefore, we intend to explore the following hypothesis:

Paged virtual memory can be implemented in an effective manner for a many-core real-time system in a way that minimises the worst case execution times for real-time tasks: not through the traditional means of a better page replacement algorithm, but through a more efficient and real-time-appropriate 'partial page' loading algorithm. Furthermore, this partial paging approach is most effective with smaller page sizes and, by limiting queuing for memory service, partial paging reduces entropy in the system and thus increases certainty about timing.

In judging success we will compare the performance (as measured by average and maximum execution times) of partial paging with a "traditional" whole-page approach, considering different page sizes and page replacement policies.

1.5 THESIS STRUCTURE

To illustrate and elucidate the above points, the remainder of this thesis will be structured as follows:

In Chapter 2 we review the literature that covers many-core systems in general, and network-on-chip systems in particular along with questions of memory management in a many-core system.

In Chapter 3, beginning on page 53, we look in depth at the theory of virtual memory and consider what we mean by thrashing before considering the behaviour of simulated systems using different page replacement algorithms and simulating the execution of a benchmark from the PARSEC suite. These experiments build the case for using a partial paging approach to the use of virtual memory as they show that a better page replacement algorithm does not significantly improve performance.

In Chapter 4 on page 71 we describe a simulated partial paging system, with 1KB pages and using a CLOCK page replacement policy. This shows significantly improved performance over a traditional full-page paging approach. We also consider the results of a simple load control mechanism, which is shown to deliver better results for some situations. In this chapter we develop an entropy model of the system and consider the impact of entropy on the range of completion times before using extreme value theory to make predictions about probabilistic worst-case execution times.

In Chapter 5 on page 141 we consider how a partial paging system can be optimised, considering smaller page sizes, FIFO page replacement and, finally, alternative memory connection mechanisms. We show that smaller page sizes can deliver improved performance, especially when we consider safety-critical limits, but that very small pages raise the fault rate to such an extent that any advantage may be lost. We also show that while FIFO's typical performance may be similar to that of CLOCK, at safety-critical limits it can be much slower. We also show that FIFO's initial high degree of predictability is lost due to the injection of entropy into the system. We show that alternative connection mechanisms tend to deliver similar performance to the Bluetree memory tree we model in most simulations.

In Chapter 6 on page 201 we offer our conclusions and our argument as to why the research hypothesis has been proved. We discuss the contributions this research has made and areas for

future work and research, including using open source hardware to build cores that support partial paging.

Appendix A on page 207 briefly outlines some experimental work we undertook to consider using partial paging to support coherent parallel code. Here we find that the congestion in the memory interconnect makes using large numbers of cores self-defeating.

Appendix B on page 213 offers more detail on the translation of traces to the XML we used in the simulations. Further appendices offer additional detail on results referred to in the main text. The bibliography then follows.

2 BACKGROUND AND RELATED WORK

In this chapter we will review the literature around many of the questions that are relevant to memory management on a many-core network-on-chip (NoC) system. In doing so we will first examine the motivational factors that have led to the creation of NoCs, before considering practical NoC implementations as well as alternative, but closely related, designs such as those found in graphics processing units (GPUs).

In this chapter we only briefly discuss virtual memory, paging and page replacement policies, considering those in depth in Chapter 3.

In Section 2.2 we discuss the problems of increasing power density and in 2.3 we consider the limits of bus-based connections. In 2.4 we discuss network-on-chip designs and in 2.5 we consider alternative many-core designs. In 2.6 we discuss the problems of memory management in NoCs and in 2.7 review page replacement policies. In 2.8, we discuss the limitations on parallel speed-up imposed on many-core devices by the need to execute code in serial and we present a summary of this chapter in 2.9.

2.1 THE PROBLEM: A NEED FOR FASTER REAL-TIME COMPUTING IN A COMPLEX ENVIRONMENT

Commercial pressures as well as scientific enquiry continue to drive the search for faster computing hardware and more efficient software, and there is a significant commercial imperative behind delivering faster computing and what has been described as advance through "technology jumping" [51]. However limitations of CMOS technology mean it is no longer possible to make ever-faster single compute core CPUs [95, 153] and instead chip manufacturers have concentrated on making multicore devices and software developers have been tasked with finding and exploiting parallelism [122]. For real-time systems technological advances open up the prospect of developing systems that will "transform how humans interact with and control the physical world" [136] but multicore and many-core[1] devices pose some potentially significant problems for real-time tasks, such as:

1 There is no hard definition that separates these two categories but we follow e.g., [156] in taking many-core to mean "10s" and upwards of computing cores.

- Such devices typically have only relatively small amounts of fast memory available (as cache or scratchpad memory) to each core.
- What code or data is available in such local memory is dependent on the often complex interaction of past references and memory replacement policies.
- Replacing any local resource is likely to involve interaction with other cores as paths to external resources are likely to be shared, and thus access contested, and may be of different lengths.

All of these issues make it difficult to deliver efficient real-time services and to predict WCETs in particular. In the rest of this chapter the genesis of, and problems facing, network on chip systems, while in Chapter 3 we consider the problems of virtual memory as a solution to these issues more closely.

2.2 HITTING THE POWER WALL

Since the late 1950s the number of transistors in central processing and similar units has doubled every 12 - 24 months. Since the introduction of the integrated circuit this has meant some combination of an increase in the size of silicon wafer and an increase in the density (through a decrease in the size) of the transistors etched on to the wafer [115].

The barriers to the continuing diminution in size of transistors are daunting [127] but have generally been surmounted by manufacturers. However, what was a long-sustained parallel increase in computing speed has ended with the failure of the related phenomenon of "Dennard Scaling" [26] which, by allowing the lowering of the voltage of operation, delivered a constant energy density on the silicon even as the transistor size fell.

The transistor supply voltage (normally referred to as V_{dd}) has shown only minimal reduction as component size has decreased [53, 77]: the result has been to dramatically slow the rate of decrease of energy required to drive a transistor. Dynamic power usage varies quadratically with V_{dd} and so power density has risen, heating chips up and increasing static power losses and so further increasing power density and threatening thermal runaway [92, 116].

By 2004 - 2005 manufacturers found it was no longer possible to continue to make faster single CPU devices [95, 153]. Instead, as illustrated in Figure 2.1 on the next page (from [62]), designs shifted towards multicore systems, and typically today's general-use computers have several processors or computing cores connected via a time-division multiplexed bus or some similar shared medium of interconnection.

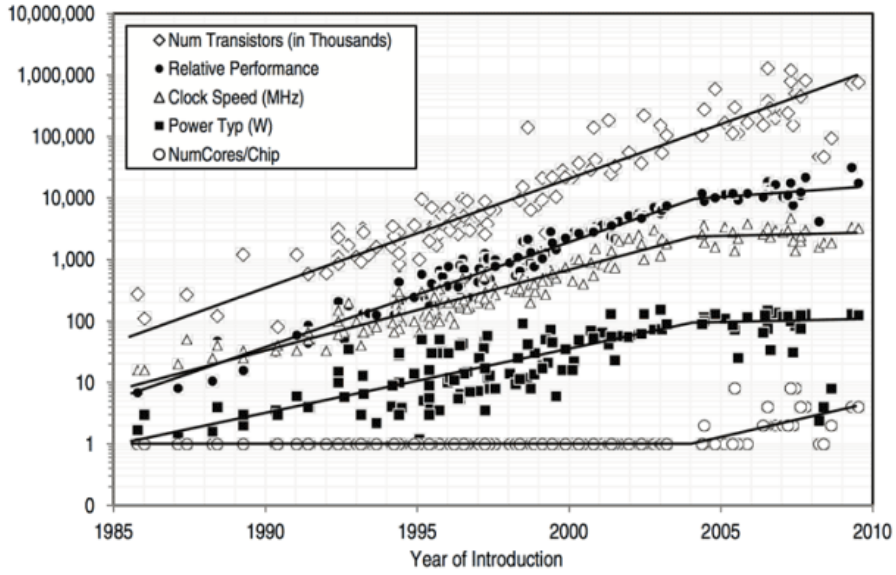

Figure 2.1: As increases in processor performance diminish, designs switch to multiple processor systems

Frequency varies approximately linearly with voltage of operation, so we can replace a single high frequency, high voltage processor with two processors running at half the voltage (and thus half the frequency) and, for a sufficiently parallel load, approach the same computational outcome while using half the power.

2.3 FROM BUSES TO NETWORKS-ON-CHIP

Using multiple processors raises the question of how they connect to each other and to external resources (such as memory) and how such a connection deals with any issues of contention. Buses have traditionally been deployed and, as they enable signal sharing, they have been used on systems with small numbers of processors to implement, in hardware, the "snoopy" caching paradigm [72]. This ensures efficient use of high speed caches and so compensates for the fact that having a shared medium means only one processor may "master" the bus at once. Snoopy caches use this broadcasting of memory reads and writes to ensure that the caches of all connected processors are coherent. This has, in turn, enabled the development of tightly coupled symmetric multiprocessing (SMP) systems which have proved highly effective as a general computing design [143].

But buses are limited as connection media and bus-based designs are not a sustainable means of delivering increasing computing performance as the number of processors connected

on a bus grows. The one-at-a-time requirement is an increasing hindrance, particularly as bus communications have a non-trivial set-up time. Further, the requirement that signals on the bus reach all processors makes buses power hungry and, as they grow in size and increase in capacitance, slower [22].

Taken together these factors impose a practical limit on bus size of around 20 processors [7] and so the limits of buses as connection media have long been recognised [8, 25].

Network-on-chip (NoC) designs are a response to the inability of buses to scale. Instead of using a bus, NoCs use packet networks embedded in the silicon itself. By placing multiple processors on a single wafer, NoCs exploit the technological advances in yield and miniaturisation that have given us bigger processors with more transistors and, as NoCs combine on-chip routers with the processors, they can multiplex packetised communications, potentially lowering latency and allowing for greater parallelisation in communications [43].

A yet more radical alternative solution would be to find new materials or new states of existing materials with which to build high density memories on-chip, but despite some hopeful signs there is still no immediate prospect of this happening at a commercial scale [30]. NoCs are, however, viable as a platform for a future generation of processors and are already being used commercially. Many questions, though, about what constitutes effective and efficient systems software for such systems remain open, especially when we consider many-core systems.

2.4 NETWORK-ON-CHIP HARDWARE

Network-on-chip devices are not standardised, may be designed for a specialised domain of operation and are likely to employ proprietary technology rather than open standards [46]. So here we will concentrate on the general characteristics of NoC devices, while illustrating these with some specific NoC implementations.

2.4.1 Network topologies

Many different network topologies are discussed in the literature, such as meshes, tori, trees and hypercubes, as well as ad hoc structures. The mesh, torus and hypercube illustrated in Figure 2.2 (respectively as items a, b and d) are shown here as *direct networks* where the computing node is combined with a router (also referred to as a *switch* in the literature). The *fat tree* illustrated in Figure 2.2(c), which is based on the SPIN network [6, 10], employs dedicated routers and is not a direct network (i.e., it is an *indirect network*). This is called a

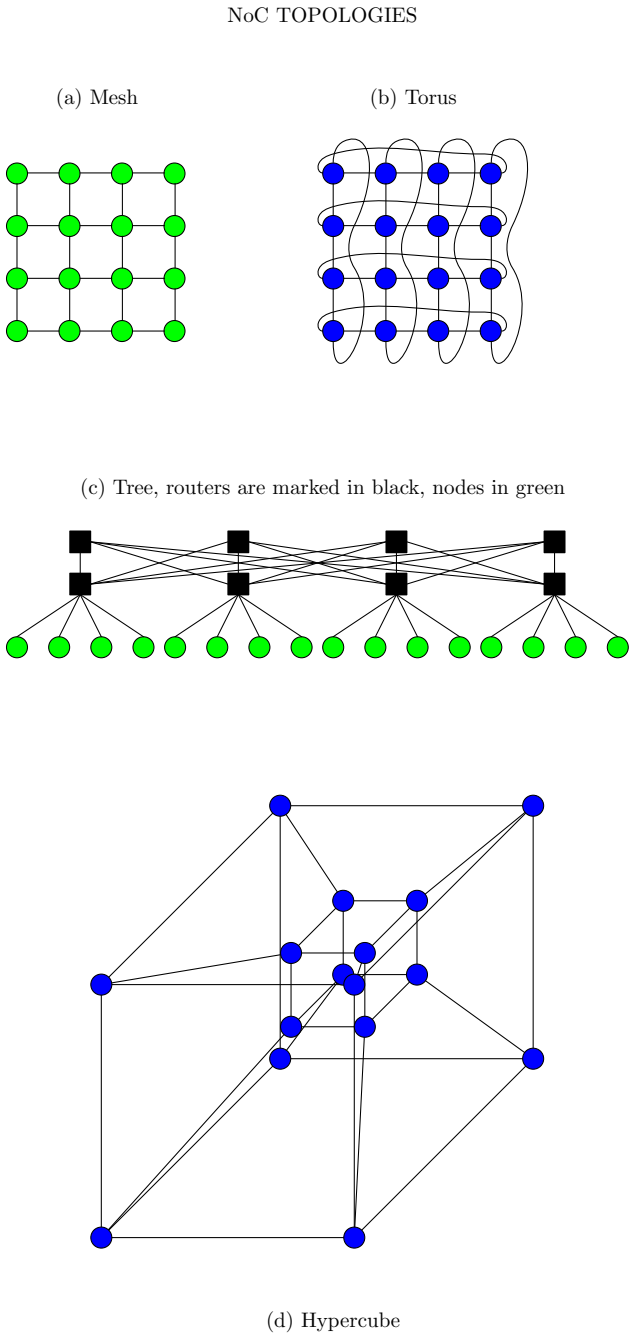

Figure 2.2: Typical NoC topologies

"fat" tree because the bandwidth is higher closer to the tree's roots, so reducing the risk of congestion [102].

In direct networks the *node degree* measures the number of channels (which are illustrated in Figure 2.2 as lines) connecting a node with its neighbours, while the *network diameter* is the maximum distance between two nodes in a network. A network is said to be *regular* when all nodes have the same degree and *symmetrical* when all nodes look alike. A network's *bisection width* is the number of channels that would be cut if the network was divided into two equal numbers of nodes. The bandwidth of these channels is referred to as the *bisection bandwidth* and may be cited as a measure of the performance of a given network topology [46].

Routing packets through the network expends energy and longer routes add to communication latencies, so there is a performance advantage for networks with small diameters and large bisection bandwidths. For manufacturers this is balanced by the high cost of producing devices with many links and routers. Simple network designs are also the easiest to extend. For instance, as Dobson points out [52], while in a hypercube average communication distances only grow $\propto \lceil \log_2 n \rceil$ (where n is the number of dimensions), as the node count grows as 2^n, extending this topology by adding a further dimension requires changing node degree for every node. Yet in a mesh one need only add new nodes at the edges. Meshes are also relatively easy to fabricate on a two-dimensional slice of silicon, compared to the generally higher performing topologies such as tori and hypercubes [69]. In [162] it is estimated that a 2D torus increases wire length and wire congestion by a factor of two in comparison to a mesh design.

Meshes are thus the most common design and are used in even the most powerful of NoC systems [61]. But meshes are neither regular nor symmetric as there is a discontinuity at the boundaries of the mesh. This adds to the energy cost of a mesh network and may cause a local hotspot of traffic at the centre of the network or at edges connected to external devices, such as memory controllers. The asymmetry matters when we consider timing questions, as differing route lengths imply different connection times. The Bluetree memory tree, discussed in 2.4.6, addresses this by using a non-mesh design for memory interconnect.

Ad hoc network topologies may be favoured for *application specific integrated circuit* (ASIC) devices as, with the application already defined, knowledge of traffic patterns means that redundant nodes may be removed. This makes the device cheaper to produce while reducing power consumption and maintaining performance at the expense of limiting its general usefulness. The Æthereal architecture, for instance, allows designers to specify irregular networks when building NoCs for consumer electronics and similar domains [73]. This removal of generality may limit the usefulness of such designs when considering systems software and here we primarily consider what are nominally mesh-based homogeneous NoCs.

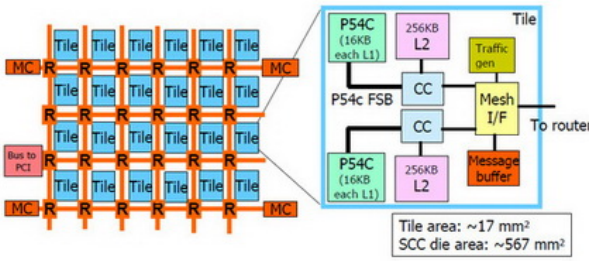

Figure 2.3: Node (tile) on the Intel SCC

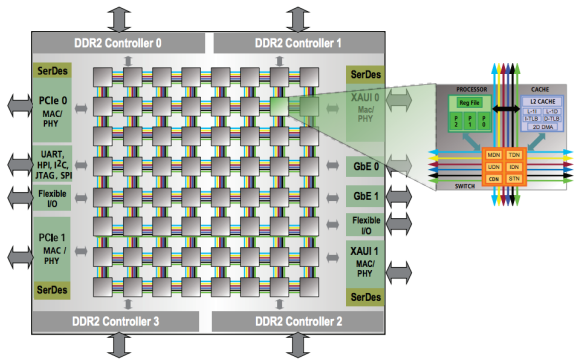

Figure 2.4: Tilera TILEPro64 schematic

2.4.2 The nodes

Nodes on a NoC may vary in their constituents. The Intel Single-chip Cloud Computer (SCC) [108], a 48 core experimental system (illustrated in Figure 2.3, from [108]), groups two processors together in each node (referred to as a tile), along with two level 2 caches, a mesh interface (which connects to a router) and a fast 16KB SRAM message passing buffer. In the TILEPro64 [21], each of the 64 nodes consists of a processor, cache memory and a switch (router) which connects to five different mesh networks, each carrying a different class of traffic (Figure 2.4 from [21]).

The Sunway TaihuLight Supercomputer [61] uses a heterogeneous design to deliver what was claimed to be the most powerful supercomputer of 2016, with its SW26010 processor containing four core groups (CGs). Each CG contains a memory controller, a fully-capable "management processing element" RISC processor and a 64 element cluster of limited-capability RISC processors ordered in a mesh. All CGs are linked as a NoC and each CG is linked to main memory through its memory controller.

The TILEPro and the SCC include caches in each tile, but use them differently. The TILEPro explicitly supports cache-coherent memory access to off-core DRAM by all cores through the strict partitioning of memory, while the SCC in its default setting uses its off-core level

2 (L2) caches to speed access to private memory only. Each SCC tile includes a message passing buffer (MPB) and all processors can use the shared memory it supports as a means of communication: all can write to any address in the shared 384KB (i.e., 48×8KB) address space and the processors, based on second-generation Intel Pentium designs, support an additional instruction to invalidate on-core level one (L1) cache lines (which cache the MPB memory) and an additional "`test&set`" register to support atomic locking.

Across the SCC there are eight voltage domains: one for the on-chip memory controllers (see 2.4.5), one for the mesh and six for the tiles (i.e., four tiles per voltage domain). Switching voltage in any of these six domains incurs a significant latency of ∼1ms (500,000 cycles). As a faster, if less powerful, alternative, frequency may be set for any one tile or for all eight cores in a voltage domain.

In each tile in the TILEPro the cache, processor, switch and the five networks are connected via a crossbar so that traffic may flow between all. The designers of the TILEPro say the use of five networks is only marginally more expensive than using one, as the majority of the cost of the network is in buffer space and that the different classes of traffic would all require separate buffering.

2.4.3 Packets and switching

Routing and switching in NoCs are generally handled by proprietary hardware and routers, or router-processor combinations and are often referred to as *IP blocks* (from "intellectual property") for this reason.

The switching methods employed in NoCs favour small packet sizes - of bits rather than kilobytes - and although the speed of on-chip communication may be relatively high, there is a premium on avoiding latency caused by communication. Contention and congestion can spread through a NoC as a result of the switching methods chosen: if one node on the network is subject to heavy traffic from another then even unrelated nodes could be affected if routing paths coincide. Larger packets may block the network for longer, increasing average latency. Routing packets around a hotspot may be possible, but could increase the worst case execution time of running software. In [167] it is suggested that, for a given algorithm with a known total inter-node communication demand, the different factors point to breaking communications down to an optimised packet size rather than seeking to send as much data as possible in a single packet.

Routing memory requests through the on-chip mesh ensures there may be many routes between any core and a memory controller at the edge of the NoC, but also adds to the problem of calculating WCETs as different routes will be of different lengths and complexity

[12]. Creating a separate mesh for memory packets means memory requests are no longer in competition with other traffic but there will still be contention amongst the memory requests themselves.

2.4.4 Routers' and on-chip networks' power consumption

NoCs are used to overcome the "power wall" problem, but a NoC's on-chip infrastructure is quite likely to use significant amounts of power itself. In [16], a relatively early study of power use in NoCs, the authors suggest standby power usage of on-chip routers could be around 50% of power usage under traffic and that the routers use around 3 - 5 times more power than the on-chip wires. A more recent paper, [141], reports that, for various real-world NoC designs the interconnect infrastructure can use between 10% and 36% of total available power if all cores are being fully utilised. The authors state that routers typically cannot be switched off even if cores are (as they need to be available to route packets) and that these proportions rise even if opportunities are found to make cores 'dark' to meet power constraints as a result. Further, the authors outline how lowering voltages and smaller technology processes increase the share of static power demands in on-chip routers.

2.4.5 Off-chip communication

On-chip networks show significantly lower latency than traditional bus or ring networks, but at the edges of the chip these signals must interact with a world filled with legacy devices. Latency here remains problematic for computing efficiency. In many commercial designs high speed device controllers are placed on the chip itself as a means of minimising latency. In the Intel SCC four DDR3 memory controllers are placed on the edge of the chip, while a field programmable gate array that provides an interface to PCI Express (PCIe) devices [34] is placed immediately "off package". In the TILEPro64 there are four DDR2 memory controllers, as well as four PCIe interfaces and four network interfaces and general input-output controller support [162] - all on the chip. One of the five networks on the chip, the "I/O Dynamic Network", handles input-output device communication. Another network, the "Memory Dynamic Network", handles external memory requests, which come exclusively from the TILEPro's cache blocks. In both devices the number of internal hops requests must make to reach the relevant memory or device controller has a measurable effect on latency. In the SCC latency is of eight cycles per mesh hop, while in the TILEPro each hop requires one cycle with an additional cycle if the packet needs to change direction.

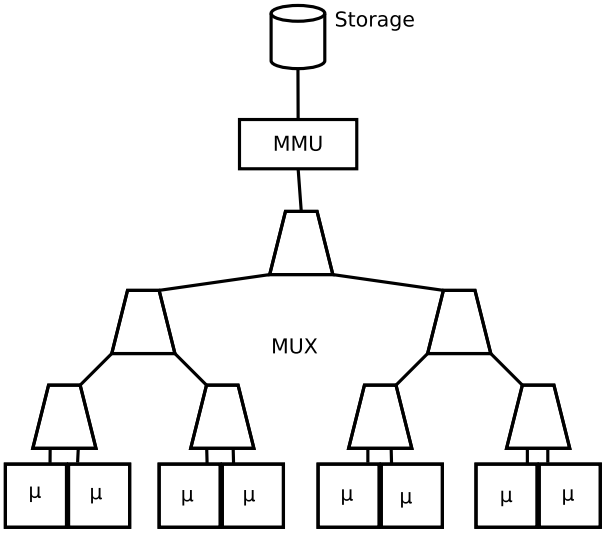

Figure 2.5: Bluetree memory tree

2.4.6 Shared memory trees

The gap between CPU and memory controller performance is growing and can be decisive in overall system performance. While placing memory controllers directly on NoC silicon allows direct connection to the on-chip network it also means router and other latencies matter in determining overall performance [66]. In the examples cited above different cores are different distances away from memory controllers and so even the fastest routes take different amounts of time. An alternative is a separate network which connects to all NoC tiles to grant equal access to shared, off-chip, memory such as Bluetree [12, 131, 66, 144]. Here requests for external memory negotiate a series of 2:1 multiplexors down a tree where a memory management unit (MMU) lies at the root. Each tile is (if the tree is set up in this way) an equal 'distance' from this MMU. See Figure 2.5 for a representation of an 8 core NoC connected to a Bluetree shared memory tree.

We discuss the memory connect in more detail in 4.2.4 and test alternatives to the Bluetree model in 5.5.

The arbiters on Bluetree can be programmed at runtime to enact a priority scheme, allowing this approach to be used with mixed criticality systems.

2.5 ALTERNATIVES TO NOCS: GRAPHICS PROCESSING UNITS

In recent years graphics processing units (GPUs) have emerged as an alternative method of handling massively parallel computing loads. GPUs offer hundreds or even thousands of

cores, though at a lower frequency than would be expected with a CPU, and can deliver superior performance for highly data parallel tasks [101]. Heterogeneous combinations of GPUs and CPUs have been studied and used to maximise computing performance [114], including tackling problematic levels of power consumption by GPUs [155].

GPUs generally have a different form of memory hierarchy from CPUs. In [121] NVIDIA describe a how a 512 core GPU is divided into 16 *streaming multiprocessors* (SMs), each of 32 cores. Each SM shares 64KB of memory (which is typically partitioned between shared memory and cache) and each SM also shares a register file (RF) of 32,768 32-bit registers. In [159] the authors point out that later NVIDIA designs significantly increase RF size per SM but hold or decrease level 1 data cache size and that thrashing of the cache is likely to be a significant problem.

In [68] the authors note that future GPU designs will have to rely on greater amounts of on-chip shared memory to avoid costly accesses to off-chip global DRAM. Currently, they state, applications are tuned to match the GPU design and its limited supply of on-chip memory, and that this is becoming more complex as GPUs are more widely deployed and used for general computing tasks.

2.6 MANAGING MEMORY AND SECONDARY STORAGE

At the heart of the problems we consider in this thesis is how to best manage systems which have only small amounts of local quickly-accessible memory but which need to execute programs that are stored in larger, but slower-to-access storage such as physical (volatile) memory or secondary storage.

In the classic studies of the memory hierarchy problem (e.g., [19, 47]) such secondary storage is invariably considered to be a rotating magnetised medium of some sort, but in the last decade it has become increasingly likely that secondary storage will be provided by a Flash memory based solid state drive (SSD) which has lower latency and shorter (and essentially constant) mean read access times than spinning media [99].

Figure 2.6 on the following page shows a simplified schematic of a typical memory hierarchy in an embedded real-time system. As we move down the latency of the storage increases and it also becomes more difficult to calculate a WCET [12].

Although high in the hierarchy, caches present a particular problem for real-time system design, as what is present in a cache is dependent on the string of previous memory references. While the *principle of locality* [49] (discussed at greater length in 3.1.2 below) is likely to ensure this means caches increase the average performance in general computing tasks, the average

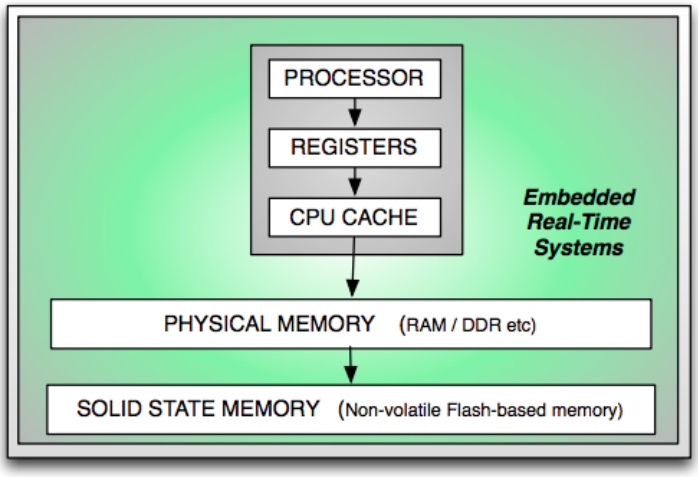

Figure 2.6: Memory hierarchy in typical real-time system [12]

case is not a safe basis of performance measurement in real-time systems design: the focus must be on the worst case. The alternative of assuming that all attempted cache accesses result in a miss is safe but likely to significantly over-estimate even the worst case execution time. Accounting for cache behaviour is extremely complex [163].

In the case of a NoC, CPU cache may not be present and instead on-chip memory may be managed as a scratchpad: here on-chip memory is mapped to a range of physical addresses inside the global physical memory mapping. Typically the programmer, with compiler support, ensures that needed resources are made available when needed in this scratchpad area. Scratchpads avoid the timing uncertainties of caches: if an object is in the physical memory range that is mapped to the scratchpad then it will be accessible in a fixed and known time, while a cache-based access may incur a hit or a miss with the inherent uncertainty that involves [151].

Scratchpads are more energy- and area-efficient than caches as they do not require an associative array [23]: another reason why they are favoured by designers of network-on-chip devices and multiprocessor system on a chip designers more generally [15]. An alternative approach is to 'lock' down lines in caches, effectively turning off replacement policies in the cache (policies which are often opaque and poorly documented and so troublesome for WCET determination in any case). This maintains the advantage of addressing being (relatively) transparent to the programmer while potentially also offering certainty. It does not, however, eliminate the issue of conflicts in cache line allocations (which are generated by the lower order bits of a cached address matching) and cache pollution, but nevertheless generate similar results to the use of scratchpads in terms of WCET [135].

2.7 REPLACEMENT POLICIES

In a memory hierarchy where the total range of addresses which need to be accessed exceeds the total space available in fast or local memory then a policy on replacement is required. Such replacement can be under programmer control, often in the form of an explicit "overlay" where blocks of code or data are removed and replaced. This technique was once widely used in general computing but has now been superseded by virtual memory-based techniques. It remains, though, an option for embedded programmers, especially where MMU support is not available [125, 79], but here we discuss the principles of algorithmic page replacement, with a fuller discussion of practical policies following in 3.1.4 on page 58.

2.7.1 The working set

Short of an implementation of the "clairvoyant" optimal OPT (or MIN) policy which selects which memory objects (typically pages) it would be most efficient to replace on the basis of foreknowledge of future memory reference patterns [19], holding those pages currently in use, *the working set*, is the best approach. A working set page replacement policy [47] holds a variable number of pages in memory, removing them when the time since they were last used exceeds a threshold.

This approach is claimed as the most efficient in general computing and best at avoiding the thrashing phenomenon. Although, as we discuss below, it is generally not seen as a *practical* approach itself, it serves to illustrate the qualities that an effective and efficient policy should have and most practical policies are designed to be close analogues.

The working set of a program in execution is the set of pages that have been accessed by the program in an arbitrary (but generally small) amount of time (the *working set window*, often referred to as θ). In the examples we illustrate here θ is proxied for by instruction counts. In Figure 2.7 [109], it can be seen that the working set size of a program, in this case a MySQL daemon servicing a simple client open and close on Linux, can vary significantly.

Programs in execution go through phases of *locality of reference*. Spacial locality is seen when future memory references are to made to the same location or to close-by locations. Temporal locality occurs when spatial locality is observed in a short (but generally arbitrary) timeframe.

In Chapter 1 of [150] four bases for the principle of locality are set out and are presented here in a shortened form:

- Program execution is generally sequential.

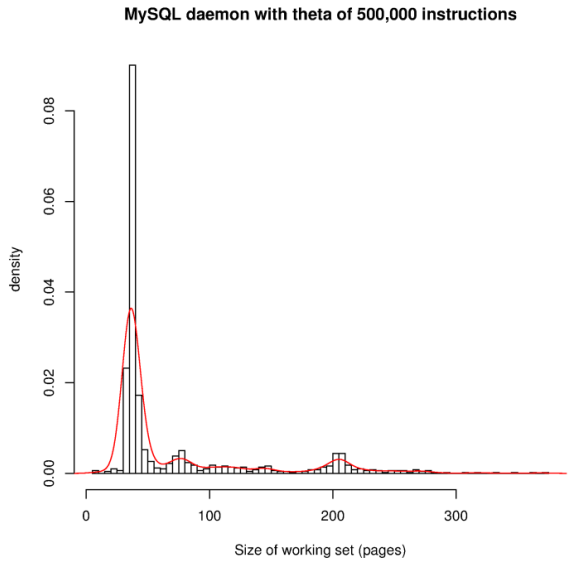

Figure 2.7: Distribution of working set sizes for MySQL daemon

- It is rare for large numbers of procedure calls (which disrupt locality of execution) to follow each other sequentially, so for a small enough time window execution in any block of code is likely to be spatially localised.
- Iterative constructs (such as loops) tend to be small, and so execution of these produces references which are concentrated spatially.
- Data structures are often in the form of sequential records (such as arrays) and so iterating through them requires sequential access to addresses in much the same was as instructions.

Studies of programs written in imperative high level languages, such as [145] which included C, probably the most widely used language in real-time systems development [37], support these arguments, though there is also the suggestion that the memory indirection used in object-orientated languages and frameworks (such as Java) my disrupt locality [82] and that a research effort has been undertaken to improve locality of reference in object-orientated code [89].

In each phase locality can be strong and successive memory references are likely to be closely clustered in space. In the transitions between one phase and another, however, successive references may not be to the same locality. In those transition periods the working set is likely to grow in size and so a working set policy relies on accessing a variable sized store of pages.

In reality working set policies are difficult to implement as they require hardware to monitor the time of each memory access and software or hardware to maintain a list of pages ordered

by access times. Paging policies in real world computing systems tend to be analogues of such policies, typically based on a 'CLOCK' policy [38] where pages are periodically checked to see if they have been recently accessed and those pages which have not are marked as candidates for replacement. In the Linux operating system this operates through a '2Q' policy [88], where pages that are frequently accessed are placed on a higher priority list and thus become less likely to be evicted.

For small memory systems such as those found in NoCs there is an additional impracticality with a working set policy: we simply do not have enough free memory to implement it. Setting a small θ so that in periods of phase transition there is room for the number or retained pages to grow means operating with an extremely high fault rate at other times as few pages are retained. This is why, in these experiments, we used either CLOCK or FIFO page replacement. In fact our general conclusion is that a better page replacement algorithm is less important than a better page loading algorithm.

2.7.2 FIFO and other policies

Algorithmic page replacement policies are based on the fact that memory references show this spatial and timing locality: this is not a provable quality of programs but rather a consequence of how they are written and produced [149]. Thus the algorithms used are typically founded on observed program behaviour and also the behaviour of the hardware in use in any given system [14]. Perhaps as a result there a plethora of different policies have been proposed to manage replacement decisions: ten are named in [168] though even this list is far from comprehensive, leaving out, for instance, First In First Out (FIFO).

As noted in the literature algorithmic policies may be divided orthogonally. Firstly as to whether they are adaptive or non-adaptive (in the sense of whether they monitor which objects are in use after their initial load and select candidates for replacement based on this information), and secondly as to whether they are local or global policies.

FIFO is a non-adaptive policy: here memory objects are simply stored in a form of load-time order and when space for a new memory object is required the oldest present object or objects are evicted. FIFO may seem crude but in general it is simple to operate (it requires no scanning of memory objects present nor sophisticated hardware support) and can still generate good results [154] as, although the list of memory objects present is not updated to reflect use patterns, locality is observed to go through phases [47] and so older objects may no longer be needed. The simplicity of FIFO, despite its obvious drawbacks, is a reason it is commonly used in hardware cache replacement [138] and even in sophisticated operating systems: in the OpenVMS operating system a modified FIFO policy is used on Alpha-based systems

where there is no explicit hardware support for marking page access [70]. A working set list, effectively a circular queue, is maintained and pages (see 3.1.3 on page 55) are added in their order of access. When the queue is full and a new page needs to be added at the end it replaces the page at the start of the queue. (FIFO page replacement policies are subject, though, to "Bélády's anomaly" where increasing the number of page frames can lead to a higher number of faults [20].)

The question of whether to operate a global or a local replacement policy occurs with multiprogramming. Here local memory may hold objects which come from a number of different programs and the question is whether a program in execution that needs more space should face seeing a trim of its existing use of memory (a local policy) or whether all memory objects should be available for replacement (a global policy). Any policy which aims to securely hold in place those memory objects currently in use by the running program must, by definition be "local" as it focuses on the program in execution. Local policies are likely to deliver better results but global policies dominate in real-world usage because of their relative simplicity [47, 87].

2.7.3 Load control

In his formulation of the working set, Denning [47] emphasises the need to apply *load control* to a multi-computing environment: warning that once the load (i.e. programs being executed) reaches the point that programs cannot maintain their working sets in memory then adding more programs results in a fall in processor utilisation as waiting for memory requests to be serviced dominates performance.

Denning's proposal is that when the number of jobs submitted exceeds a number (which he labels M) which is determined to be a maximum limit of active jobs (this would be set somewhat higher than the optimal multiprogramming point) then only M jobs could be active and allowed to hold space in main (fast) memory and use the CPU or I/O devices. All other jobs would be held in an inactive queue. This would prevent a catastrophic fall in efficiency as more jobs are added.

This model presents difficulties for us. The severe constraints on local memory seen in a NoC mean we are already adopting a one-process-per-core model, but even then (cf. 4.3.1) it is unlikely that many programs could maintain anything like a working set in memory. Additionally the real time criterion makes it difficult to suspend program execution for long periods. We explore alternative load control mechanisms in 4.7 and subsequently.

2.8 PARALLEL COMPUTING: THE IMPACT OF AMDAHL'S LAW

NoCs may appear to be ideally suited to tackle highly parallel tasks and in [90] Asinovich et al. discuss 13 fundamental classes of parallel computing problems which they believe will be important in future engineering and scientific computing and which, therefore, ought to be prime use domains for highly parallel many-core computing systems. Yet we must also account for the well-known limitation on parallel computing speed up known as "Amdahl's Law" [81]. This governs the limit on how much speed-up adding additional, parallel, resources will bring. In its simplest form the law can be stated as:

$$S(f,N) = \frac{1}{(1-f) + \frac{f}{N}} \tag{2.1}$$

Where S is the fractional speed-up, f the fraction of the program that runs in parallel (and $1-f$ that which runs in serial) and N the number of processors attacking the parallel part.

But if we consider heterogeneous systems we should restate this as:

$$S(f,N,p_S,p_P) = \frac{1}{\frac{1-f}{p_S} + \frac{f}{Np_P}} \tag{2.2}$$

Where p_S is the measure of core performance when running serial code and p_P core performance when running parallel code[2]. For a homogeneous system our baseline assumption should be $p_p = p_s$. Though we should note that cores may be heterogeneous, for instance as in the case of the Cell Processor [91], or that it may be that small cores used to run parallel tasks can be combined into a more powerful serial core or that we can use dynamic frequency management to boost the performance of a single core when it is running a serial task.

Formula (2.2) is certainly a simplification: p_S and p_P will vary from application to application as a result of memory access patterns, the effect of cache misses and so on but, at the price of further approximation we can use (2.2) to model how a homogeneous NoC might perform. From Esmaeilzadeh et al. [57] we have the cubic polynomial for the Pareto frontier[3] that marks the edge of the power/performance relationship for amd64 type cores at the 45nm production node:

$$W(p) = 0.0002p^3 + 0.0009p^2 + 0.3859p - 0.0301 \tag{2.3}$$

2 Esmaeilzadeh et al. [57] present an alternative, if related, formulation when considering future multicore performance: $S = 1/(\frac{f}{S_{parallel}} + \frac{1-f}{S_{serial}})$, where $S_{parallel}$ and S_{serial} are the speed-ups seen by future many-core designs in, respectively, parallel and serial computing, in comparison to today's "baseline" multicore (SMP) systems.

3 i.e., the observed limits for maximum efficient performance

Number of cores	Modelled SPECmark per core
1	35
2	22.76
4	13.41
8	7.26
16	3.77
32	1.94
64	1.01
128	0.55
256	0.31
512	0.20
1024	0.14

Table 2.1: Estimated effect of increasing core numbers on individual core performance for fixed power budget at 45nm node

Where W is the power consumed (in Watts) and p is core performance (in SPECmarks[4]). We can then use cubic splines to interpolate a value of p for a given W and, using the power budget for an imaginary chip that performs at 35 SPECmarks on the Pareto frontier, which is close to the observed maximum performance of any core, to model how a many-core NoC system with an arbitrary number of homogeneous 45nm cores would perform for various workloads that mix parallel and serial portions of code - as illustrated in Table 2.1 and in Figure 2.8. It can be seen that increased performance is dependent on having highly parallel tasks to run.

In the future, as we move further from the 45nm node, chip performance will improve (even if not at the historic rate), and many-core systems may become progressively better for handling less parallel tasks. But in their modelling of the move to many core chips Esmaeilzadeh et al. suggest that even on an optimistic view of future core scaling little would be gained by using much more than 32 cores on even highly parallel tasks. Figure 2.9 (from [57]) shows their projections for the impact of Amdahl's law in many-core systems, using both conservative projections of the impact of further transistor scaling and the then extant

4 The general point is that the higher the SPECmark the better the core performance, see http://www.spec.org/spec/glossary/#specmark and the Standard Performance Evaluation Corporation's website more generally for further information (link accessed 18 May 2013).

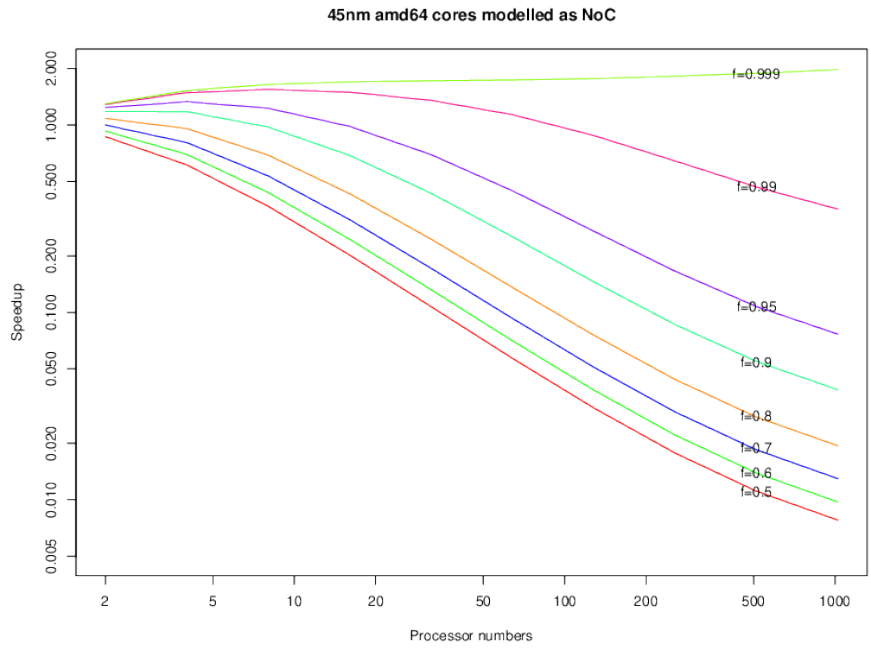

Figure 2.8: Modelled effect of Amdahl's Law for many core Network on Chip system (Here f denotes the proportion of the code that is fully parallelisable)

projections from the International Technology Roadmap for Semiconductors (ITRS). They conclude that the bulk of what they call "the dark silicon gap" - the difference between continued historically achieved increases in performance projected forward and those we can actually expect to see - is caused by a lack of realisable parallelism in tasks. However, in real word usage they conclude that it will be power constraints - the need to stop chips from overheating - that will contribute the most dark silicon, i.e., the proportion of a chip that cannot be used at a given time due to constraints on power use.

We give outline a practical example of the difficulties of securing efficient computing for highly parallel tasks with a many-core NoC in Appendix A. Our example appears to confirm practical limits on many-core systems tackling highly parallel tasks but we cite congestion in the memory connection as a key factor here.

This "dark silicon" problem - the inability to use all of a chip given limits on heat dissipation - is a consequence of the increased power density as transistor sizes continue to decrease. It may emerge as a dominant factor in designs of both future NoC hardware and system software. For hardware designers it may suggest using chip space to implement heterogeneous designs rather than seek to fill all the space with fast, high-energy-using, general processing capacity. For system software design, managing heat issues and the need to managing halting or

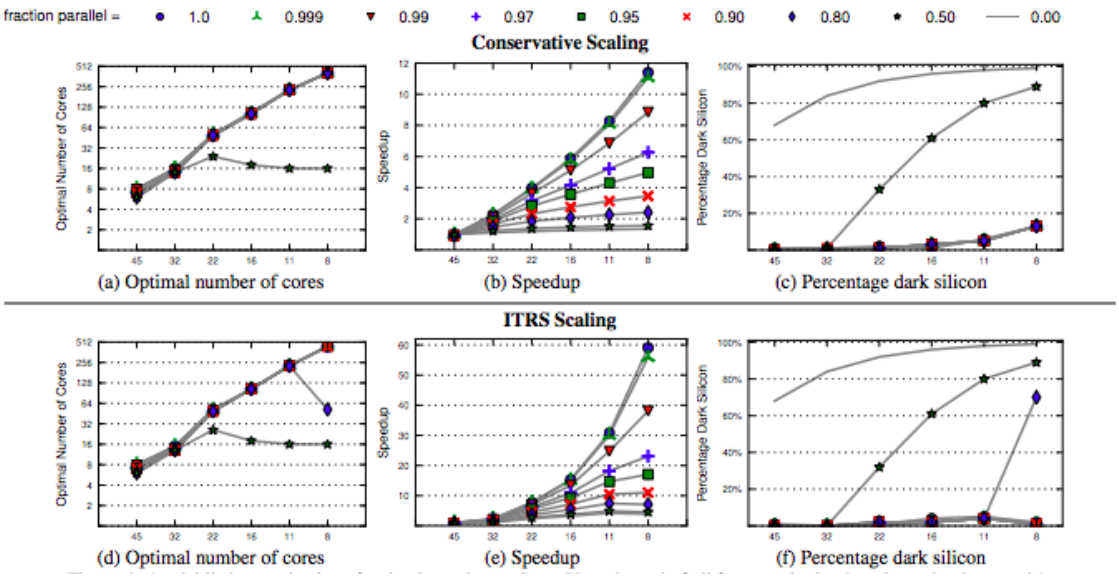

Figure 2.9: Esmaeilzadeh et al's projections of the impact of Amdahl's law in manycore systems (The horizontal axis indicates the technology mode)

slowing a particular core while seeking both good performance and deterministic processing will be challenging.

Figure 2.8 on the previous page is illustrative rather than definitive and is based on a number of simplifying assumptions (such as the parallel portions - measured by f - of the workloads being decomposable into a number of equal parallel sub-tasks that match processor counts) but does illustrate the point that if the point of using more cores is only to speed execution then the tasks will need to be highly parallel and all processors in use for the maximum amount of time. Appendix A suggests this will be difficult or even impossible to deliver with a congested connection to external memory for many-core devices. Instead, in Chapters 4 and 5 we concentrate on multiple real-time tasks running concurrently but independently.

2.9 SUMMARY

In this chapter we have outline the factors that have seen multi- and many-core computing grow and describe some of the new problems that growth has created for real-time systems designers and programmers in particular.

In 2.1 and 2.2 we discuss how scientific advance and commercial pressures continue to make faster computing at least theoretically possible, but note that the end of "Dennard Scaling"

(the ability to maintain a constant power density even as chip frequency increased) means that the long-followed option of ever-faster single core CPUs is no longer feasible. Instead many-core designs are increasingly common and system designers and programmers must look for ways to run code in parallel if performance increases consumers expect are to be maintained.

In 2.3 we discuss why buses, while allowing for cache coherency between a small number of cores, scale poorly for many-core designs and that, instead, packetised communications through an on-chip network are increasingly preferred as core numbers grow.

A variety of such network-on-chip designs and topologies exist but we note that simple mesh designs are most likely to be implemented (2.4.1) though these generally have poorer performance characteristics than alternative designs, as they lack symmetry and bandwidth. As a result resources can be significantly different distances from individual cores and traffic on the on-core network can face bottlenecks, while the plethora of routes from cores to external connections can complicate timing considerations. We illustrate some of these issues in a discussion of commercial and experimental NoC systems (2.4.2).

Shared memory trees, which ensure that global memory is equidistant from all cores, offer at least a partial answer to this problem and we introduce them in 2.4.6 (though they are not discussed in depth until 4.2.4).

In 2.5 we briefly discuss the use of graphics processing units (GPUs) as a many-core alternative to NoCs and note they face similar memory-bottleneck problems.

In 2.6 we discuss the problems of a memory hierarchy and note that using caches may increase timing uncertainty in real-time contexts, or else force the use of excessively pessimistic timing estimates, while in 2.7 we introduce the subject of memory replacement and the impact different policies may have and how they typically aim to exploit locality of reference in computer programs.

Finally, in 2.8, we use the findings presented in [57] to show that, in many-core designs, very high levels of parallelism will be needed to see code speed-up: memory bottlenecks or anything else which makes code more serial and less parallel could make many-core designs less practical if the sole criterion for their use is speed-up of execution.

3 VIRTUAL MEMORY ON A MANY-CORE SYSTEM

Virtual memory (VM) is ubiquitous in general computing but has generally been avoided in embedded and real-time environments. In this chapter, in 3.1 we consider how virtual memory can address the evolving challenges facing real-time developers, discussing segmentation but concentrating on paged virtual memory. In 3.2 we consider thrashing - a problem seen with VM implementations when memory replacement demands begin to overwhelm the system. In 3.3 we discuss real paged program behaviour and how a novel approach to paged virtual memory - *partial paging* - can reduce the number of memory requests in the system and so mitigate thrashing. We summarise this chapter in 3.4.

3.1 THE CASE FOR VIRTUAL MEMORY IN REAL-TIME SYSTEMS

3.1.1 Real-time software is becoming bigger and more complex

The case for using virtual memory in real-time and embedded systems is under-pinned by the growing complexity of the demands made on software in such systems. Embedded systems are becoming bigger and are expected to do more and, in general, demands and expectations are increasing more rapidly than resources.

A decade or so ago the most popular embedded computing devices were low-end mobile phones. Something approaching 10^9 of these were being produced annually in 2009, each typically using somewhat less than 10^6 object codes [54]. More recent figures suggest that while the sales of mobile phone devices are of the same order as 2009 [158], the software they are running is likely to be both modularised and several orders of magnitude bigger [106]. At the higher end, code size and complexity are also growing: a Boeing Dreamliner, introduced commercially in 2011, runs on 14 million lines of code, while the F-35 Lightning-II jet fighter, deployed operationally from 2015 onwards, uses 24 million lines of code [166].

While embedded software deployed in real-time contexts is getting bigger and more complex, the need to combine and concentrate computing centres in embedded systems for reasons of practicality and cost control is growing. For instance, between 2002 and 2015 the number of electronic control units (ECU) in the Volvo XC90 car increased from 38 to 108 [104].

Combining ECU functionality to control hardware costs becomes increasingly important [63] while that same process may increase software complexity and security requirements [107].

And, as the realm of real-time system deployment spreads (such as into driverless cars), and more personal and sensitive data is handled by real-time systems, concerns about the security of such systems are also growing.

3.1.2 Virtual memory

Our principal contribution in this thesis is to propose a new approach to virtual memory that allows embedded systems with small amounts of local, fast, memory to access the advantages that VM offers whilst mitigating the performance and deterministic drawbacks that have traditionally led to VM being avoided by embedded and real time systems designers [134].

Virtual memory offers [50]:

- a means to logically separate the physical and logical/virtual location of resources so allowing more efficient use (generally in a way which is transparent to the programmer) of limited, fast, memory.
- a way to make programs more generally useful, as they do not necessarily need to be dependent on the layout of a given machine's physical memory map.
- a mechanism to ensure resources are shared in a secure manner as well as means to isolate errant or malicious programs.

In VM a program in execution presents a *virtual* (or *logical*) memory address of a resource (whether code or data) to a processing unit. This must be translated into a *physical* address that represents the absolute location of the sought resource. A logical address might simply represent a mapping between physical addresses and some other range of addresses (or differently ordered addresses) but is much more likely to cover a much larger range than physical memory (or physical memory and memory-mapped devices) alone would support: generally a virtual address space is likely to cover the whole of the space mappable by a processor and this allows for programs which are much bigger than the available amount of physical memory to be run. Using logical addresses allows each program being executed to appear to have access to the full range of addresses supported in hardware.

The process of translation allows:

- For a hierarchy of resources - for instance at a given instant the virtual address of an item being sought could translate to a location in storage medium other than volatile memory or even to a resource currently stored on another system entirely.

- For security - the necessity of address translation allows accesses to resources be monitored and thus for resource protection (for instance limiting access to certain addresses to systems software only).
- Resource sharing and multiprogramming - alongside security, translation allows for sharing of virtual to physical mappings, and by partitioning the physical address space can enable multi-computing.
- For modularity - by being freed from the need to use absolute resource locations programs can be built from blocks which can, in effect, collaborate inside a shared virtual address space.

Creating a hierarchy of resources can allow a system with small local memory to transparently appear as of arbitrary size, though typically for a 32-bit processor we have a 4GB address space (2^{32} bytes) and for a 64-bit processor an address space of 2^n where n might typically be 42, 48 or some other number up to 64. This hierarchical system can still deliver efficient computing because of the principle of locality [49]: address references tend to cluster in time and space, so if the most accessible (i.e. fastest) part of a memory hierarchy contains resources which are close in distance to other recently accessed resources the system is likely to be efficient. For instance, traversing loops or accessing compact data structures are behaviours typical of real-world computer programs.

3.1.3 VM mechanisms: segmentation and paging

Mechanisms are required to implement VM and policies are needed to ensure these mechanisms are used efficiently [50]. In this subsection we consider the two fundamental mechanisms: segmentation and paging.

Segmentation allows programmers to allocate variable sized blocks for code and data - segments - and resources inside the segment can be accessed by referencing a segment number (or address) and an offset within the segment. Each segment is linear and contiguous in physical memory. In simplified terms the typical arrangement is that a segment has a *base address* which can be looked up in a per process table stored in physical memory, perhaps as a segment itself (typically a processor register points to the base of this table and this register is loaded with this address when the execution of a particular process begins). The table will also store other information about the segment, such as its extent (an attempt to access beyond the segment's length can then be trapped as an exception) and the access privileges the current process enjoys (so, for instance, attempting to write to a read-only segment can be

trapped as an exception). Segments can be shared amongst multiple processes using these mechanisms.

Segmentation is not transparent to the programmer - segments are allocated under program control - and this is one of the reasons segmentation is no longer widely used in modern operating systems. Hardware constraints have also been a factor and Intel's x86-64 architecture only has very limited support for segmentation [84]. Segmentation is also prone to external fragmentation because it can use variable and dynamically sized segments: this fragmentation is highly undesirable where fast memory is in extremely short supply. While we note that memory compaction approaches that are compatible with the demands of real-time systems have been proposed [42], we do not examine segmentation any further here.

The alternative mechanism, on which we will concentrate in this thesis, is *paging*. In paging the main or fast memory is divided into equal-sized blocks known as *page frames* into which resources from remote locations are loaded, generally in portions of the same size as the page frames. These portions are known as *pages*. We stipulate they are 'generally' loaded at the same size because in this thesis we explicitly adopt another approach of only loading a part of a page.

As with segments a table - a *page table* - will hold information about the mapping between the page frames and the pages as well as status information about page frames (such as whether the mappings are valid). Historically page fames have tended to be of 4KB size: seemingly chosen as a good trade off between magnetic disk throughput speeds and waiting times for subsequent requests. More recently the trend in many general and server computing devices has been towards bigger pages [161, 67].

Taking 4KB pages as an example, secondary memory can be regarded as being divided into pages numbered by the lowest address in a page shifted right by 12 bits - hence the address `0x80074D` would be in page `0x800` and so on. Similarly, a resource at this address would be at offset `0x74D` if loaded into a 4KB page frame.

Figure 3.1 on the facing page shows a simplified version of address translation - a virtual address is broken into an offset (lower numbers) and a page number. The page number functions as an index to a page table or a system of page tables, which returns a physical page address which is recombined to with the offset to produce a physical address.

For large addresses page tables are usually found in several layers - as a full 64-bit address space with 4KB pages would require over 10^{15} page table entries with a flat page table system. For example, on a typical 32 bit Linux system on Intel hardware there are two levels of page tables of 4KB pages, while on Linux on x86-64 hardware there are four (and even then only the lower 48 bits of a virtual address are used) [28]. On such an x86-64 system the highest level in the page table system is indexed by bits 47 - 39 and each of the 1024 table entries

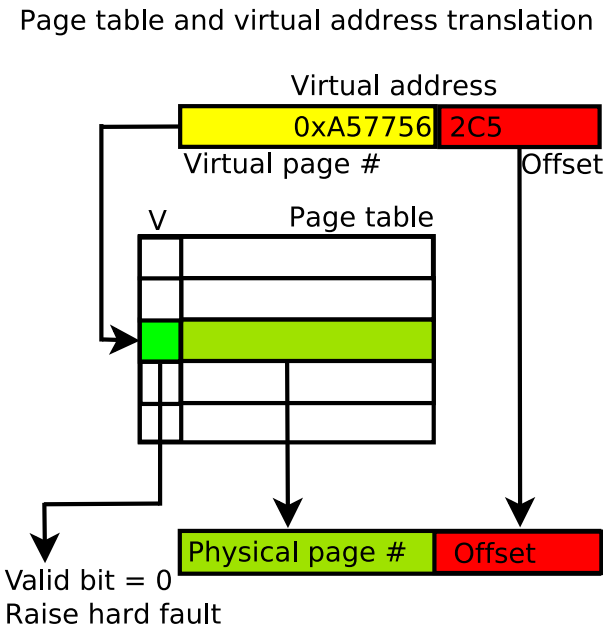

Figure 3.1: Paging - a virtual address is used to index entries in a page table

references a 128 GB region: those 128 GB regions not in use need only be marked as 'not present' at this level of the table, and do not require child tables lower down the hierarchy.

TRANSLATION LOOKASIDE BUFFERS Page tables are dynamic and so any table in current use has to be present in volatile memory. As each program in execution has its own virtual address space and mappings, generally each must have its own page tables. To speed access to the tables system a special hardware register usually points to the table at the very top of the hierarchy. To avoid the need for a page table look up on on every instruction load or access to data, page tables are typically supported in hardware by *translation lookaside buffers* (TLBs). These TLBs cache page lookups and, by avoiding the need for a search through (multiple) page tables on each instruction, can greatly speed execution. In [126] "typical" values for TLBs are given as:

- TLB size: 16 - 512 entries
- Hit time: 0.5 - 1 clock cycle
- Miss penalty: 10 - 100 clock cycles
- Miss rate: 0.01% - 1%

As TLBs "run hot" - being accessed on every use of a VM address and likely to be using high speed associative array circuits - they can also use a significant amount of power. In [146]

the authors quote Intel as saying TLBs consume 13% of core power for "memory intensive workloads" while in [117] a figure of 15 - 17% for cores designed for embedded systems is quoted.

3.1.4 VM policies: page replacement

The number of page frames available is limited and in most practical systems will be smaller than the number of pages needed to execute the current workload to completion. Thus system software needs to operate a *page replacement policy*, which may take various forms, to allow new pages to replace those already present. A good policy is one that consistently removes the page with the *longest reuse distance* (which may be infinite). Replacement may be a costly process repeatedly picking the 'wrong' page for replacement (i.e. a page with a shorter reuse distance) will slow the system down and may lead to the phenomenon of *thrashing* (see 3.2) where more and more time is lost to page replacement.

3.2 THRASHING

In the proceeding 3.1 we reviewed the case for using virtual memory and some of the policies a practical paging-based VM system must implement. Now we consider one of the fundamental problems with VM: so-called *thrashing*.

Thrashing "turns a shortage of memory space into a surplus of processor time" [48]: this surplus, though, generally manifests itself as idleness as the processor waits for a memory request to be serviced elsewhere in the system. Thrashing is the situation when the delays caused by waiting for such requests to complete come to dominate system performance. If pages in the working set cannot be kept in memory then fault rates will be high and a shortage of memory can be converted into an excess of CPU idle time as progress stalls.

We can model a simple system as an M/G/1 queue (cf. Chapter 4 in [35]) - i.e., as having a *M*arkovian distribution of arrival times into the system, a *G*eneral distribution of service times with *1* server [41, 40].

It will be noted that an M/G/1 system is not a perfect match to the system we simulate from 4 onwards. At a basic level the queue in our system is not operating in continuous time but in discrete units of time (i.e., CPU cycles). Further, as we discuss in 4.9.1, there is, in effect, a queue-within-the-queue with a stochastic process governing which request gets to the head of the queue (i.e., there is not a strict ordering of requests). Although the system we simulate

processes four requests at once, for a queuing packet this is little different from a service capable of faster processing yet only admits one request at a time.

Thus we believe it offers insight into behaviour and also shows why it is worthwhile trying to reduce queue lengths even when queue lengths are well above the level generally accepted as indicating thrashing.

3.2.1 Modelling a queue for memory service

We here outline the basic findings for the length of a Markovian (M/G/1) queue for memory service drawing on [40] and [35].

As in [35] we model arrivals with a Poisson process with parameter λ, with $X(t)$ measuring how long a packet or customer arriving at time t will have to wait before service begins and $g(x)$ being the distribution of service times. Then, taking $p_0(t)$ to be the probability that at time t the system is empty (and therefore that $X(t) = 0$), and $p(z,t)$ to be the density for $X(t) > 0$, we have for the *distribution function* of $X(t)$:

$$F(x,t) = p_o(t) + \int_0^x p(z,t)dz \tag{3.1}$$

We measure the *virtual waiting time* of an arriving packet/customer and so assume that waiting time is consumed in unit (virtual) time, hence if $X(t) = \Delta t$ at time t and there are no arrivals, then $X(t+\Delta t) = 0$. As arrivals are subject to a Poisson distribution then in time Δt we expect $\lambda\Delta t + o(\Delta t)$ arrivals (and conversely we expect the probability of there being no arrivals to be $1 - \lambda\Delta t + o(\Delta t)$: as is the convention, $o(\Delta t)$ can be assumed to much more rapidly converge to zero than Δt itself.

Hence:

$$p(x,t+\Delta t) = p(x+\Delta t,t)(1-\lambda\Delta t) + p_0(t)g(x)\lambda\Delta t + \lambda\Delta t\int_0^x p(x-y,t)g(y)dy + o(\Delta t) \tag{3.2}$$

$$p_0(t+\Delta t) = p_0(t)(1-\lambda\Delta t) + p(0,t)\Delta t(1-\lambda\Delta t) + o(\Delta t) \tag{3.3}$$

This gives us:

$$\frac{\partial p(x,t)}{\partial t} = \frac{\partial p(x,t)}{\partial x} - \lambda p(x,t) + \lambda p_0(t)g(x) + \lambda\int_0^x p(x-y,t)g(y)dy \tag{3.4}$$

$$p_0'(t) = -\lambda p_o(t) + p(0,t) \tag{3.5}$$

If the system is in equilibrium then $p(x,t) = p(x)$ and $p_0(t) = p_0$ and any derivative with respect to t is zero, hence 3.4 becomes:

$$0 = p'(x) - \lambda p(x) + \lambda p_0 g(x) + \lambda\int_0^x p(x-y)g(y)dy \tag{3.6}$$

and 3.5:

$$0 = -\lambda p_0 + p(0) \tag{3.7}$$

Taking the Laplace transform of 3.6 with:

$$p^*(s) = \int_0^\infty e^{-sx} p(x) dx \tag{3.8}$$

$$g^*(s) = \int_0^\infty e^{-sx} g(x) dx \tag{3.9}$$

We get[1]:

$$0 = sp^*(s) - p(0) - \lambda p^*(s) + \lambda p_0 g^*(s) + \lambda p^*(s) g^*(s) \tag{3.10}$$

From which (and 3.7):

$$p^*(s) = \frac{p(0)(1 - g^*(s))}{s - \lambda + \lambda g^*(s)} \tag{3.11}$$

From 3.7 we have $p(0) = \lambda p_0$ and from 3.1, p_0 here being a Dirac-delta:

$$p_0 + \int_0^\infty p(x) dx = p_0 + p^*(0) = 1 \tag{3.12}$$

And so we can restate 3.11 as:

$$p^*(s) = \frac{\lambda(1 - p^*(0))(1 - g^*(s))}{s - \lambda + \lambda g^*(s)} \tag{3.13}$$

Let the traffic intensity (or utilisation[2]) $\rho = \lambda \mu_b$ where μ_b is the mean service time: recalling that p_0 is the probability that the system is empty at any given time it can be seen that $\rho = 1 - p_0$ and hence:

$$p^*(s) = \frac{\lambda(1 - \rho)(1 - g^*(s))}{s - \lambda + \lambda g^*(s)} \tag{3.14}$$

From which:

$$p^*(s) = \frac{\frac{\rho}{s\mu_b}(1 - \rho)(1 - g^*(s))}{1 - \frac{\rho}{s\mu_b}(1 - g^*(s))} \tag{3.15}$$

From which we can see:

$$w^*(s) = p_0 + p^*(s) = \frac{1 - \rho}{1 - \rho h^*(s)} \tag{3.16}$$

1 Using Laplace transform pair $\frac{df(t)}{dt} \leftrightarrow sf^*(s) - f(0^-)$

2 NB: We are considering a system that can only process one request at a time.

Where:

$$h^*(s) = \frac{1 - g^*(s)}{s\mu_b} \tag{3.17}$$

As a (double-sided) Laplace transform evaluated at $-s$ is equivalent to a moment generating function evaluated at s we can see that:

$$-w^{*\prime}(0) = \mathbb{E}\{waiting - time\} = \frac{\rho\mu_b(1 + c^2)}{2(1 - \rho)} \tag{3.18}$$

Where c is the co-efficient of variation of the service time (i.e., the ratio of the standard deviation to the mean) and $\mathbb{E}\{waiting - time\}$ the expectation for waiting-time. This is the Pollaczek-Khintchine formula.

Little's law [98] states:

$$N = \lambda W \tag{3.19}$$

Where N is the (mean) total number of requests in the system, λ the arrival rate as before, and W the mean time taken to service a request.

We can see $W = -w^{*\prime}(0) + \mu_b$, so:

$$N = \lambda\frac{\rho\mu_b(1 + c^2)}{2(1 - \rho)} + \lambda\mu_b \tag{3.20}$$

And

$$N = \frac{\rho^2(1 + c^2)}{2(1 - \rho)} + \rho \tag{3.21}$$

Two classic evaluations found in the literature are for $c = 0$ and $c = 1$ and both have their applications in computing [35].

If we take $c \approx 0$, which is a reasonable assumption for a well-behaved memory management unit dealing with DRAM:

$$N = \frac{\rho(2 - \rho)}{2(1 - \rho)} \tag{3.22}$$

While for any device where the service times are exponentially distributed (a baseline assumption for a magnetic disk [118]) we can (in virtual units) say $c^2 \approx 1$ and so get :

$$N = \frac{\rho}{1 - \rho} \tag{3.23}$$

In our case (cf. 4.9) we have for the benchmarks chosen $c \approx 0.35$ ($c^2 \approx 0.122$) and $\mu_b \approx 1.27$.

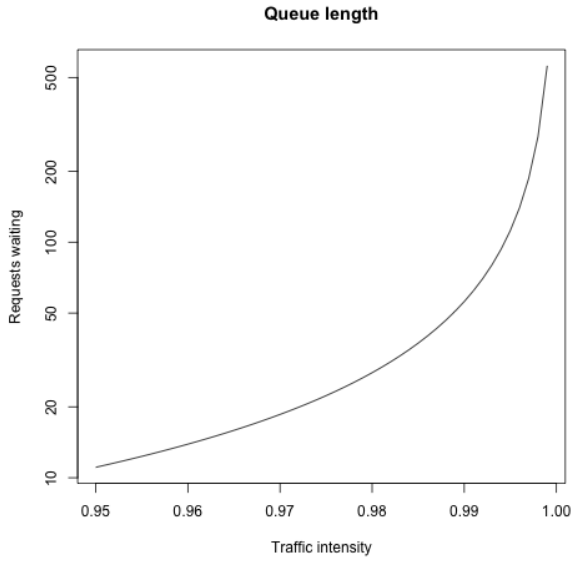

Figure 3.2: Waiting requests for modelled system with M/G/1 queue

Denning's view is that thrashing - which brings "severe performance degradation" [48]- begins when $N = 1$ [47] but Figure 3.2, which shows a semi-log plot of waiting requests for our modelled system from $\rho = 0.95$ to $\rho = 0.999$ and suggests that when $N > 1$ there is still log-linearity in performance. The elbow in this curve is an artefact of the scaling of the plot but it does suggest catastrophic decline does not begin until traffic intensity becomes very close to 1.

A linear regression gives us the approximation that for q_u the queue length, $q_u \approx 8 \times 10^{-16} \exp(36.559\rho)$ when ρ is between 0.95 and 0.99: indicating a substantial return in the form of shorter queues and hence smaller waiting times for even a small decrease in ρ.

3.3 REDUCING MEMORY REQUESTS

To examine ways we might reduce ρ we considered the behaviour of a thrashing system.

From the PARSEC parallel computing benchmarks [24], we traced the memory access string, using the Lackey program from Valgrind [119], of the x264 video encoding program[3] when configured for a maximum of 16 concurrent threads and running on Intel x86-64 commodity hardware. Each memory reference was encoded as a line of XML[4], as an instruction, a load from a memory address, a store to a memory address or a modify (a combined load and

3 See http://wiki.cs.princeton.edu/index.php/PARSEC#X264 - accessed 20 December 2017
4 See https://github.com/mcmenaminadrian/lackey_xml - accessed 21 December 2017

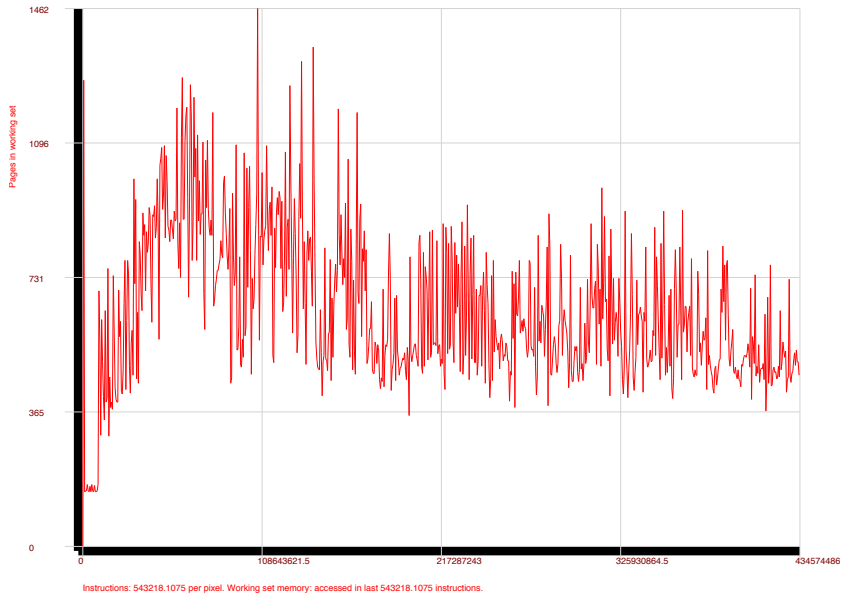

Figure 3.3: Working set size for PARSEC x264 benchmark, in 4KB pages

store). For instance, an `lea` (load effective address) instruction would generate two lines of XML, the first for the instruction itself and the second a load for the address being accessed.

The benchmark program's memory use is large, certainly in the context of typical real time systems: static analysis shows it accesses a total of 14164 4KB pages, including both instructions and data. Figure 3.3 shows the working set for the benchmark[5] and it can be seen it is consistently well above 300 pages for almost the whole benchmark's life. The peak, at 1462 pages, is > 5.7 MB.

We simulated running this on a 16 core system where the cores shared 512KB of local memory. In modelling how various page replacement policies worked we assumed a global clock for the whole system and that any item resident in local memory (i.e., within the 512KB of shared memory) could be reached in one tick of this clock by any core. In contrast, if a page is not present then it will take 50 ticks per 16 byte 'line' to load the page, so a 4KB page will take $256 \times 50 = 12800$ ticks to load. However if a page is faulted in by another thread while the waiting is going on then it will end as soon as that has happened. It can be seen this model will significantly under-report the time taken to load memory as it ignores all the problems of inter-process and processor communication and all issues of coherency. Nor does

5 The "working set window" here is chosen by the graphing software as 543218.1075 [sic] bytes of instructions as the best fit for the chart size.

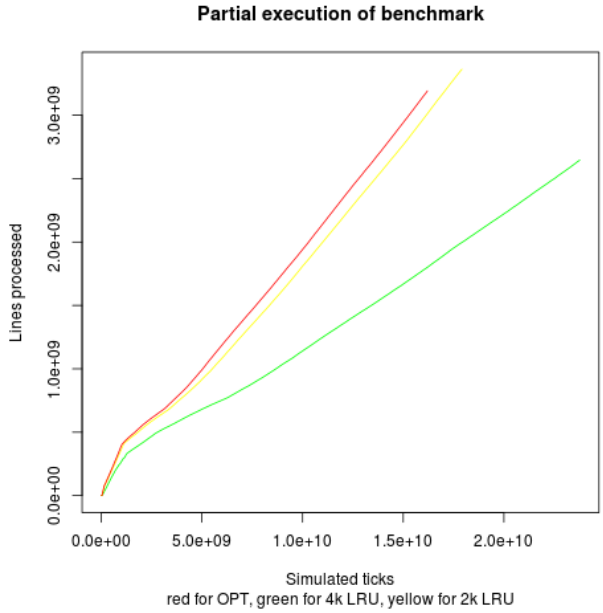

Figure 3.4: OPT and LRU for 2KB pages compared

it allow for any congestion in the memory interlink. Further it assumes that pages that are replaced can be written back to main memory without any timing cost.

Using the trace it is possible to model an OPT page replacement policy and this is the red line in Figure 3.4. The effect of thrashing is shown in Figure 3.5: although multiple processors join execution after around 3 billion ticks, the slope of the plot of lines (of XML) processed remains constant: additional processors cannot speed up execution because performance is dominated by the cost of transferring pages from the global store to local memory.

Figure 3.4 does show that smaller page sizes appear to work better in this constrained memory environment with the performance of a least recently used (LRU) page replacement algorithm with 2KB pages coming close to the performance of OPT with 4KB pages.

3.3.1 Episodic and lifetime accesses to pages

Analysis (Figures 3.6 and 3.7) shows that, on average, only a small fraction of any page is used during each time it is in memory, but that individual pages may be loaded and unloaded hundreds or even thousands of times: there are 14,164 4KB pages referenced by the benchmark but a total of 9,749,394 page loads referenced in our data for the 4KB LRU and 7,463,883 page loads referenced in our data for the 4KB OPT. Neither of these benchmarks were run to completion but the data suggests that the mean number of times each individual page would be loaded and unloaded will be $\sim 10^3$ times if the benchmark completed.

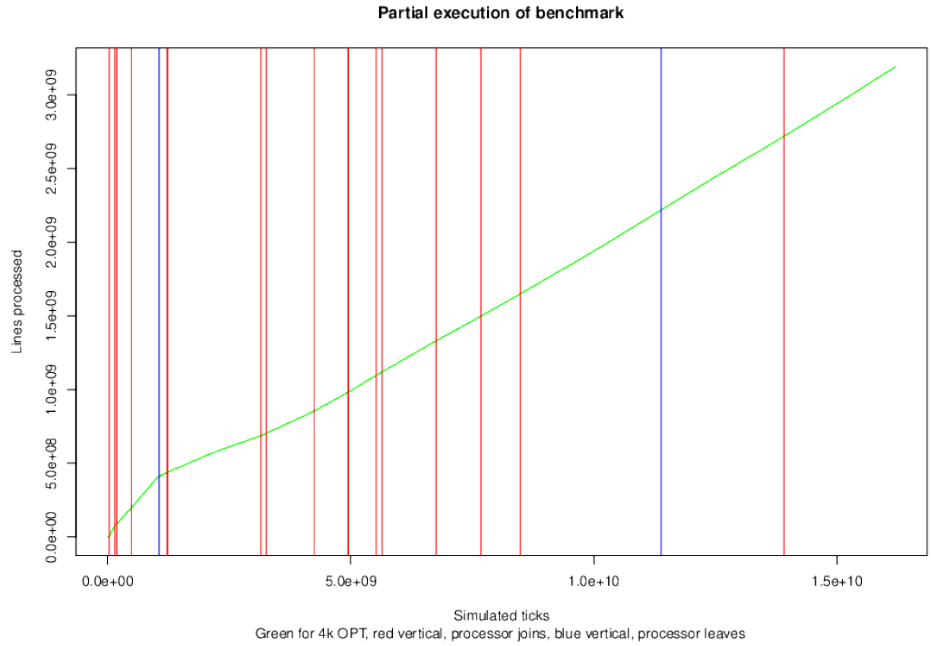

Figure 3.5: Processor joins and leaves for OPT simulation

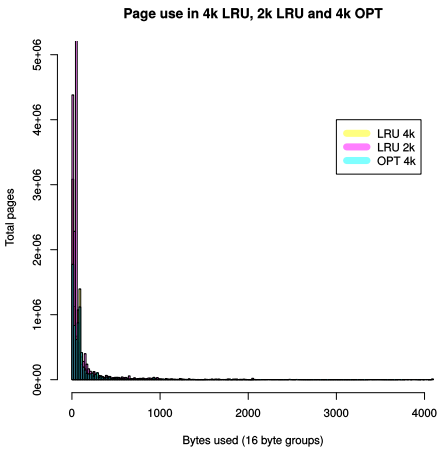

Figure 3.6: Bytes used in each page in simulation

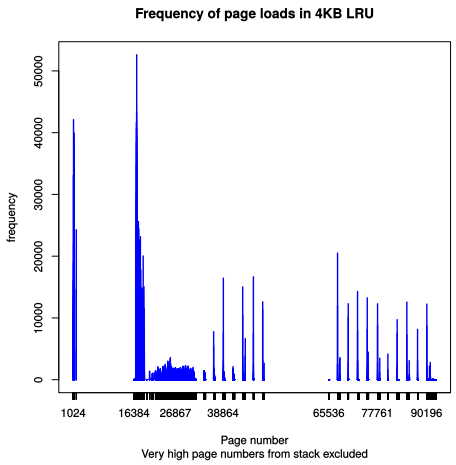

Figure 3.7: Frequency of page loads

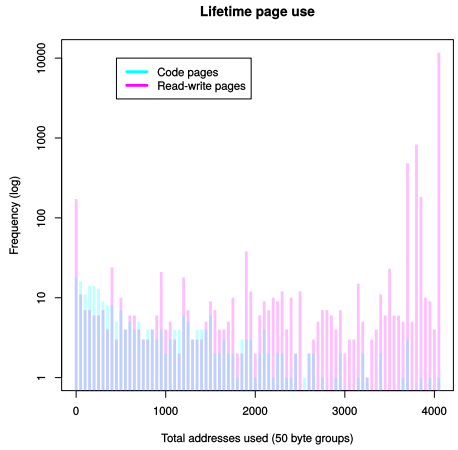

Figure 3.8: Page use over the whole life of the benchmark

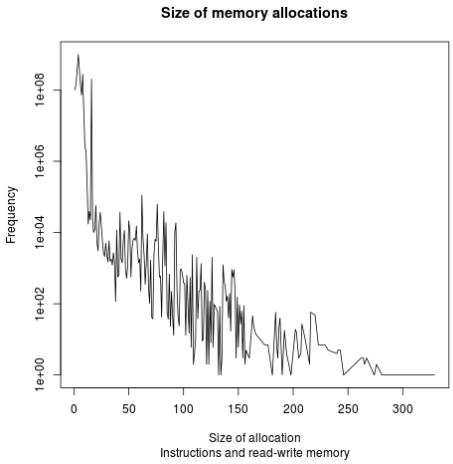

Figure 3.9: Frequency of different sized time-contiguous memory accesses

3.3.2 More efficient loading: partial paging

As Figure 3.8 shows, for many pages the benchmark eventually accesses the whole of a referenced page or substantial proportion of a referenced page: the low use on each load is because of a shortage of space. But with a standard paging mechanism a whole page is loaded (and in our single threaded execution model all progress on the affected core is stalled while this takes place), even though it is likely that only a very small portion will be used before the page is then ejected. A more efficient page loading algorithm, preserving the advantages of VM and page-based addressing, but not forcing the system to load a whole page on every hard fault is sought.

Figure 3.9 shows the time-contiguous memory accesses (i.e., when a thread of execution accesses a block of code of a particular length before a jump or when it allocates, modifies or reads memory locations of a particular length before moving to a new locality) are typically four or more orders of magnitude greater for allocations of 16 bytes or fewer than for any other allocation size.

Therefore we propose to replace full page loading with *partial paging*.

In what we will call 'traditional' paging a reference to an address in a page not mapped to local memory as indicated by the page table - i.e. a hard fault - typically creates to following pattern of operating system and program behaviour:

1. Program execution is interrupted and program state is saved (typically this will involve pushing register contents on to the stack) and the processor switches to kernel mode operation.

2. Assuming no free page frames exist, a page must be selected for replacement, and marked as invalid in the page table (and, if required, the page being replaced must be written back to memory) - otherwise a free page frame can be used.

3. The page that contains the missing address is loaded into the selected (freed) page frame.

4. The page table is updated to reflect the new mapping.

5. Program state is restored, execution returns to user mode and the instruction that referenced the missing address is re-executed.

In partial paging when a reference to an unmapped address is made then the essential steps to service a hard fault now become (this is an outline - we give a formal definition of a partial paging algorithm in 4.2.2):

1. As before, program execution is interrupted and program state is saved.

2. As before, and assuming no free page frames exist, a page must be selected for replacement and marked as invalid in the page table - otherwise a free page frame can be used. If a page being replaced needs to be written back to main memory, only those parts of the page which which we previously loaded need be written back. A bitmap, recording which parts of the page were in use, is reset to a null state (all zeros).

3. Only that part (16 bytes) of the page that contains the missing address is loaded into memory and the bitmap is updated (a bit switched to 1) to reflect that this is the only valid part of the page.

4. The page table is updated to reflect the new mapping.

5. Program state is restored and the instruction that referenced the missing address is re-executed.

In other words: a reference to a page that is not present in local memory generates a hard fault as before (pages tables are updated and pages replaced as required) but instead of a whole page being loaded only a 16-byte proportion is transferred. On subsequent accesses to this page further 16-byte sections may need to be loaded: these *'small faults'* do not require the page tables to be updated but a bitmap is maintained to track what portions of a page are present. This necessitates that the bitmap also be checked on all memory references to see if the part of the page being sought is present.

Small faults will require an interrupt to be raised and program state to be saved as with a hard fault, but by saving on the time lost in loading parts of pages that are not referenced

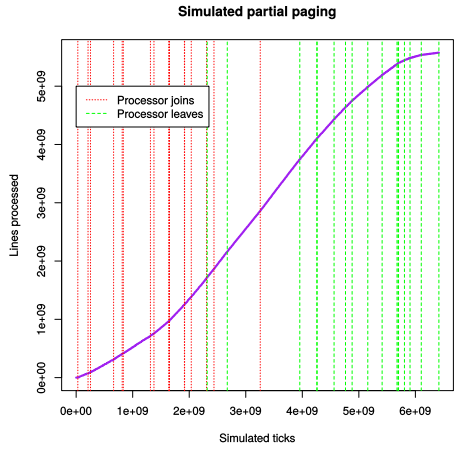

Figure 3.10: Performance of partial paging simulated

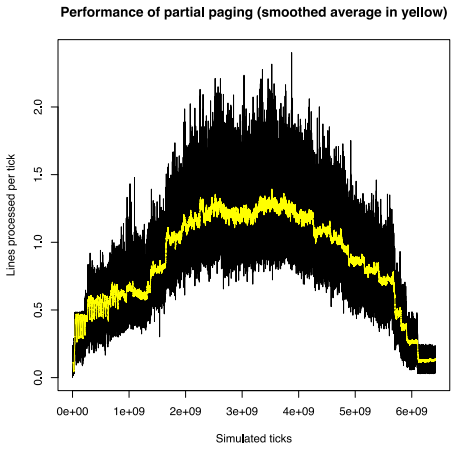

Figure 3.11: XML lines processed per tick with partial paging model

before they are replaced, we hope to show that partial paging is more efficient in such memory-constrained environments.

As a proof-of-concept of such a partial paging approach we constrained the system described above (3.3) to 30KB per core (for a total of 480KB of shared memory, with the 'missing' memory being used to notionally implement the logic needed for this approach) and used 2KB pages, as with the earlier 2KB LRU model and again tested it with the trace from the PARSEC x264 benchmark. We increased the time required to access a block of memory that was present to four ticks to allow for any additional time to check the bitmap. This was not, though, a rigorous test, merely an attempt to illustrate the potential for partial paging.

A page replacement algorithm must still be used and a 2Q/LRU model, as used for the 2KB and 4KB LRU approaches described above, was applied.

The results of this simulation are shown in Figure 3.10 (cf. Figure 3.4) and Figure 3.11. Figure 3.10 (cf. Figure 3.4) shows that the partial paging system performs much better than any of the traditional paging systems and combined study of both shows that whilst adding new processors sees the efficiency (each line of XML represents a memory reference) of any individual processor fall, the overall system efficiency does rise and fall as processors join and leave the execution.

3.4 SUMMARY

Embedded real-time systems are being faced with more computationally complex and demanding tasks and many-core designs such as network-on-chip systems offer an opportunity

to handle them efficiently with their access to multiple processing elements (3.1.1). The use of virtual memory, a mature technology in general computing (3.1.2), will allow larger programs to run as it is adapted to the memory hierarchy we see in NoCs and similar many-core devices. VM-based systems will also be able to share resources more securely and isolate poorly-written or malicious code.

However, NoCs and similar designs face a large "memory gap" as each processor only has access to a very small amount of fast memory. We can see that this, if we use a traditional paging-VM design, is likely to generate severe thrashing, with much time lost to loading whole pages even though only small portions are likely to be used before the page must be replaced (3.2). Thus we propose an alternative design: partial paging. This retains many of the elements of a standard paging-VM design but instead of loading a whole page loads only small portions of a page on demand (3.3.2).

When competition for limited fast memory is strong, partial paging is likely to diminish rather than remove thrashing, but such diminution, if reflected in even small decreases in demands for memory could still deliver worthwhile performance improvements (3.2.1).

4 MODELLING A PARTIAL PAGING SYSTEM

We have demonstrated a motivational case for using a system which, while retaining demand paging mechanisms only loads a part of a page on demand (i.e., what we call *'partial paging'*) and now wish to model a system that more closely reflects the demands and constraints of the real world. All this must be done in software as, as far as we are aware, no hardware that implements a partial paging system currently exists.

In this chapter we outline the workings of, and the results from, the many-core NoC we simulate in Sections 4.1 and 4.2. In doing so we describe the partial paging algorithm in some depth in the latter (4.2.2). In 4.2.3 we discuss how partial paging might be implemented in hardware.

In Section 4.3 we describe the benchmarks we use to test and measure the system and in Section 4.4 we give the broad results of our simulation, while in Section 4.5 we show how a 'traditional' whole-page-loading paging VM system performs. Our results show that performance is dominated by congestion in the link between the cores and the memory management unit. The partial paging system performs better here but for both partial paging and the 'traditional' system the length of the queue for memory service (i.e. the amount of *'blocking'*) is such that processors are typically idle more than half of the time.

In Section 4.6 we discuss the issues we must consider when comparing these results and whether we can use partial paging's superior performance in the simulations as a clear sign that it would outperform traditional paging when using real hardware.

In 4.7 we discuss a simple *load control* mechanism (cf. 2.7.3) - which we call *delayed partial paging* - implemented by assuming that the system is delayed on determining whether an address being accessed is currently available in local memory (we do this by assuming there is a delay in the reading of the bitmaps used in partial paging cf. 4.2.3). This delay, in effect, devotes relatively more computing resources to requests already in the queue for memory service than to actions that might generate further requests. We demonstrate that even this simple load control mechanism could deliver better WCETs for a set of tasks, with the worst case timing for what are likely to be the most demanding programs lowered at the expense of increasing typical completion times for many or most other programs.

In Section 4.8 we outline the observed worst-case timings with partial paging before considering in Section 4.9 how we can model the factors driving the worst case and predict

worst case outcomes at safety-critical limits. In doing so we will demonstrate that partial paging delivers better average performance because it reduces queue lengths (blocking) in the memory system and also, for the same reason, lessens uncertainty about completion times, allowing less-pessimistic WCETs to be used. We develop a model of entropy in the system which offers an explanation for the greater certainty (smaller range) for completion times we see with partial paging and we also use statistical-based extreme value theory methods to consider completion times at safety critical levels.

Section 4.10 is a brief summary of this chapter.

4.1 A MANY-CORE NOC SYSTEM

Many-core NoCs are likely to become more common and we simulate such a system here.

The predictions of the early 2010s that the time of widely-deployed 1024-core NoC-based systems was imminent have proved false: for instance, pioneers Adapteva have had to abandon their plans for a commercial 1024-core system and they have effectively ceased development[1]. Core numbers in commercial products have have grown though, albeit at a slower rate than once thought likely, and are expected to continue to grow in future years [17].

4.1.1 NoCs versus GPUs

We concentrate here on NoCs rather than GPU designs. This is despite the fact that, over the course of the last decade, GPUs have come to dominate both commercial and research efforts to tackle highly parallel computing tasks [86]. GPUs are suited for *regular* parallel tasks such as matrix multiplication rather than a wider range of general computing tasks and can even under-perform with *irregular* parallel tasks (i.e., those where it is hard to find independent sub-tasks) [120]. In Appendix A we simulate using a many-core NoC, with partial paging, to compute a regular parallel task.

Although we concentrate on NoCs here, the issues we consider are relevant to GPUs:

- GPUs do not escape from the memory bottleneck problem: GPU designs restrict threads to very small amounts of local memory, whether in the register file, cache or as scratchpad and avoiding fetches from DRAM is important for both energy and speed reasons [68].

1 See http://www.adapteva.com/andreas-blog/adapteva-status/ - checked 13 January 2018

- Support for virtual memory on GPU devices as part of heterogeneous cores is already available on commercial hardware, though GPUs show much greater latency in handling page faults than CPUs [157].

4.2 ELEMENTS OF THE SIMULATION

Hardware to simulate a many-core NoC operating partial paging was not available and nor were we aware of any commercial or public simulation system that would allow us to simulate such a system (e.g., the Microblaze simulation we used in [110] could be easily modified to simulate partial paging for a single core but not for multiple cores [123]).

Hence we needed to build our own simulation in software (using C++ with the GNU toolset [60] along with the Qt SDK to provide user interface elements [36]). The simulation is focused on handling memory references and does not compute the results of the code. Instead it uses traces of memory references generated by Spike, the reference instruction set simulator for the open RISC-V core technology originally developed at UC Berkeley [160]. Thus our concentration is on ensuring that the simulated memory operations - rather than computations themselves - are correctly handled (though in Appendix A we also have to ensure the computation is correct).

This correctness of the code was verified by stepping through it line by line using the GDB debugger and in also ensuring that the stepping counts and fault counts matched expected values.

4.2.1 A 128-core homogeneous NoC

Our basic model is of a 128 core homogeneous NoC, with each tile having a single core and 16KB of local memory. As stated above each core runs single process.

4.2.2 The partial paging algorithm

Algorithm 4.1 on the next page (with 4.2) provides an outline of the memory reading algorithm used with partial paging when using 1KB pages.

Here a VPN suffix indicates the 'virtual page number', PPN suffix indicates 'physical page number', i.e., the global memory physical page frame and PFN indicates 'page frame number' - i.e., the local page frame. The '*' in front of a memory address indicates memory dereferencing.

Algorithm 4.1 Simplified representation of memory read using partial paging on NoC

```
 1: procedure ReadAddress(Address)
Input: Address
Output: *MappedAddress
 2:     _VPN ← ShiftRight(Address, 0x0A)
 3:     if _VPN ∈ TLB_VPN then                                  ▷ Parallel lookup
 4:         return CheckBitmap(Address, TLB_PPN, TLB_PFN)
 5:     else
 6:         return HardFault(Address)
 7:     end if
 8: end procedure
 9: procedure CheckBitmap(Address, TLB_PPN, TLB_PFN)
10:     LineNumber ← ShiftRight(Address&0x3FF, 4)
11:     BitMask ← ShiftLeft(1, LineNumber)
12:     Bitmap ← GetBitmapAddress(TLB_PFN)
13:     if BitMask & *Bitmap then
14:         PageFrameOffset ← ShiftLeft(TLB_PFN, 0x0A)
15:         return  *(PageFrameOffset + BaseAddressForLocalPages + Address & 0x3FF)
16:     else
17:         return SmallFault(Address, TLB_PPN, TLB_PFN)
18:     end if
19: end procedure
20: procedure HardFault(Address)
21:     _PFN ← EvictPage()
22:     Bitmap ← GetBitmapAddress(_PFN)
23:     *Bitmap ← 0                                  ▷ zero Bitmap for evicted page
24:     _PPN ← GlobalPageLookup(Address)
25:     MapNewPage(Address, _PPN, _PFN)
26:     return ReadAddress(Address)
27: end procedure
28: procedure SmallFault(MappedAddress, _PPN, _PFN)
29:     FetchRemoteMemory(_PPN, _PFN, MappedAddress & 0x3F0)
30:     UpdateBitmap(_PFN, MappedAddress & 0x3F0)
31:     return *MappedAddress
32: end procedure
```

Algorithm 4.2 Partial paging - supporting functions

```
1: procedure EvictPage
2:     PageTableEntry ← PageReplacementOutcome()                ▷ e.g. CLOCK
3:     Dirty ← ReadStatusFromPageTable(PageTableEntry[_PFN])
4:     if Dirty then
5:         WriteBackPage(PageTableEntry)
6:     end if
7:     InvalidateTLBEntry(PageTableEntry)
8:     InvalidatePageTableEntry(PageTableEntry)
9:     return PageTableEntry[_PFN]
10: end procedure
11: procedure WriteBackPage(PageTableEntry)
12:     _PPN ← GlobalPageLookup(PageTableEntry[_VPN])    ▷ Enable page state update
13:     for i ← 0, 0x3F do
14:         Bitmap ← GetBitmapAddress(PageTableEntry[_PFN])
15:         if *Bitmap & ShiftLeft(1, i) then
16:             *(_PPN + i * 0x10) ← *(PageTableEntry[_PFN] + i * 0x10)
17:         end if
18:     end for
19: end procedure
20: procedure MapNewPage(Address, _PPN, _PFN)
21:     _VPN ← ShiftRight(Address, 0x0A)
22:     *((PageTableBaseAddress + _PFN * SizeOfPTRec)[_VPN]) ← _VPN
23:     *((PageTableBaseAddress + _PFN * SizeOfPTRec)[_PPN]) ← _PPN
24: end procedure
25: procedure FetchRemoteMemory(_PPN, _PFN, LineNumber)
26:     *(_PFN + LineNumber * 0x10) ← *(_PPN + LineNumber * 0x10)
27: end procedure
28: procedure UpdateBitmap(_PFN, LineNumber)
29:     Bitmap ← GetBitmapAddress(_PFN)
30:     *Bitmap ← *(Bitmap|ShiftLeft(1, LineNumber))
31: end procedure
32: procedure GetBitmapAddress(_PFN)
33:     return BitmapBaseAddress + SizeOfBitmap * _PFN
34: end procedure
```

A call to ReadAddress leads (as in traditional paging systems) to a TLB lookup. If the lookup fails then a HardFault is raised (in a system using CLOCK as page replacement a walk of the page tables is likely to required first, but we have ignored all page replacement mechanisms here). If the TLB lookup succeeds then a call to CheckBitmap is made (in fact we would expect hardware to conduct the TLB check and the bitmap check in parallel - see 4.2.3).

If the TLB check passes but the bitmap check fails then a SmallFault is raised and only a proportion of the page (16 bytes) is loaded, and the bitmap is updated accordingly

A missing page is handled by HardFault in a different way from a traditional paging system: whilst the page tables and the TLB entries are updated, again, in effect, only 16 bytes are loaded as with the SmallFault[2]. Subsequent accesses to the page may generate further small faults. As information about which parts of a page are present are held in a per-page bitmap, on a hard fault only those parts of a page that are marked as present in the bitmap need be considered for write-back if required (as shown in EvictPage here). In our simplified system pages marked as have been written to are marked as dirty and are written back and pages not marked dirty are simply discarded on replacement. A page is initially loaded as read-only (clean) if the first reference to it is an instruction or a load. A page first accessed via a store is loaded as read-write (and marked as dirty). A page loaded as read-only will be subsequently be promoted to read-write if a store is called on an address inside the page.

4.2.3 Checking bitmaps

In 4.2.2 we state that the TLB check and the bitmap check should be done in parallel, and it would also be our hope that such checking would be sub-cycle to ensure the maximum speed of execution (though, as we discuss in 4.7 adding delay at this point could be used to enforce load control). However, while this thesis is on the potential software impact of using partial paging, we can outline at least one way that could be explored to deliver bitmap checking in hardware.

Taking the 1KB pages example discussed in this chapter we know that our system has 16 pages and that each bitmap will then be 64 bits long (as we map in 16 byte lines). Of a full 48 bit address, bits 4 - 9 inclusive then give the ordinal number of the bit we need to check. Thus we need a 6:64 decoder circuit to transform the binary coded line number into a high output that can then be tested (via a logical AND) against a bit in the bitmap. In 4.1 a schematic for part of a simple 6:64 decoder, using inverters and AND gates, is shown: for instance, if the

2 When counting hard faults and small faults we do not count the small fault generated here, but only the hard fault.

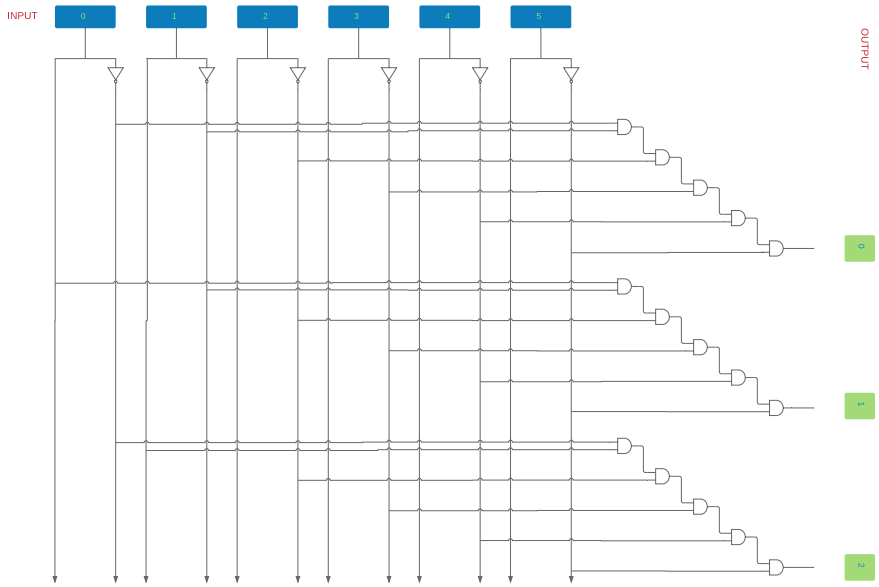

Figure 4.1: 6:64 decoder circuit (part)

coded line is 0 (i.e. all six input lines are held are held low) and then output line 0 (and only output line 0) will be high[3].

Figure 4.2 gives an outline of how a parallel check could work. Below we highlight the key points in the process that correspond to the numbered panels is Figure 4.2 :

1. Virtual Address in: As discussed above bits 4 - 9 give the 16 byte line (here marked L) for the address.

2. 64 bit output: a decoder (with input from the L bits) sets one of the 64 output lines out to high (all the others are held low).

3. Bitmap comparison: there 64 input lines and for every bitmap line 0 will be compared to bit 0 (using a logical AND), line 1 with bit 1 and so on. These results are output on 64 lines. Thus any bitmap where the bit coded in L is present will give a high output on that line (and that line only).

4. Combine output: for each bitmap the 64 lines out are combined (via logical OR) so that for each bitmap we have only a single high or low output.

5. TLB Output: As in a 'traditional' system the comparison of the virtual address with the data stored in the TLB generates an output, including a page frame number (there are

3 Typically decoders also have an additional input known as 'enable' to drive the circuit. Enable can be thought of as a third input to all the AND gates (as three input AND gates). For clarity we have not added this line here.

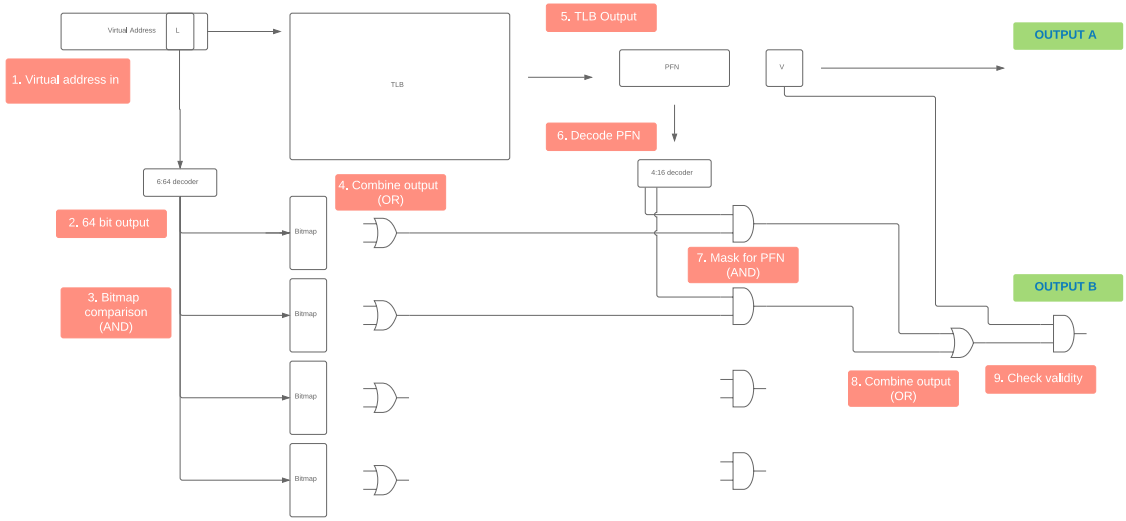

Figure 4.2: Outline of parallel TLB and bitmap check

16 for the 1KB page system, coded in 4 bits) and V - a valid bit which indicates if the TLB output is valid and thus usable.

6. Decode PFN: the PFN given by the TLB is passed through a 4:16 decoder, with one of the 16 output lines corresponding to the returned PFN set to high (and all others held low).

7. Mask for PFN: Each line from the PFN decoder is combined (with logical AND) with the matching single output line from the bitmap. Thus we only have high output at this stage if both the line coded in L and the PFN match.

8. Combine output: The output of the all the bitmap checks is combined (using logical OR), giving either a single 0 (low) or 1 (high) output.

9. Check validity: the single output from the previous stage is then tested (via logical AND) against V.

Two outputs are marked A and B. If A (i.e. V) is 1 (high) then the page has been loaded and mapped and is in the TLB. If V is low then (subject to a page walk in a CLOCK or similar page replacement environment where pages are removed from the TLB but retained in the page tables) a hard fault is required to load and map the page. If A and B are both 1/high then the line sought in the page is present and execution can continue. If A is high but B is low then the page is present and mapped but the line sought is not and a small fault is raised.

It can be seen that to avoid further memory fetches on checking the bitmaps these will need to be in something like the register file. As we have notionally allocated a page frame to the

bitmaps it may make more sense to think of this in terms of lost chip real estate needed to accommodate the fast circuitry needed to execute these checks.

4.2.4 Connection to external memory

Notionally the tiles are arranged in a "Manhattan" type mesh network but we have avoided considering questions of core-to-core (or more accurately, router-to-router) communications here. A system that relied on router-to-router transport to connect to external resources such as memory is potentially poorly suited to any safety-critical or hard real time operation as connection routes are non-symmetric, with each core seeing external resources in a different way and potentially having a multitude of routes (with different timing characteristics) through which to access the resources [12]. Cores and tasks can be mapped to ensure they are in the most advantageous position (though this is an NP-hard problem [132]) but optimising for the typical case in this way, as well as being complex, may not produce the best results for real time.

For real-time systems, there is a premium on timing predictability: such predictability allows efficient use of the hardware and avoids overly-pessimistic estimates of the worst case execution time (WCET) generated by adding time to observed or calculated WCETs to account for unknowns. Thus predictability favours connecting cores to external memory (which is the external resource we consider here) via a mechanism that ensures all cores see memory in the same way.

In smaller multi-core systems, such connections are typically provided by a shared bus. While only one core can master the bus at once, all have equal access and a bus also allows other cores to see memory transactions and so maintain coherent caches. But buses scale poorly (see 2.3 on page 33) and the complexities of analysing cache coherence in a many-core system may lead towards a pessimistic WCET being assumed: if programs depend on variable input data then static analysis will be unable to fully determine the contents of any caches and as "unknown parts of the state lead to non-deterministic behavior" [164]. Pessimistic bounds must be assumed for safety's sake, so removing any advantage that we might assume would come from caching.

Alternative approaches could include a crossbar where all cores can connect directly to the external memory or a hierarchy of buses where a limited number of cores are connected to a bus at a 'leaf' and these leaf buses are themselves connected together, perhaps to just one bus which also interfaces with the memory or to a deeper hierarchy of buses.

For both a crossbar approach and a bus-based approach it can be seen there must be some sort of arbitration: only one core can be connected at once through the crossbar and only one

processor (or lower tier bus) can master a bus at once. A typical, and widely-used, algorithm to manage such collisions in packetised networks is exponential backoff [112], which involves progressively doubling waiting delays from an initial semi-randomly selected delay. It can be seen that a bus hierarchy would have to be combined with a system of buffers, as a core might successfully master the leat bus but that packet might have to wait to make further progress up the 'bus tree'. Similarly, some buffering system at the interface between either the crossbar system and the memory management unit (MMU) or the root bus and the MMU would also be sensible.

A third alternative - the one we principally study here - is a tree-like hierarchy of buffers and binary multiplexors. Here each core is connected to a buffer able to hold a packet and requests for external memory negotiate a series of 2:1 multiplexors down a tree where a memory management unit (MMU) lies at the root. In this way each core is an equal distance from the MMU and packets wait in the buffers for the way ahead to clear.

Intuitively we should expect all three of these methods to generate similar results. In each the primary drivers of performance are (a) the speed and capacity of the MMU and (b) the general level of demand for memory (i.e., competition for access to the MMU). The tree does not have a backoff protocol and will have a large buffering capacity, but these need not be significant advantages if the backoff algorithm applied for the buses and crossbar schemes is reasonable and if there is enough buffering to ensure that the MMU does not lie idle while memory is in demand. On the other hand the depth of the tree will slow down requests and responses in comparison to the other schemes when demand for memory is low.

We demonstrate in Section 5.5 that the performance of all three schemes is broadly similar.

4.2.5 A Bluetree-like memory connect

We used a modified version of the Bluetree memory tree described above in 2.4.6 and by Garside in [64]. Figure 4.3 illustrates the general arrangement: although the NoC can be thought of as an 8 x 16 Manhattan grid, the binary memory tree can be considered as connecting to them in a serial order, so that column tiles 0 and 1 on the zeroth row of tiles are connected to MUX 0 on the zeroth row of MUXes, tile 2 and 3 to the MUX 1 on the zeroth row, and tiles 0 and 1 on the first row are connected to MUX 4 on the zeroth row and so on. In this way tile 0 on the zeroth row is to the left of every other tile, while tile 1 on the zeroth row is to the left of every tile except tile 0 on the zeroth row. And any tile on the 5th row is to the right of all the 56 tiles on rows 0, 1, 2 and 3 and so on.

Garside describes a system with a static priority where "[b]y convention, the left-hand input has priority over the right-hand side of the multiplexer", we experimentally discovered

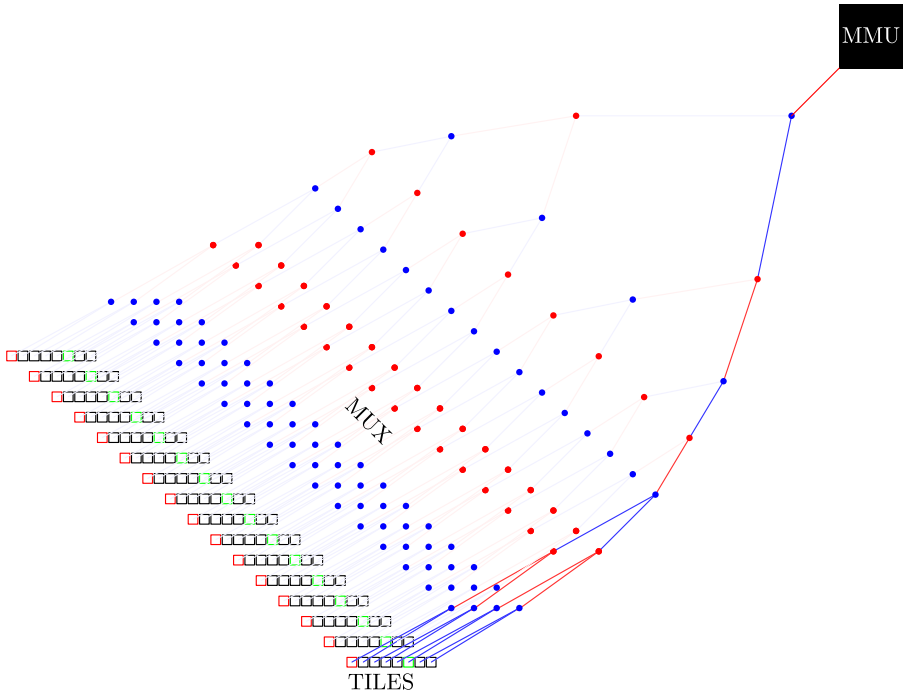

Figure 4.3: NoC tiles and memory tree (one row of tiles' paths to root and MMU highlighted)

that this approach produced grossly unfair results when the tree was connected to a very large number of cores and when all those cores were active. It is the effective serial ordering described above that makes this priority scheme extremely unfair.

In our experiments we ran the same eight benchmarks on each row of the NoC, so that tile 0 on row 0 was running the same benchmark as tile 0 on every other row and so on: making the performance of the same tile on each row directly comparable.

As Figure 4.4 shows the 'rightmost' processor takes several orders of magnitude more ticks to complete the first run of the benchmark if we always give priority to the leftmost packet at the mux[4] if two packets arrive simultaneously. To give a fairer performance we must instead model the muxes in our memory tree as though they have an internal flip flop or similar mechanism to store 1 bit of information. This would be set to 0 at the start, indicating that (for instance) the left hand input has initial priority and then flips on each packet, so that priority alternates between left and right in the event of a clash. Such an approach is still initially biased towards one direction, in that all muxes on the system begin with the same directional priority, but in a tree with a lot of traffic this bias will be unlikely to be significant over the life of the benchmark.

4 It should be noted that these tick counts are not directly comparable with the ticks in the fully developed model described in the rest of this chapter - being based on an earlier prototype - and should be considered as illustrative only.

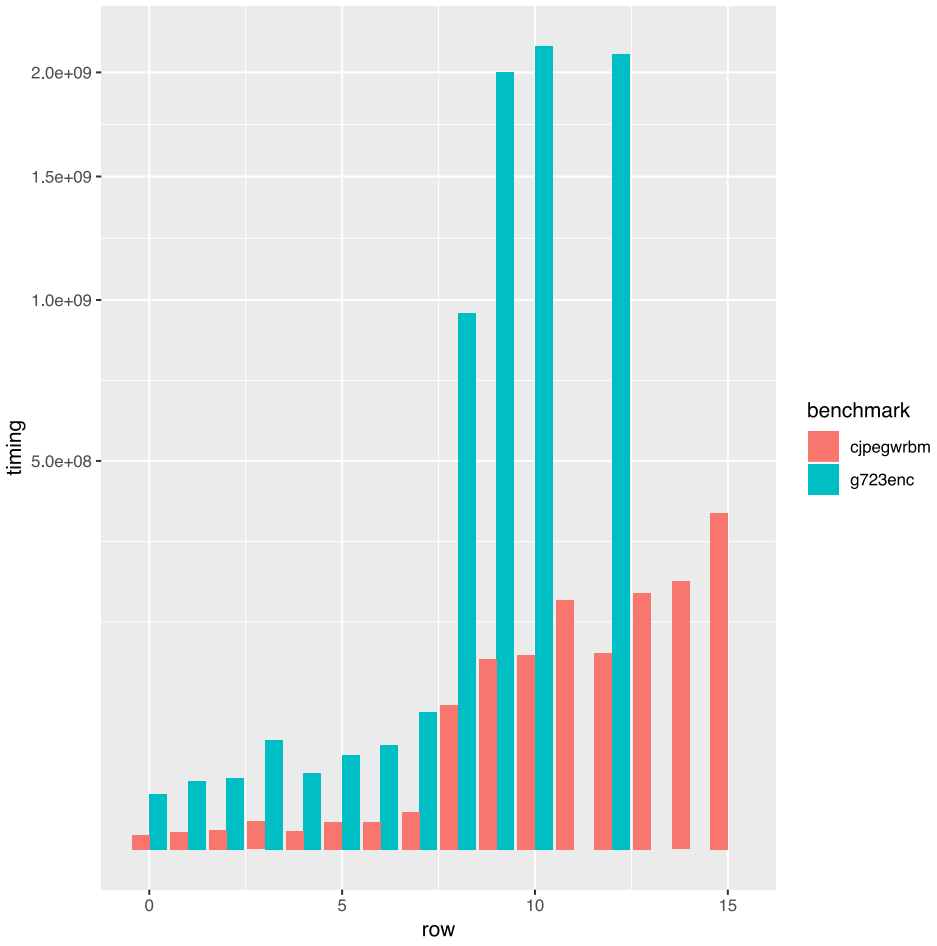

Figure 4.4: Simulated ticks taken to complete first run of benchmarks when using standard Bluetree priorities

The exception to this rule is at the connection to the MMU. We model a flash device capable of handling four requests in parallel and at the MMU two requests can be accepted in parallel (if they are coming from different buffers in the MUX and two slots are available in the MMU).

In [12] Audsley notes that our approach (or any approach that imposes an arbitrary priority independently of the scheduling needs of the system) could lead to priority inversion and that each mux can instead be programmed at runtime to acknowledge priority settings encapsulated in each memory request, however we have not explored this approach here.

4.2.6 Tile design and system memory structure

Our basic model is of a system with 128 tiles, each a single core and with 16KB of local memory. We do not model any of this memory as shareable (in contrast, for instance, to Intel's SCC) or look to use such memory to build a shared operating system [96]. We assume each tile's memory is fully partitioned from all other tiles: our experiments concentrate on reducing congestion in the memory interconnect and the demand on the MMU with a view to delivering deterministic performance rather than on questions of memory sharing and coherence. They reflect real world deployments where a single compute centre with multiple cores will deal with inputs from many different devices (as discussed with the Volvo example in 3.1.1 above).

We assume that global off-chip memory is addressable in a 48-bit address space and that off-chip global page tables need to be read each time a mapping between local and global memory is required, i.e., on a hard fault or when a page needs to be written back to global memory. The 48-bit address space is indexed through a set of four level page tables.

Figure 4.5 illustrates how the four level page tables work when we use 1KB pages: bits 63 to 48 (inclusive) of a 64 bit virtual address are discarded and then the upper 11 remaining bits (47 to 37) point to a reference in a single 'super directory' table which itself points to one (of many, theoretically) 'directory' tables. The next 9 bits (36 to 28) of the virtual address then provide an index within this directory table, which points to a 'super table', and then bits 10 -18 point on to a 'table', until the bottom 10 bits point us to an offset in a 1KB page that is referenced by the offset in the table.

In a real device this system of global page tables would likely be maintained on a per process basis and might itself be in-whole or in-part paged out of memory, so requiring parts of the system to be restored from secondary storage and adding to the delays on a hard fault. Here, though, we ignore these complications as our concentration is on the relation between local and global memory and not on managing potential interaction with slower secondary storage.

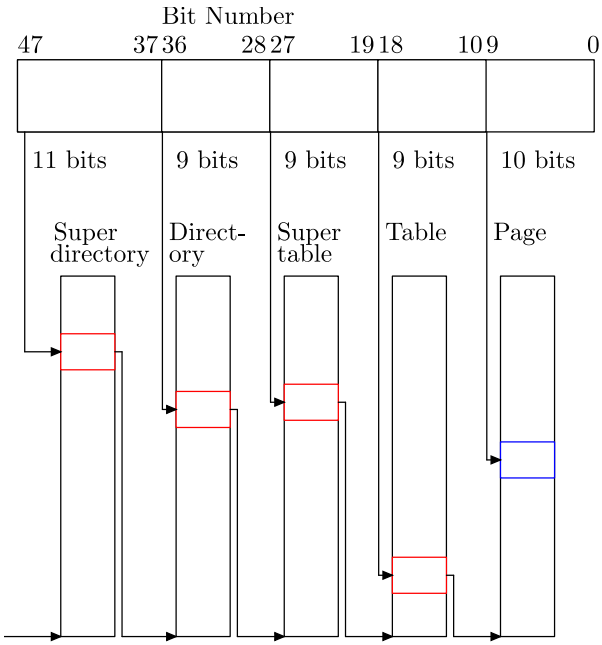

Figure 4.5: Four level page tables for 48 bit address space with 1KB pages

Thus we model the time it takes for a request to a global page table to traverse the memory tree and be processed by the MMU, but we do not look at questions of how the global page tables should be managed.

Executing HardFault in Algorithm 4.1 requires four reads to the global page table. In the context of partial paging these are the equivalent of four small faults in time cost: compared to just the one memory-tree traversal that the hard fault itself costs. These accesses to the global page table can therefore be delayed because of congestion on the memory tree interconnection, and so the speed at which they are processed is directly related to the overall efficiency of the partial paging system (more faults requires more reads and potentially longer delay): the thing we are hoping to measure.

4.2.7 Global memory as NAND flash

For a main memory store here we model a non-volatile random access device (as might be provided by NAND flash or more advanced technologies) which takes 50 cycles for a read of 16 bytes and 100 cycles for a write of 16 bytes and which can handle a maximum of four requests in parallel. These values are essentially arbitrary but are chosen to loosely reflect the capabilities of emerging solid-state memory technologies [128]. Higher latency here - which would be more reflective of contemporary as opposed to future technologies - would likely improve the relative performance of the partial paging system as it makes less demands on the memory controller compared to a traditional full-paging approach.

We assume that the flash system can process four simultaneous requests. We believe this to be a reasonable figure, but accept that our modelling of NAND Flash write times as fully fixed may not reflect uncertainty in real devices (though this determinism is reasonable for reads) [100].

4.2.8 CPU behaviour

The processors require a minimum of one cycle per instruction or local memory read or write. If the address sought is not in local memory then a fault is raised.

4.3 THE BENCHMARKS

To test the system we used eight benchmarks from the TACLebench suite [58], which is specifically designed to support research into worst-case execution times in real-time systems.

Benchmark	Name	Description	Code size (SLOC)
0	g723enc	CCITT G.723 encoder - voice encoder	480
1	admpcm_dec	ADPCM decoder	293
2	admpcm_enc	ADPCM encoder	316
3	audiobeam	Audio beam former	833
4	cjpegtransupp	JPEG image transcoding routines	608
5	cjpegwrbmp	JPEG image bitmap writing code	892
6	epic	Efficient pyramid image coder	451
7	fmref	Software FM radio with equaliser	680

Table 4.1: TACLebench benchmarks used

The benchmarks, which are all written in ISO C99, have no dependencies on operating system code or on system specific header files and where input is simulated it is included as part of the code and similarly library functions are provided as C source code. Hence the benchmarks should be fully portable for testing on a wide variety of systems and in our case allows us to focus on the efficiency of the memory management in partial compared to traditional rather than being concerned if any other factor is likely to have an impact.

The full suite consists of over 50 benchmarks, divided into 5 series: a 'kernel' series of mathematical tasks, a 'sequential' series described in [58] as designed to "implement large function blocks, such as encoders and decoders, which are used in many embedded systems", a 'test' series designed to test WCET analysis tools, two benchmarks form a 'parallel' series and two form an 'application' series. To generate a broad range of comparable timings we want to run an instance of each benchmark on every notional row of our (8×16) NoC and so we are restricted to eight benchmarks and we picked from the "sequential" series as the most appropriate to match our likely use case. There are 23 benchmarks in this series and we picked a mixture based on code size and memory usage (see Tables 4.1 and 4.2).

We adapted each of these benchmarks to run on the RISC-V [11] Spike [160] simulator[5]. These were to add makefiles, a C 'runtime' (which provides entry point code that initialises the registers and the stack for the processor) for RISC-V and code to support the Spike simulator's simple system call mechanisms (we do not use this code in production) and a linker script. The C runtime code - which is common to all the compiled benchmarks can be seen as lines 3 - 89 of the assembly listed in Appendix B.

5 Modifications to the benchmarks needed to compile them on RISC-V can be seen at the Github repository at https://github.com/mcmenaminadrian/taclebench-riscv-baremetal/tree/riscv

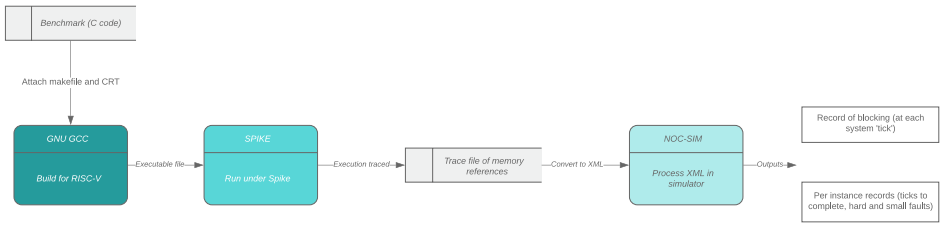

Figure 4.6: Outline of how a benchmark is processed

These traces were then converted to XML (again using the LackeyML format[6]) which was then processed by our simulator which parses the XML. So we get a single XML file for each of the eight benchmarks but we process this in parallel 16 times while our simulation (NOC-SIM) is running.

Appendix B gives more detail on the output of the conversion process. Instructions and memory loads and stores generate separate lines of XML.

The benchmarks are all self-contained and make no library calls, so can be thought of as running on "bare metal". Table 4.2 shows detail of the code and of the range of memory accessed for each benchmark. It can be seen how much switch to 1KB page sizes potentially limits internal fragmentation of pages, with much more concentrated use of a page.

Some further data on the selected benchmarks[7] can be found at `https://www4.cs.fau.de/Research/TACLeBench/`. This shows that the Epic benchmark is the most complex with 471 different code paths, uses floating point arithmetic, has a maximum call stack depth of 5 and has a total of 39 loops (of which 29 are nested). In contrast the Adpcm_dec program does not use floats or loops, has 5 different code paths and a maximum call stack depth of 3.

Table 4.3 shows the theoretical minimum execution time for each benchmark, based on each memory reference requiring one cycle: i.e. an instruction referencing an immediate or a register takes one cycle (and produces one line of XML) while an instruction requiring a memory load or store generates 2 lines of XML and will take two cycles to execute[8].

6 The DTD for this is at `https://github.com/mcmenaminadrian/lackey_xml/blob/master/lackeyml.dtd`
7 The lint test for the fmref benchmark failed and so no data is available.
8 This assumes all instructions and memory references are aligned inside 16 byte boundaries - a reference that crosses such a boundary would require a second cycle.

Number	Name	Source code length (lines of C)	Bytes of instructions referenced on execution (non-uniquely)	1K pages - pages touched	1K pages - 16 byte lines touched per page (mean)	4K pages - pages touched	4K pages - 16 byte lines touched per page (mean)
0	g723enc	898	3469176	18	34.2222 (53.5%)	7	88 (34.4%)
1	admpcm_dec	434	1024732	16	33.25 (52.0%)	7	73.5714 (28.7%)
2	admpcm_enc	406	1028408	16	34.25 (53.5%)	7	72 (28.1%)
3	audiobeam	985	1700148	38	44.4474 (69.4%)	14	118.571 (46.3%)
4	cjpegtransupp	1599	21872284	17	40.8235 (63.8%)	7	95.5714 (37.3%)
5	cjpegwrbmp	1296	954712	19	37.974 (59.3%)	9	79.6667 (31.1%)
6	epic	994	42479572	46	53.5435 (83.7%)	14	176.071 (68.8%)
7	fmref	680	1964932	32	27 (42.2%)	11	76.1818 (29.8%)

Table 4.2: Size and memory performance of selected benchmarks

Benchmark	Instructions	Stores	Loads	Minimum execution time (cycles)	Writes (stores) as share of memory accesses
0	867,294	114,153	280,845	1,262,292	9.0%
1	256,183	38,436	82,272	376,891	10.2%
2	257,102	38,560	82,618	378,280	10.2%
3	425,039	70,102	163,426	658,567	10.6%
4	5,468,071	764,038	2,063,754	8,295,863	9.2%
5	238,678	34,496	59,125	332,299	10.4%
6	10,619,893	90,596	2,727,093	13,437,582	0.7%
7	491,233	59,230	185,595	736,058	8.0%

Table 4.3: Theoretical minimum execution times (in cycles) for each benchmark

4.3.1 Benchmark working set sizes

We can measure the working set of each of the benchmarks by choosing a working set "window size" (a measure of execution time - which here we proxy for with bytes of instructions), and a page size, and measuring how many pages were accessed in that space [47]. At the limit the working set would be the full range of pages accessed: e.g., for benchmark 6, with 1KB page sizes and a working set window of 42,479,572 bytes of instructions, we can see from Table 4.2 that the working set would be 46 pages. More generally, with a smaller window size, the working set size may on average be smaller, though the pages within it will change and in periods of *phase transition* the working set size may significantly increase as pages from different phases of locality are referenced.

Figures 4.7 - 4.14 show the working set sizes of the benchmarks (in 1KB pages) with a working set window size of 100,000 bytes of instructions.

We can immediately see that benchmarks 0, 6 and 7 have working sets which exceed the local memory capacity of our cores (even before we account for local page tables, stacks and other essential calls on local memory) and so will likely display some characteristics of thrashing.

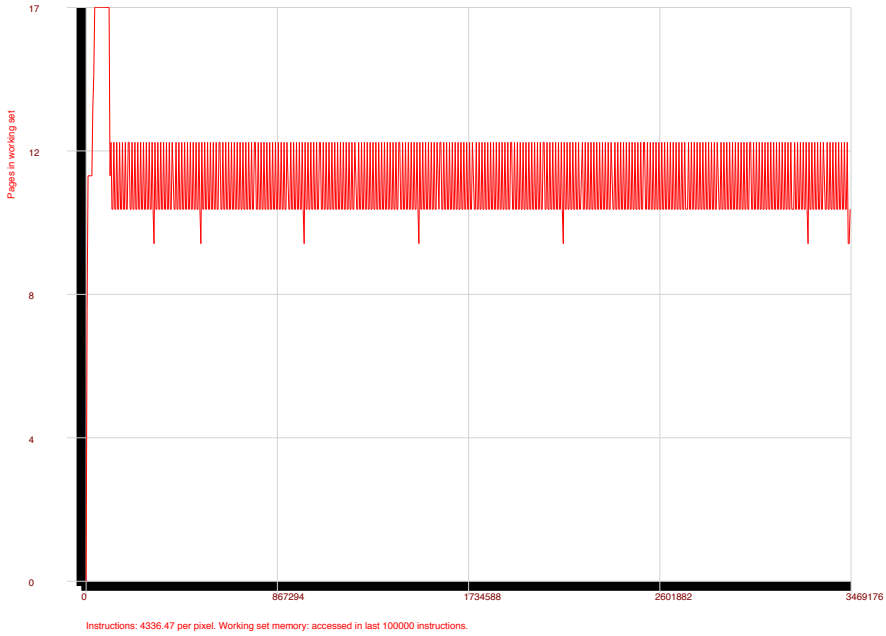

Figure 4.7: Working set size for benchmark 0 with 1KB pages

4.4 OVERALL PERFORMANCE OF PARTIAL PAGING WITH 1KB PAGES

With 1KB pages and 16KB of local memory we set aside one page frame of local memory for notional operating system or systems management software (page frame 0), one page frame for local page tables (page frame 1), one page frame for bitmaps (page frame 2) and finally one page frame for a notional local stack or storage area for the systems software (page frame 15) - so leaving 12KB of local memory for holding application code and data. This is the scratchpad-like space into which we move remapped code and data from the global store.

At the beginning of execution we assume that all needed operating system or systems software is present and that a pristine page table (with 12 empty and invalid entries) and empty bitmaps are also present. No other code or data is present though: pages 3 - 14 are marked as invalid in the page table and in TLBs and the bitmaps for these pages also mark them as empty.

This means that the first instruction of every benchmark generates a hard fault as the address for the code is not mapped. As a result 128 requests to read remote page tables are issued and the tree quickly blocks. Each processor has to issue a further three requests to read down the hierarchy of the remote page tables before each then issues a request to load

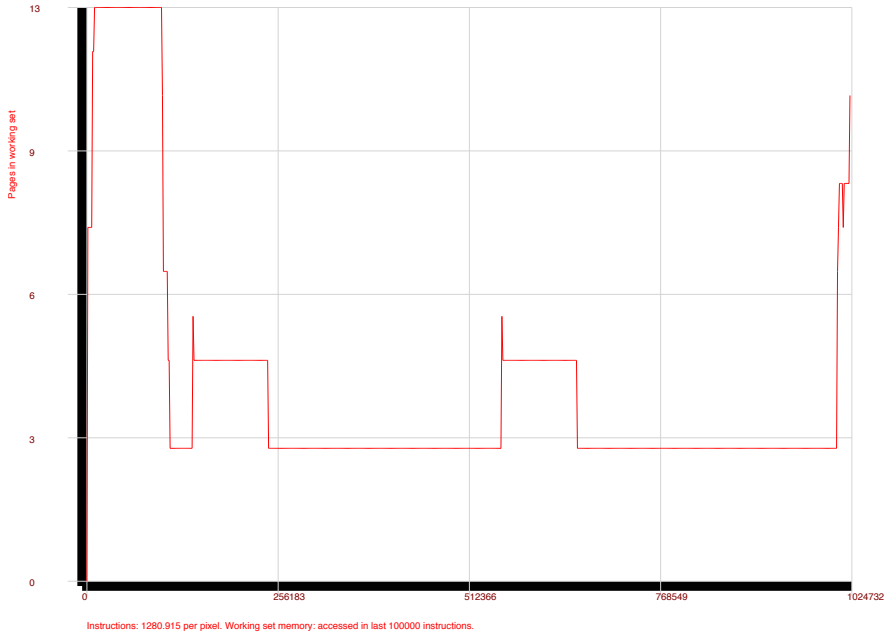

Figure 4.8: Working set size for benchmark 1 with 1KB pages

some remote memory. The benchmark initialisation code (see Appendix B) quickly jumps to another page, requiring a further read of the global page tables and a second hard fault. The subsequent code is sequential, so generating many small faults: the tree remains congested for some time - as is illustrated in Figure 4.15 before becoming less congested after around 300,000 notional cycles have elapsed.

Across a longer period of execution the number of blocks remains high and relatively stable (see Figure 4.16). It should be noted that with the mean number of blocks in the tree is (as marked[9]) ≈ 80 then for 1.9×10^8 "wall clock" cycles a total of 2.4×10^{10} processor cycles will have elapsed across our 128 core device but $\approx 1.5 \times 10^{10}$ (63%) are idle cycles due to blocking in the tree.

Figure 4.17 shows plots for completion times against blocked packets for all eight benchmarks and shows that, for each individual benchmark there is a strong linear relationship between the number of blocked packets and the overall completion time for the benchmark. Using the R programming language's linear regression modelling we can see that a (statistically highly significant) linear relationship between blocks and overall performance exists for all the benchmarks and that the coefficients are all similar (Table 4.4).

9 15,316,137,781 blocks over 192,454,954 ticks

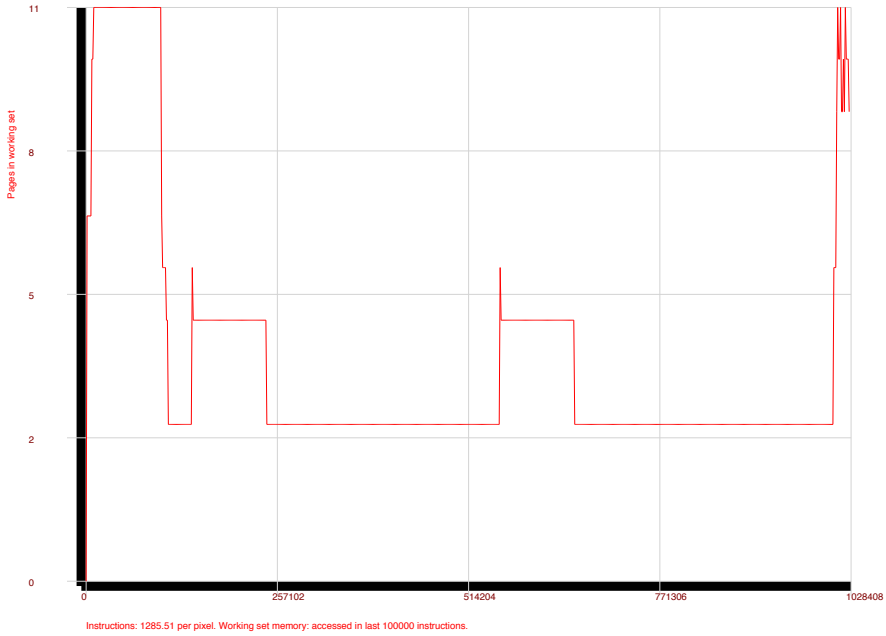

Figure 4.9: Working set size for benchmark 2 with 1KB pages

Here we model execution times as a dependent variable of the number of blocked packets and it can be seen that each time a packet is blocked the execution time rises by approximately 1.1 cycles (1.0 in the case of benchmark 6). In Table 4.4 the intercept represents the modelled time of execution if there were no blocks. Hence the proportion measure is to some degree a benchmark of the crude efficiency of the CLOCK system but also of the locality of memory accesses: if CLOCK minimises the need to load pages or if the reference string of the benchmark is highly localised then the ratio will approach 1. The null hypothesis is that completion times are independent of blocked packets and the significance measures give us good reason to reject this. Plotting the relationship between blocks and completion times for all benchmarks on a single chart (as in Figure 4.18) further illustrates this relationship.

4.4.1 Fault rate and blocks per fault

Figure 4.19 shows that benchmark 4 has a significantly lower fault rate (calculated combining hard and small faults) than the other benchmarks, but that each fault it causes is typically just as expensive as those generated by other benchmarks: indeed the chart shows that the cost of each fault, for any benchmark, is high.

Benchmark	Observations	Intercept	Std. Error	Significance[10]	Proportion[11]	Coefficient	Std. Error	Significance
0	64	1.627×10^6	1.900×10^5	4.18×10^{-12}	1.29	1.134	5.298×10^{-3}	$< 2 \times 10^{-16}$
1	2559	4.570×10^5	6.168×10^2	$< 2 \times 10^{-16}$	1.21	1.094	8.819×10^{-4}	$< 2 \times 10^{-16}$
2	2681	4.558×10^5	5.327×10^2	$< 2 \times 10^{-16}$	1.20	1.097	8.244×10^{-4}	$< 2 \times 10^{-16}$
3	561	1.048×10^6	1.245×10^4	$< 2 \times 10^{-16}$	1.59	1.056	3.047×10^{-3}	$< 2 \times 10^{-16}$
4	283	9.728×10^6	2.393×10^3	$< 2 \times 10^{-16}$	1.17	1.119	2.899×10^{-3}	$< 2 \times 10^{-16}$
5	1764	4.270×10^5	1.681×10^3	$< 2 \times 10^{-16}$	1.28	1.098	1.398×10^{-3}	$< 2 \times 10^{-16}$
6	16	2.694×10^7	1.878×10^6	9.14×10^{-10}	2.00	1.048	1.692×10^{-2}	$< 2 \times 10^{-16}$
7	671	1.046×10^6	7.323×10^3	$< 2 \times 10^{-16}$	1.42	1.081	2.268×10^{-3}	$< 2 \times 10^{-16}$

Table 4.4: Modelling execution times as a dependent variable on blocks in the 1KB partial paging system

[10] Smaller numbers indicate more significant results
[11] The ratio between Intercept and minimal theoretical execution time

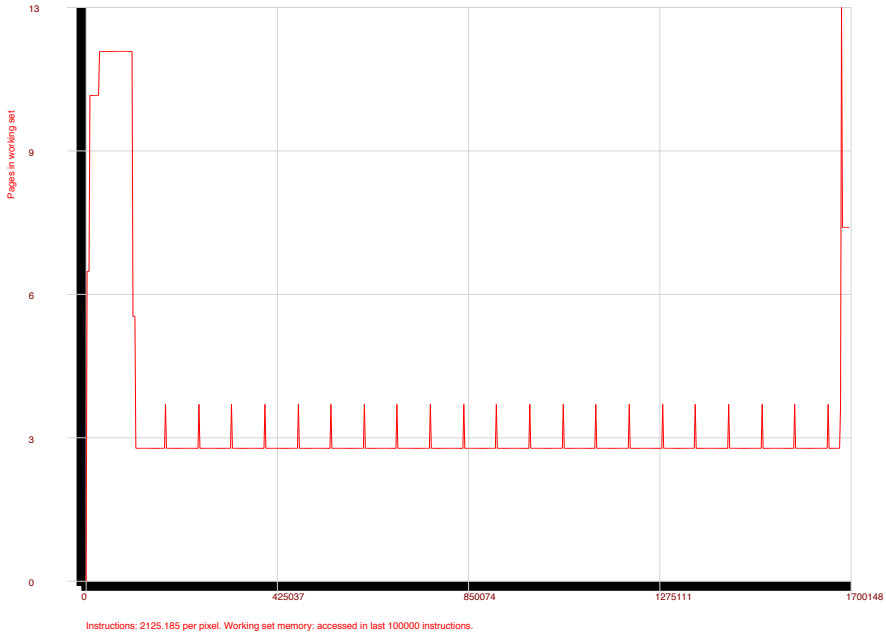

Figure 4.10: Working set size for benchmark 3 with 1KB pages

The number of blocks is dependent on the mix of dirty (written to) and clean (only read) pages that are faulted out. Clean pages do not need to be written back to main memory on replacement (and so don't face a double-jeopardy effect in terms of tree congestion). Hence the "blocks per fault" recorded here is effectively an upper limit on the cost of a fault. Tables 4.5 and 4.6 (specifically the high cost of servicing memory requests for the given fault count) re-enforce the evidence there that Benchmark 3 replaces many dirty pages (and has the highest cost per fault as a result).

The negative slope seen in Figure 4.19 for most benchmarks seems initially puzzling: a lower fault rate leads to more blocks per fault when the lowered congestion might be expected to deliver faster traversal of the tree. But as Figure 4.20 makes clear, the number of blocks per fault is generally not dependent on the total number of faults and what we see in Figure 4.19 is that it is the additional time taken to process memory requests when blocking is high is lengthening execution times and so lowering the fault rate as measured in faults per tick.

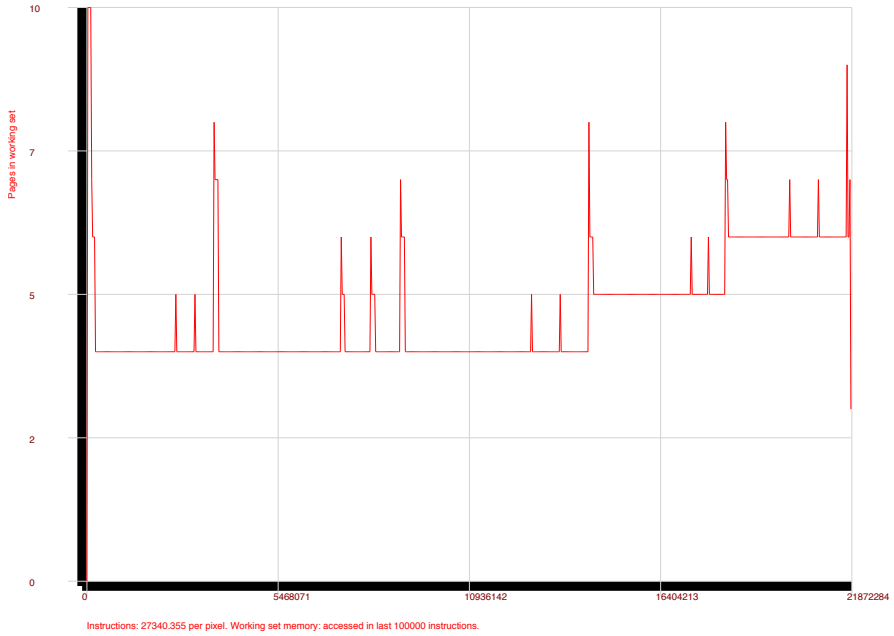

Figure 4.11: Working set size for benchmark 4 with 1KB pages

4.5 THE PERFORMANCE OF 'TRADITIONAL' PAGING

A comparison with the number of blocks seen with a traditional 1KB paging mechanism illustrates the potential for performance improvements that come with partial paging (Figure 4.21): here the long-term mean number of blocked packets is higher at $\approx$ 98 blocks[12] on every cycle - indicating processors are idle 76% of the time while waiting on blocks in the tree.

Figure 4.22 shows the excess (in comparison to partial paging) blocks using a traditional whole page approach: in the first 190 million cycles the mean excess blocking in traditional paging over partial paging is $\approx$ 18.38 blocks per cycle.

Appendix C considers a linear model of benchmark performance for traditional paging (cf. Table 4.4).

12 21,685,673,693 blocked packets over 221,592,468 cycles

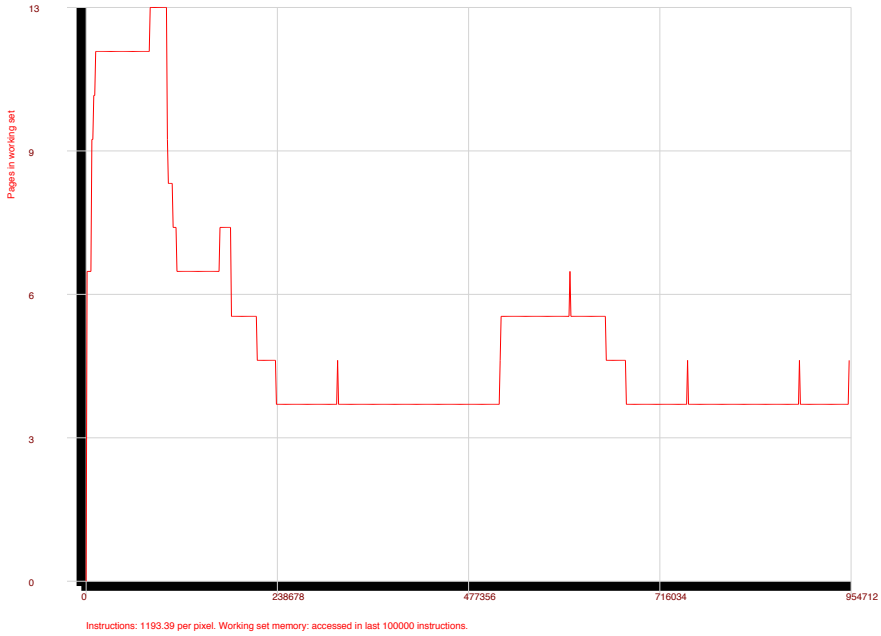

Figure 4.12: Working set size for benchmark 5 with 1KB pages

4.6 COMPARING THE COSTS OF TRADITIONAL AND PARTIAL PAGING

While excess blocking with traditional paging is an indication of likely advantages of the partial paging method, it is not sufficient to suggest partial paging will be more efficient. Additional factors to be considered include the extra time required to handle the small faults in the partial paging model, the difference in time taken to handle hard faults (in that the partial paging model needs to also update bitmaps), the difference in service times at the memory management unit (as the traditional approach has to serve all sixty-four 128-bit 'lines' from the page) and the additional time it takes for these lines to return to the processor (although they do not block they still incur a transport time).

We now examine the evidence we have collected on each of these factors.

4.6.1 The cost of small faults

As discussed above (3.1.3), in a typical VM-using computer system every memory read or write is checked for a TLB hit. In modern, fast, high-end, CPUs it does not always make sense to think of a TLB look-up as sub-cycle as even a basic (L1) lookup in a very fast, pipelined

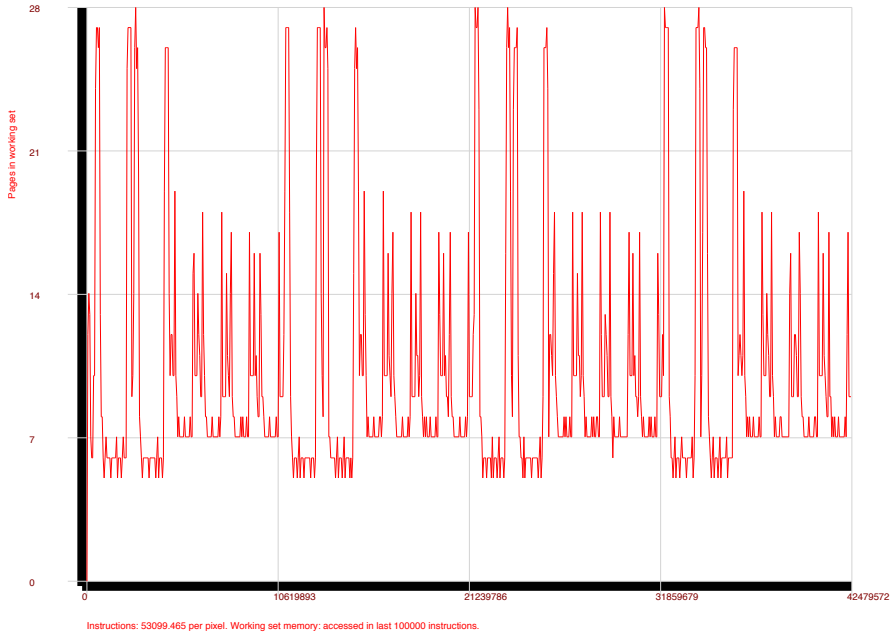

Figure 4.13: Working set size for benchmark 6 with 1KB pages

processor could take more than one cycle and, with TLBs typically set out in a hierarchy, a miss at L1 can prompt a more costly lookup at L2 or even L3 [140]. But our assumption here is that we are using slower CPUs in a NoC environment with a small number of TLBs that can manage sub-cycle lookup: our calculation tabulated in Table 2.1 suggests that cores in a 128 core NoC might run at around $\frac{1}{64}^{th}$ of the frequency of a single core (if we assume that performance and frequency are linearly related), suggesting a single 3GHz core devolving to 128 cores each running at about 47 MHz and while this is a simplification the general rule will be a many-core system will see substantial drops in processor frequency. The complex circuitry likely to be used for such high-speed lookups do come with a high cost in energy however [33].

If we want to make bitmap checking sub-cycle also, then we will need to check both the page part of the address and the lower part of the address (at the granularity of the partial page - in our case 16 bytes) in parallel. We do not consider the hardware question in any depth here but, as discussed in 4.2.3 above, this would require additional circuitry and, if to be done at speed this may have a high energy requirement. We might expect such specialised circuits to consume a similar amount of power and chip area as TLBs. Alternatively power requirements could be lowered at the cost of raw speed and, as we show in 4.7, in a highly-congested

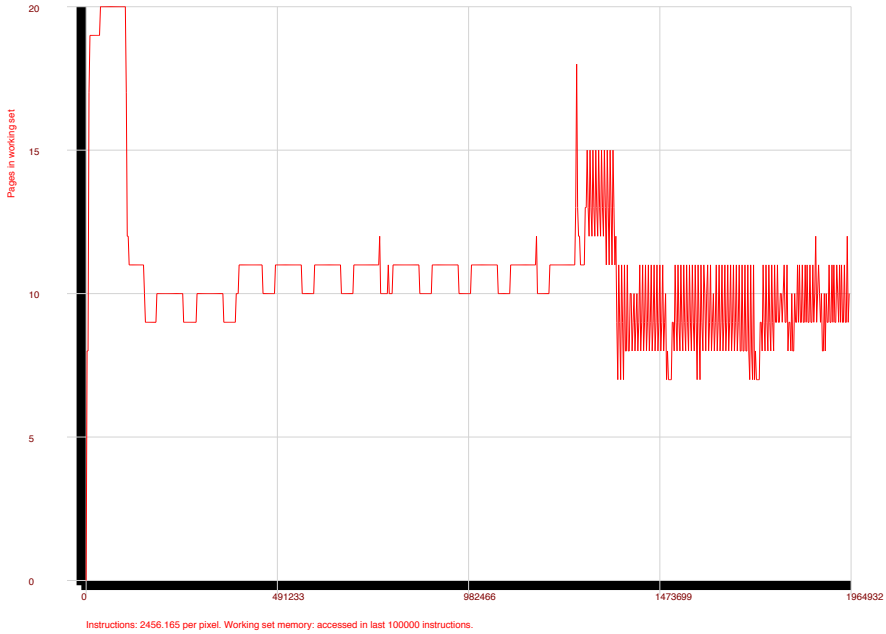

Figure 4.14: Working set size for benchmark 7 with 1KB pages

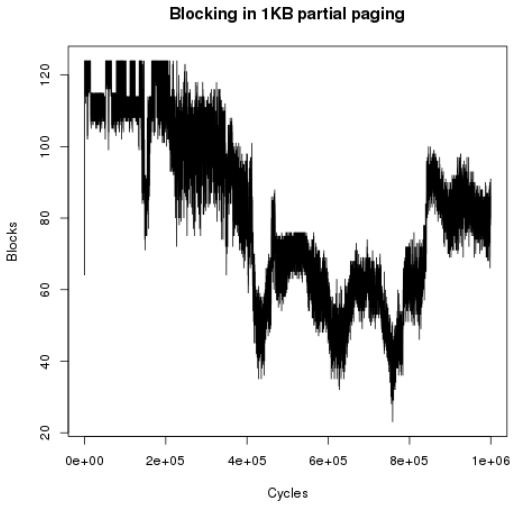

Figure 4.15: Congestion in the memory tree as execution begins with partial paging

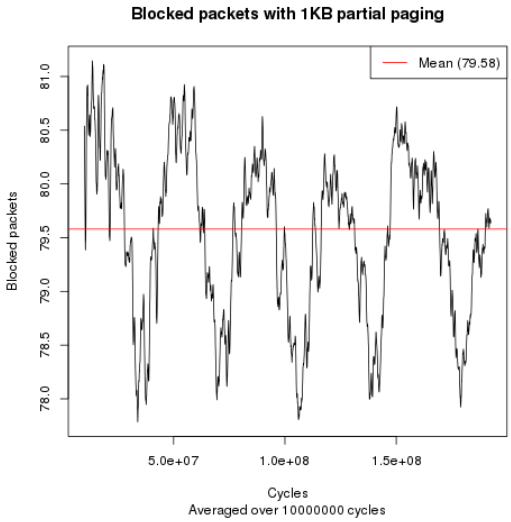

Figure 4.16: Blocks across benchmark execution

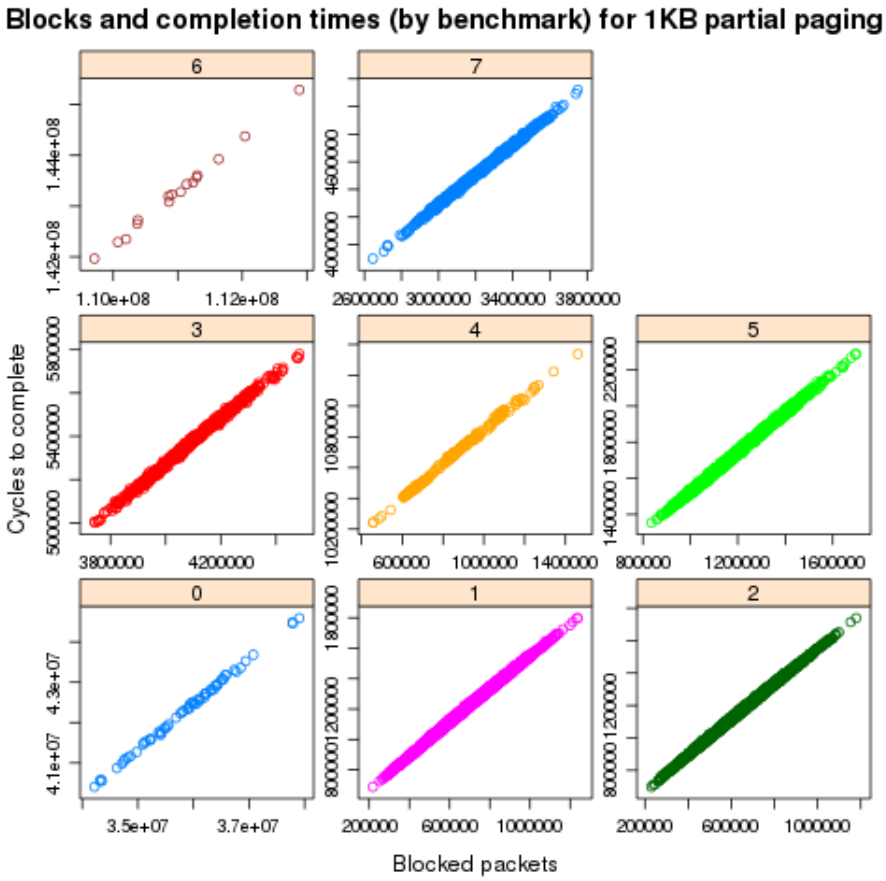

Figure 4.17: Completion times and blocked plackets for all eight benchmarks with partial paging

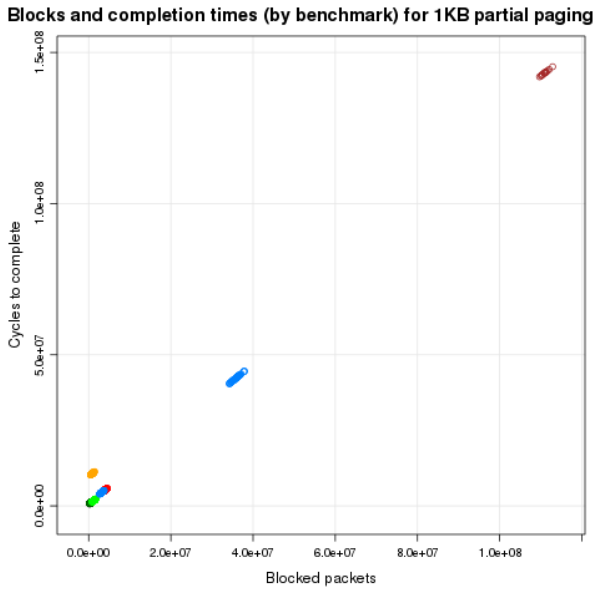

Figure 4.18: Completion times and blocked packets for all benchmarks

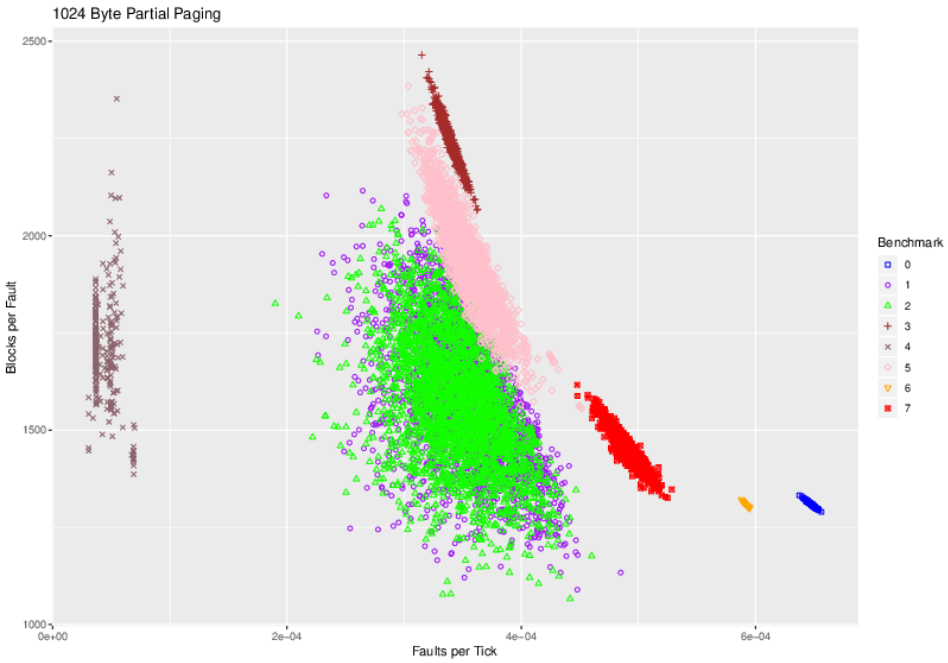

Figure 4.19: Blocks per fault and faults per tick for 1KB partial paging

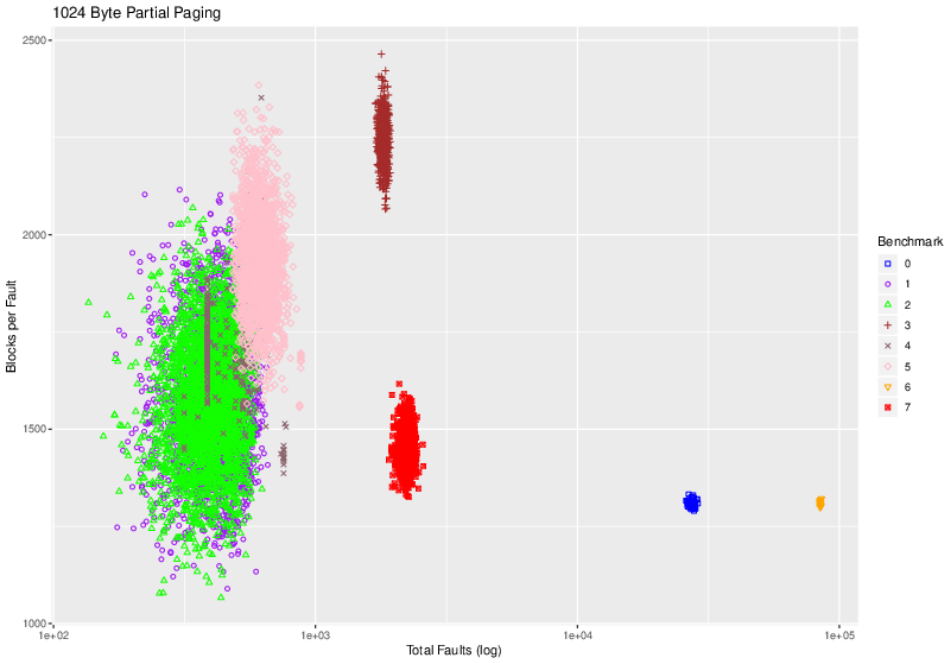

Figure 4.20: Blocks per fault and total faults compared for 1KB partial paging

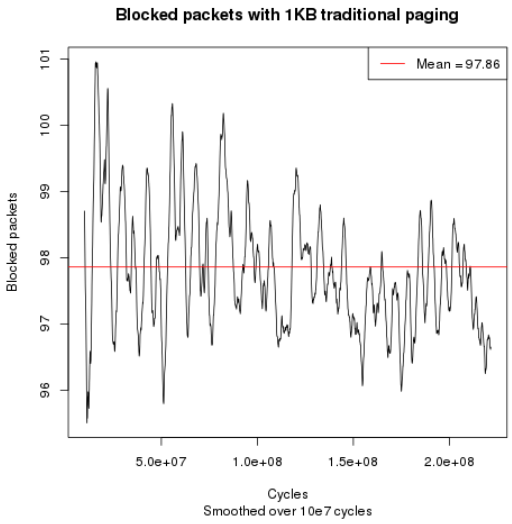

Figure 4.21: Blocks across benchmark execution with 'traditional' paging

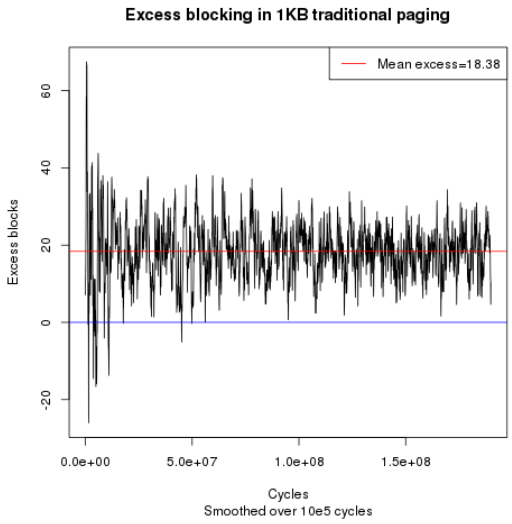

Figure 4.22: Excess blocks with traditional paging compared to partial paging with 1KB pages

environment such a slow-down can also function as a form of load management and deliver performance improvements in some circumstances.

As we note above in 4.2 we are not aware of any existing hardware that allows for the small fault model and a software approach could be attempted. In [110], we made a simple modification to the instruction-accurate OVPSim Microblaze simulator to model small faults as interrupts handled in software. In that case 54 machine code instructions had to be executed[13] when the accessed memory was already present and marked as such in the bitmap. In the case when new memory has to be loaded this rises to 138 instructions - to both load the memory and update the bitmap.

Although in that case, which involved a single processor and so no congestion in the memory connect, the software-driven partial paging approach did, using a memory reference pattern generated from a partial trace from the PARSEC benchmark suite (the same x264 video processing benchmark described in 3.3 on page 62 which has a very high fault rate for small memories), deliver better estimated performance when normalised for the cost of memory transport, such a 5400% increase in the cost of a typical instruction would not deliver better performance in the many-core system with the benchmarks used here.

13 The code path from line 256 here https://github.com/mcmenaminadrian/mb_boot/blob/newmodel/startup.S

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	43361267	1428821	1400909	5760495	11037274	2072418	145291356	5012280
Mean	42368816	1416176	1388897	5646235	10964717	2026200	143276995	4733750
Service cost	1511834	29000	28575	238075	39044	80825	5996075	172184
Blocks	35921194	870627	844281	4364623	1088954	1446305	110996151	3420685
Count								
Hard Faults	908.1	17.0	17.1	41.3	19.1	23.8	3118.5	103.0
Small Faults (and page table reads)	26570.6	563.0	553.2	1857.3	737.3	852.8	81605.8	2159.8
Share								
Service	3.6%	2.0%	2.1%	4.2%	0.4%	4.0%	4.2%	3.6%
Blocks	84.8%	61.5%	60.8%	77.3%	9.9%	71.4%	77.5%	72.3%
Admin	8.7%	9.9%	9.9%	6.8%	14.1%	8.2%	9.0%	8.6%
Efficiency of execution	3.0%	26.6%	27.2%	11.7%	75.7%	16.4%	9.4%	15.5%

Table 4.5: Performance (means) of benchmarks with partial paging and 1KB pages: initial execution

4.6.2 Other costs

Tables 4.5 and 4.6 show how the various benchmarks performed used partial paging: we have separated the initial run, when every page must be faulted in at least once, from the subsequent executions. The figures show the mean performance across all 16 instances of each running benchmark, as well as the maximum time taken.

The cost of servicing memory requests at the MMU is separately accounted for and the simple relationship in our simulation of one instruction taking one cycle (unless delayed for other reasons) allows us to make some comparisons of the cost of partial paging and traditional paging.

The fundamental weakness of the traditional paging approach in a low memory environment is the need to load (and also, in some cases, write-back) a whole page. If competition for memory is intense then this adds to congestion in the memory interconnect (loading a 1KB page in 16 byte lines requires 64 requests to traverse the network). As Table 4.2 suggests, even if memory were unlimited, a full page loading system will load much memory which is not required by the program. A traditional system will, though, be simpler in that there will be no need to clear and manage a bitmap on a hard fault and the traditional system also has an additional free page as no space is needed for bitmaps.

A key metric is "efficiency of execution" - which is the ratio between the number of lines of XML (i.e. the number of memory references) and the total number of ticks taken to execute

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	44592927	1798061	1739293	5779890	11338555	2288311		5120014
Mean	42266895	1198274	1143068	5351357	10608943	1734867	NA	4526916
Service cost	1514750	25332	23258	238244	31100	75219		171796
Blocks	35831158	677368	626420	4074286	788357	1191021		3219209
Count								
Hard Faults	899.2	12.5	12.1	35.2	9.1	13.5		96.4
Small Faults (and page table reads)	26459.6	412.8	385.1	1781.2	447.1	604.5		2120.7
Share								
Service	3.6%	2.1%	2.0%	4.5%	0.3%	4.3%		3.8%
Blocks	84.8%	56.5%	54.8%	76.1%	7.4%	68.7%		71.1%
Admin	8.7%	9.9%	10.1%	7.1%	14.1%	7.9%		8.8%
Efficiency of execution	3.0%	31.5%	33.1%	12.3%	78.2%	19.2%		16.3%

Table 4.6: Performance (means) of benchmarks with partial paging and 1KB pages: subsequent execution

the benchmark. Each line should take a minimum of 1 tick, so this measure (shown as a mean) provides an absolute performance comparison across the different paging methods. "Service" is the mean proportion of time taken by the MMU to service memory requests (including remote page lookups and page write-backs), "blocks" are cycles lost to congestion in the memory tree, while "admin" is the mean cost of everything else such as servicing faults (other than blocks in the tree and MMU costs) and page replacement mechanics.

Although subsequent runs of the benchmarks deliver better average performance, they generally also show poorer worst case execution times. We consider this further in 4.8.

For partial paging benchmark 0 performs very poorly: as Figure 4.7 suggests the working set size for this benchmark hovers just above and below 12KB. With 12 1KB page frames available the fault rate is very high and processors spend over 80% of their time waiting on memory requests that are blocked by congestion in the memory tree.

Tables 4.7 and 4.8 show the results for the traditional paging approach: for every case except benchmark 0 the performance of the partial paging system is superior, with the traditional approach generating more blocks and requiring a longer service time.

Figure 4.23 shows the relationship between blocking and efficiency in the traditional case and the partial paging case (cf. Figure 4.17). Figure 4.24 compares the cost of servicing memory requests at the MMU with overall efficiency. Under partial paging processors may wait a

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	6408101	3776618	3470041	10811973	13571638	4561929		21204161
Mean	5812902	3268085	3184537	9885640	13419937	4494424	>221592468	19630862
Service cost	155450	93925	89575	333350	117675	160800		774200
Blocks	4163688	2706769	2625695	8695001	3534232	3898527		17778595
Count								
Hard Faults	24.1	17.3	17.1	41.8	19.7	24.0		86.0
Remote Page Table Reads	141.0	90.5	87.5	283.0	109.5	144.0		636.0
Share								
Service	2.7%	2.9%	2.8%	3.3%	0.9%	3.6%		3.9%
Blocks	71.6%	82.8%	82.5%	88.0%	26.3%	86.7%		90.6%
Admin	4.0%	2.8%	2.9%	2.0%	11.0%	2.3%		1.7%
Efficiency of execution	21.7%	11.5%	11.9%	6.7%	61.8%	7.4%	<6.1%	3.7%

Table 4.7: Performance (means) of benchmarks with traditional paging with 1KB pages: initial execution

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	9012279	5697371	5463362	11269496	13822943	4405844		23926071
Mean	6031915	3118898	2994237	9299653	12583912	3021206		20496658
Service cost	161046	97891	93060	319420	94708	106592		827383
Blocks	4374973	2556600	2436348	8131958	2743201	2497983		18583541
Count								
Hard Faults	16.1	9.8	9.3	31.9	9.5	10.7		82.7
Remote Page Table Reads	128.8	78.3	74.5	255.6	75.8	85.3		661.9
Share								
Service	2.7%	3.1%	3.1%	3.4%	0.8%	3.6%		4.0%
Blocks	72.5%	82.0%	81.4%	87.4%	21.8%	82.7%		90.7%
Admin	3.9%	2.8%	2.9%	2.0%	11.5%	2.8%		1.7%
Efficiency of execution	20.9%	12.1%	12.6%	7.1%	65.9%	11.0%		3.6%

Table 4.8: Performance (means) of benchmarks with traditional paging with 1KB pages: subsequent execution

longer proportion of execution time waiting for memory requests to be served, but that reflects an overall higher level of efficiency.

Figure 4.24 also shows the relatively fine grained memory access pattern of the partial paging approach: in traditional paging the points on the chart are seen to be arranged in lines (sloping from lower left to upper right) - each line represents a specific hard fault count and the line slopes up to the right in reflection of the impact of blocking on performance (as the absolute time of serving a number of hard faults is effectively fixed, low blocking leads this time to take up a greater proportion of overall time of execution). For partial paging the mix between hard and small faults means this sharp definition is absent (though a general 'tilt' towards the right is still present).

The plots show how dominant blocking is, in general, in determining overall performance - the small share of time taken up by servicing requests at the MMU is only a weak predictor of overall efficiency of performance.

4.7 SIMPLE LOAD CONTROL BY ADDING COSTS FOR BITMAP READING

Denning's [47] formulation of the practically-optimal page replacement policy - the working set policy - emphasises the importance of load control. Too high a load can lead to a very rapid decline in performance - as suggested by the near vertical section of Figure 3.2 at higher levels of traffic intensity.

For our system, adding a cost for bitmap reading is a simple way to exercise some load control as it effectively prioritises remote memory reads and writes over routine execution: packets traverse the memory tree at the same rate as before (one hop per cycle if not blocked) but other processes effectively run 100% slower and so the level of traffic in the memory tree is reduced.

Tables 4.9 and 4.10 show the results of applying this simple discipline.

It can be seen that half of the benchmarks (namely 0, 3, 6 and 7) deliver lower observed maximum completion times under this discipline (i.e., in comparison to undelayed partial paging results tabulated in 4.7 and 4.6). The lower level of congestion in the memory tree (as measured by the fall in blocking) allows them to complete more quickly: if we had scheduled the benchmarks as a set then this simple system of load control would, it appears, be likely to give us better results.

For a limited power budget these results suggest that, with programs that demand significantly more memory than is available locally, the correct balance is likely to be in favour of a

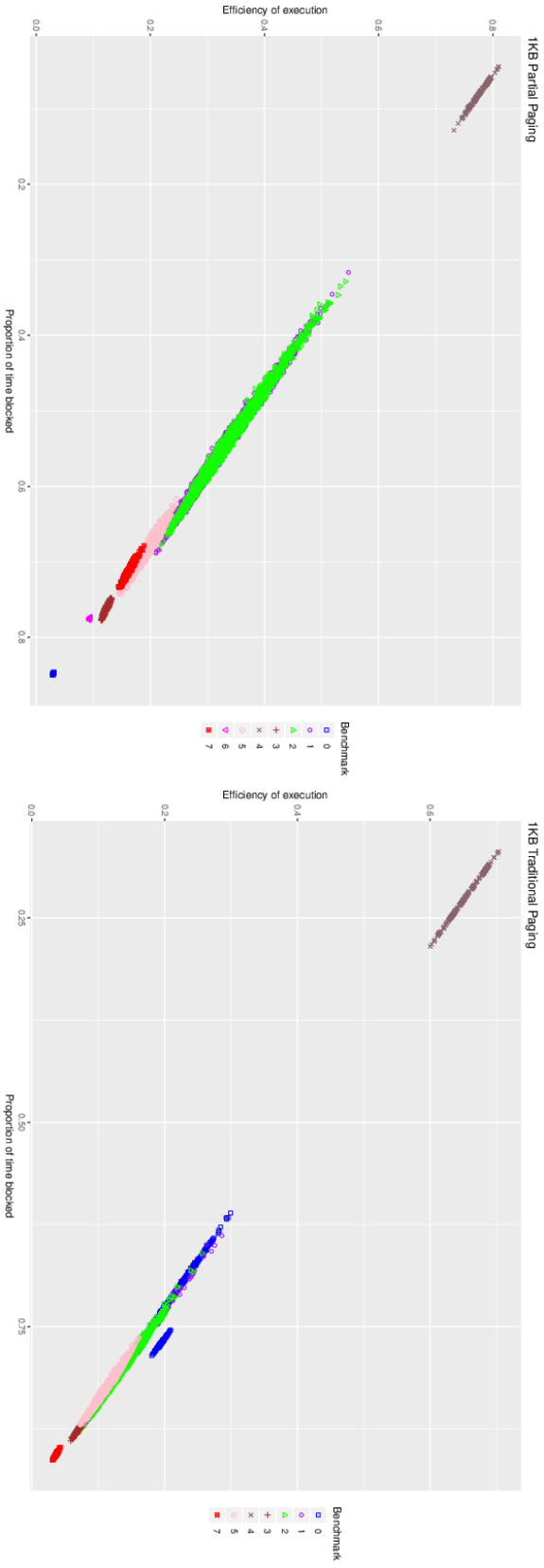

Figure 4.23: Performance of traditional and partial paging compared

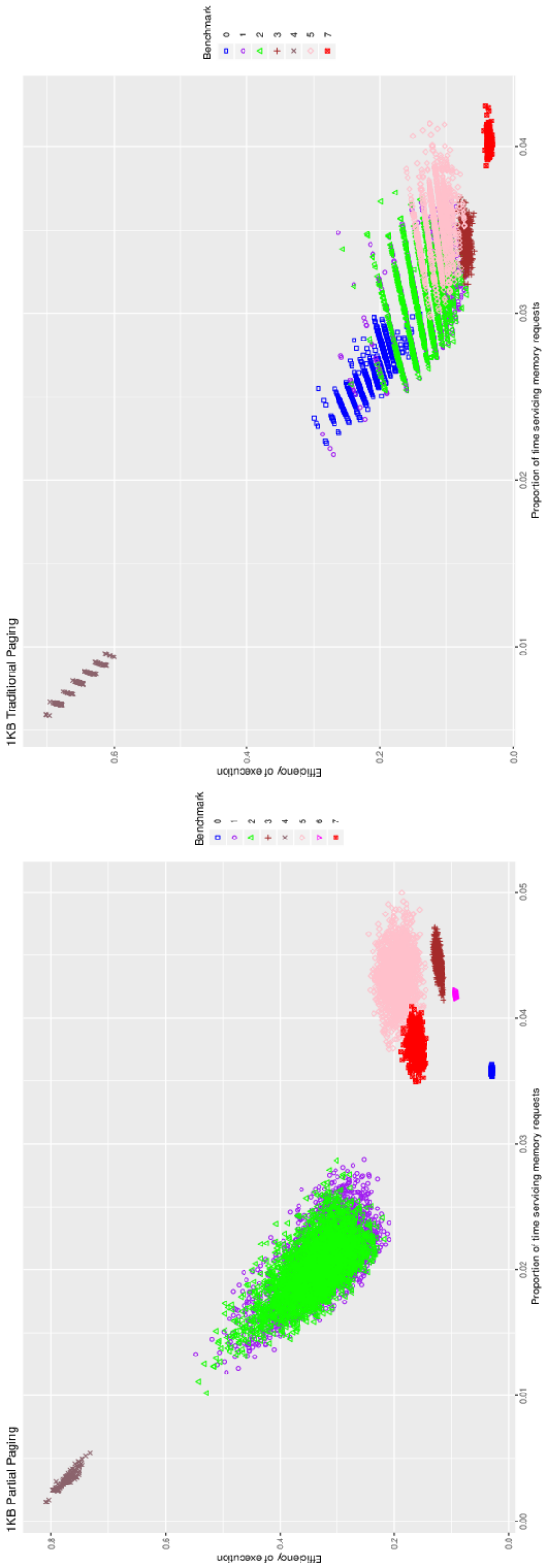

Figure 4.24: Share of time processes require to service memory requests and are blocked in memory tree compared for 1KB paging

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	31958315	1831453	1715321	5586937	20676042	2327423	126566200	4885999
Mean	29917689	1752262	1663821	5448885	20533809	2312829	125374957	4690612
Service cost	1381841	30772	28788	238522	38844	82550	5559178	149594
Blocks	23309538	765642	676825	3424605	921608	1338933	78290713	2549331
Count								
Hard Faults	852.6	18.5	17.2	41.0	19.1	24.0	2808.1	90.3
Small Faults (and page table reads)	24817.0	575.6	554.4	1856.6	737.8	855.0	76672.8	1941.4
Share								
Service	4.6%	1.8%	1.7%	4.4%	0.2%	3.6%	4.4%	3.2%
Blocks	74.6%	43.7%	40.7%	62.8%	4.5%	57.9%	62.4%	54.3%
Admin	16.6%	33.0%	34.9%	20.7%	54.9%	24.2%	22.4%	26.8%
Efficiency of execution	4.2%	21.5%	22.7%	12.1%	40.4%	14.4%	10.7%	15.7%

Table 4.9: Performance (means) of benchmarks with partial paging and 1KB pages with 1 tick cost for bitmap reading: initial execution

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	31698367	2002551	1922442	5416176	20636710	2317899		4952256
Mean	29666359	1452111	1404347	4924975	20292171	1836826	NA	4447703
Service cost	1385817	24920	22931	2344931	34401	72057		155958
Blocks	22056760	490272	445997	2919373	714056	907855		2300827
Count								
Hard Faults	840.4	12.1	11.9	32.8	9.9	12.8		86.7
Small Faults (and page table reads)	24748	417.5	387.5	1738.9	512.1	574.7		1933.4
Share								
Service	4.7%	1.7%	1.6%	4.8%	0.2%	3.9%		3.5%
Blocks	74.3%	33.8%	31.8%	59.3%	3.5%	49.4%		51.7%
Admin	16.7%	38.6%	39.7%	22.6%	55.4%	28.6%		28.2%
Efficiency of execution	4.3%	26.0%	26.9%	13.4%	40.9%	18.1%		16.5%

Table 4.10: Performance (means) of benchmarks with partial paging and 1KB pages with 1 tick cost for bitmap reading: subsequent execution

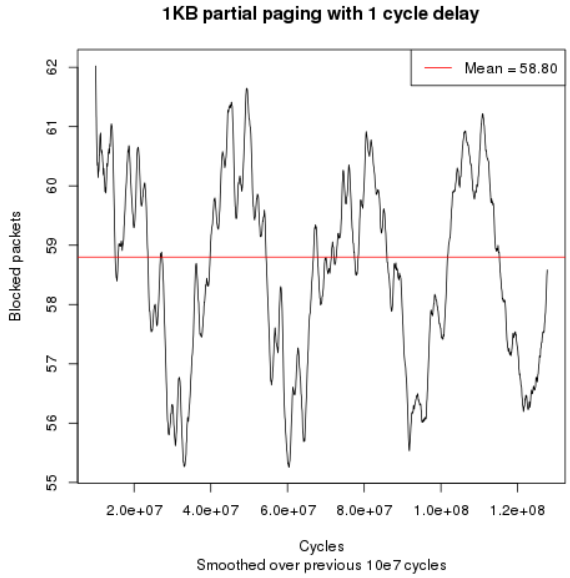

Figure 4.25: Blocks for 1KB page sized partial paging with a one cycle delay for bitmap access (moving average plot)

faster (and more power-demanding) memory connection even if the price of that is slower processors, as the lowered memory congestion may still deliver better results.

As Figure 4.25 shows, this delay reduces the overall blocking in the system: with an average of 58.8[14] blocked processors (45.9%).

Appendix D considers a linear model of benchmark performance for partial paging with bitmap-reading delay (cf. Table 4.4).

Figure 4.26 shows the use of the 1 cycle delay delivers a better minimum efficiency than undelayed partial paging and Figure 4.27 shows delay cuts the number of cycles lost for each hard fault by around 14000. A comparison of Tables 4.9 and 4.5 also shows a reduction in the number of faults for benchmarks 0, 3, 6 and 7 (i.e., those which complete more quickly under this regime). There appear to be several factors contributing to this picture:

- *Less congestion in the memory connect*: as one would expect, effectively increasing the cost of executing a local memory read from one to two cycles - while holding constant the cost of crossing an unblocked memory tree and servicing memory requests at the MMU - reduces congestion in the memory tree.

- *More efficient page replacement via CLOCK*: We are using a simple CLOCK page replacement algorithm [38] and every 1000 cycles of normal execution (i.e., excluding interrupted cycles), one page is marked as available for removal. If that page is sub-

14 7,508,642,841 blocked packets over 127,706,368 cycles

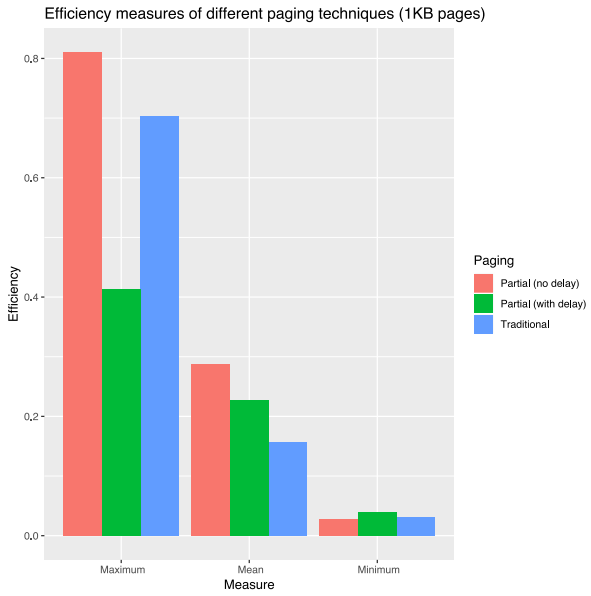

Figure 4.26: Maximum, mean and minimum efficiencies seen with 1KB pages.

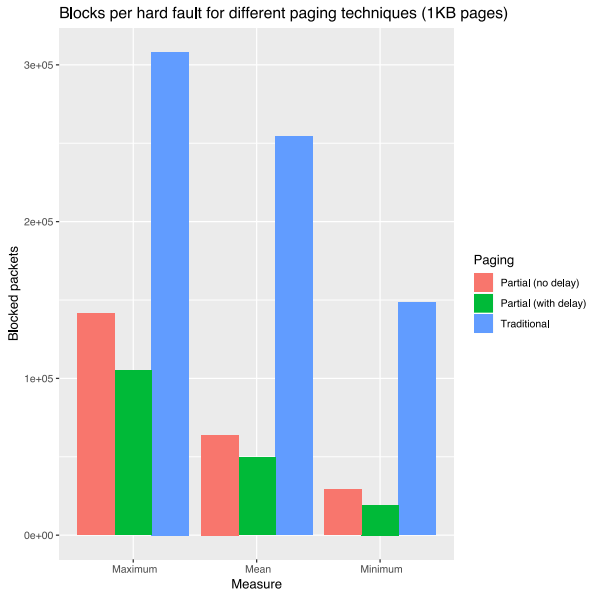

Figure 4.27: Maximum, mean and minimum number of blocked packets per hard fault seen with 1KB pages.

sequently accessed it is marked as unavailable for removal. Then, when a hard fault occurs the page table is walked and, if available, an empty page slot is chosen, or if none are available then a page marked for replacement is chosen. In periods when the fault rate is high it may well be that all pages are marked as in use and so a page has to be picked arbitrarily. Knuth [94] posits an algorithm to generate pseudo-random numbers through a small range of integers but at our range this simply generates a linear sequence which we cycle through. If this method is used frequently then it is quite likely 'bad' pages will be chosen, further increasing the fault rate and adding to congestion in the memory connect. As increasing the time cost of executing an instruction rises with additional cost of bitmap manipulation then the *effective* number of cycles between each 'tick' of the CLOCK shortens and so more pages are likely to be marked as available for removal and bad sequential or pseudo-random choices are less likely to be made. For benchmarks 6 and 7, for instance, there are around 10% more hard faults with the undelayed partial paging approach than with the delayed method.

- *Less efficient page replacement via CLOCK*: Of course there is also a countervailing phenomenon - as the effective time between pages being marked for replacement shortens then, in some cases, the wrong page will be chosen and so the fault rate will be pushed upwards as a page faulted out needs to be faulted back in. We can see some indications of this with benchmark 1.

Together these factors suggest tuning an embedded system's page replacement mechanisms to fit the task and the overall scheduling needs (e.g., to improve the WCET of a typical task or to improve the WCET of the slowest task) remains important.

Figure 4.28 shows, respectively, the number of blocks per hard fault and the recorded maximum cost of completing the initial iteration of each benchmark across the partial paging, traditional paging and partial paging with a one-cycle delay for bitmap access.

In conclusion we can see that a partial paging system used in a real time environment will have a number of options for performance tuning. If memory demands greatly outstrip locally available fast memory then tuning the page replacement mechanism to the task is likely to be very important: bad decisions about which pages to replace will significantly degrade performance. In cases where it is necessary to minimise the response time of a program that makes high demands on external memory then a delay mechanism of the type discussed here, which alternatively can be thought of as devoting more power to the connection tree and less to the cores, could produce better overall results.

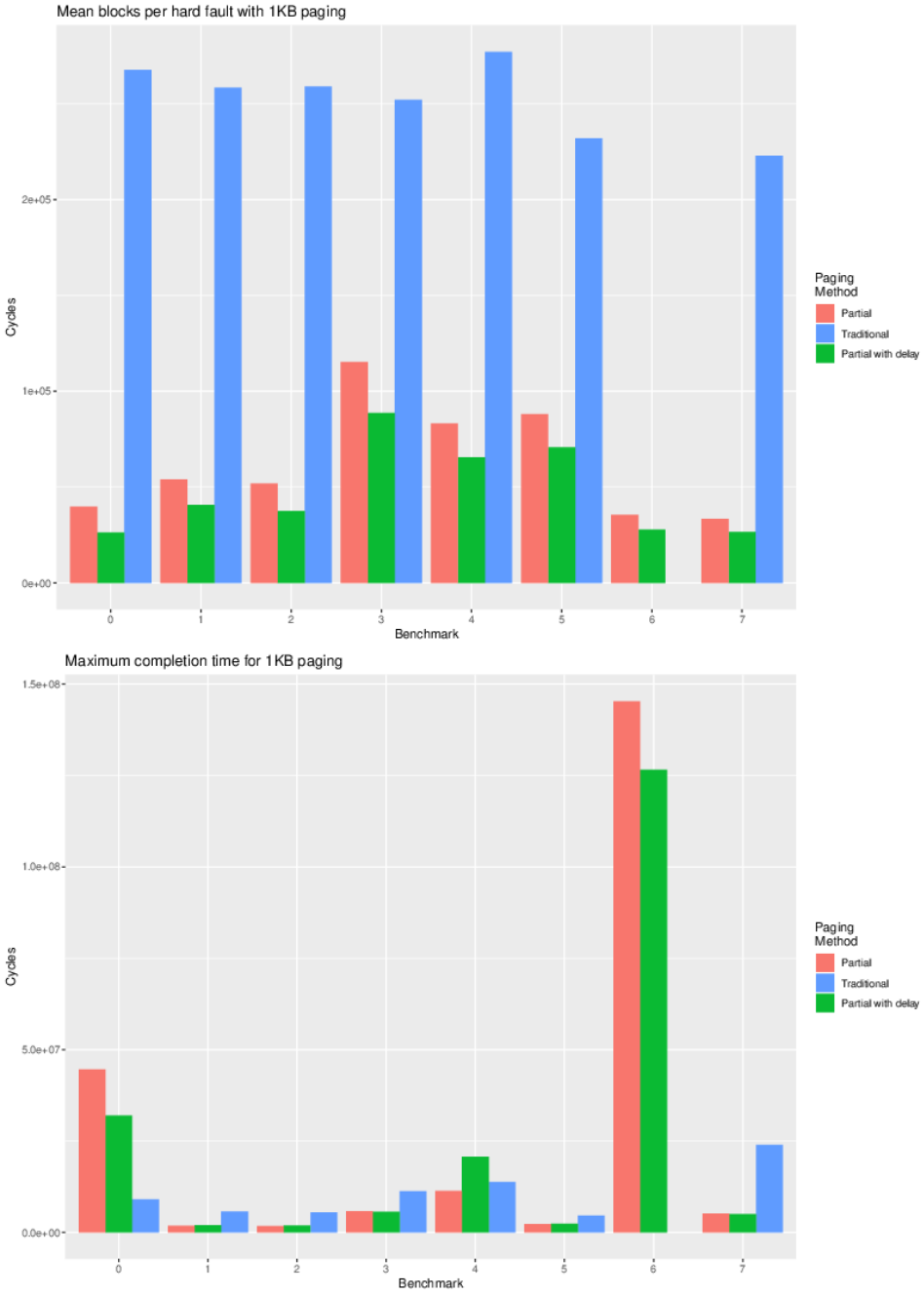

Figure 4.28: Mean blocks per hard fault and maximum completion times compared across benchmarks and paging approach for 1KB pages

4.8 OBSERVED WORST EXECUTION TIMES AND TIMING CERTAINTY

The discussion above generally considers the mean performance of the partial paging system and compares it to a traditional paging system. It shows that for both a partial paging system and a traditional system queuing for memory service - which we generally refer to as 'blocking' in the memory tree - dominates performance (Tables 4.4 and C.1) and that there is a strongly linear relationship between the level of blocking a program experiences and its typical efficiency (see Figure 4.23).

Thus, as these results would lead us to expect, the lower level of blocking in the partial paging system (see Figure 4.22) leads to faster average (and observed worst-case) execution times in almost every case (see Figure 4.28 for a comparison of observed worst case timings). In fact the results show that even if we assume a degree of delay in processing the bitmap reading at the heart of the partial paging system, the lowered blocking (which is reduced further by this delay), partial paging is faster than traditional paging in 6 of the 8 benchmarks we consider (Figure 4.28).

But for real-time systems the degree of timing certainty, and specifically the worst case execution time (WCET), rather than simply the average execution time, is likely to be more important. And we cannot assume that even a sample of several thousand completion times will capture the likely WCET for any program. In fact there is no general method to determine the maximum execution time of a program: finding one would be the equivalent of solving the halting problem [164]. Static analysis of multi- and many-core designs is also regarded as "questionable" because of their inherent complexity [5]. Thus we concentrate on observed program and system behaviour and what inferences we can draw from these observations.

The evidence we have shows we can expect a broad range of timing outcomes even for individual benchmarks. 'Blocks per fault' (BPF) is available to us as an approximation of a normalised measure of the range of timing impacts of blocking on each benchmark and a density plot of this is shown in Figure 4.29. We can only treat this only as an approximation because of the potential differential impact of page write-backs on the ratio between faults and blocks (cf. the discussion in 4.4.1 on page 92). Figure 4.29 (cf. Figure 4.20) shows a very broad range of BPF and we see this reflected in the uncertainty in timing.

This measure is not available for the traditional approach (as there are only hard faults and remote page reads to count) but Figures E.1 - E.8 in Appendix Eshow density plots for the completion time of the various benchmarks when running under partial paging with no delays for bitmap access, traditional whole-page paging and partial paging with an additional 1 cycle delay for bitmap access.

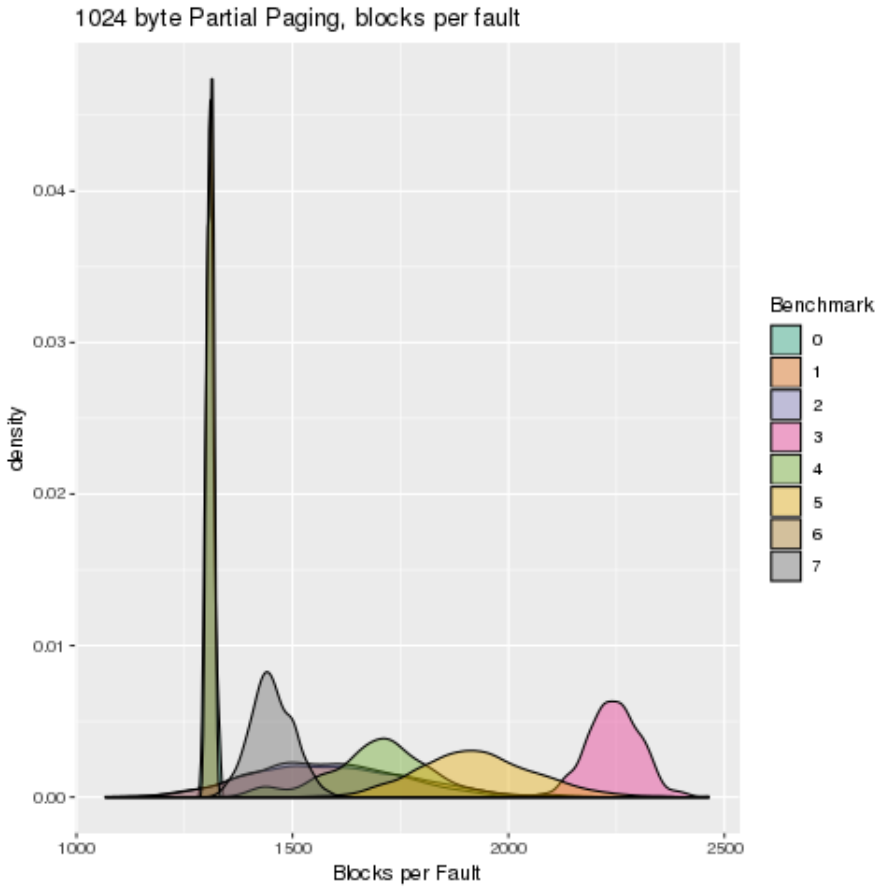

Figure 4.29: Density plot of range of blocks per fault (hard and small) for 1KB partial paging with no bitmap delay

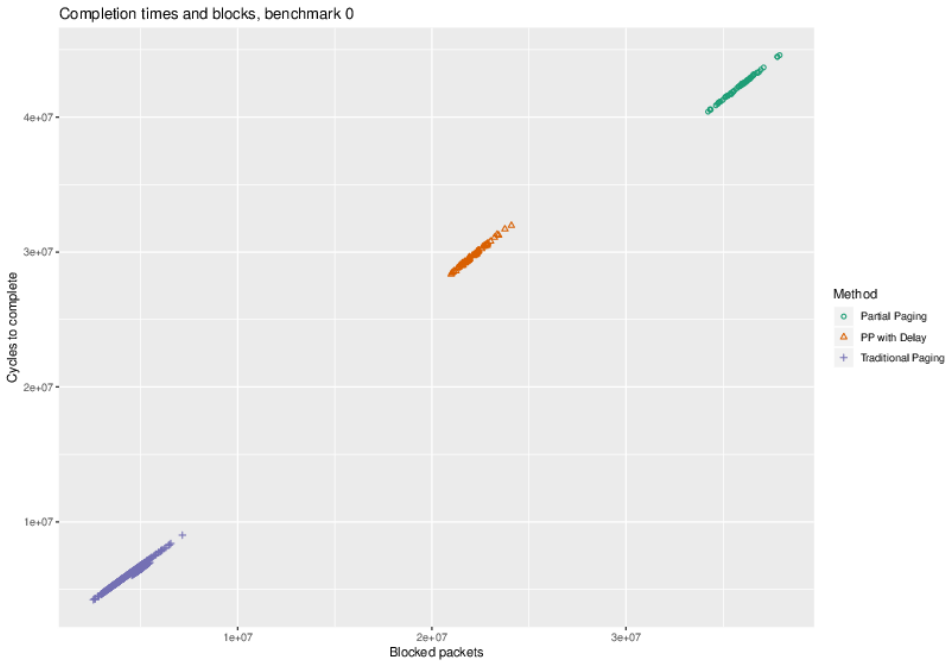

Figure 4.30: Overall cycle count and blocked packets for benchmark 0 with 1KB pages

Figures 4.30 - 4.37 show that not just the mean but also the range of completions times to be a function of the number of blocks in the system - the larger the range of blocks, the greater the range of completion times, regardless of which paging method is employed. They also show - particularly in the case of traditional paging, but for partial paging also - that completion times tend to be skewed to the right, suggesting there is an increased risk that the true worst execution time may be substantially higher than the observed time. Again we only have partial results for benchmark 6, as the traditional approach did not complete in the available computing time.

The performance for benchmark 0 reflects its pathological character under partial paging with 1KB page sizes: the traditional approach easily outstrips the partial paging implementations, even when only considering the observed worst execution time. In almost all other cases partial paging approaches deliver better observed WCETs and in half of cases we can see the undelayed partial paging approach outperforms the delayed version. However the density plots shown here are computer-generated approximate polynomial fits to the underlying data and as approximations and should not be taken as a precise guide to the extremes: we compare the WCETs for different methods with more rigour in 4.9.2.1 and 4.9.2.2 below.

For benchmark 4 (see Figure 4.34), which makes fewer memory requests and so gains only a relatively small advantage from the undelayed partial paging approach, traditional paging easily out performs the delayed version of partial paging.

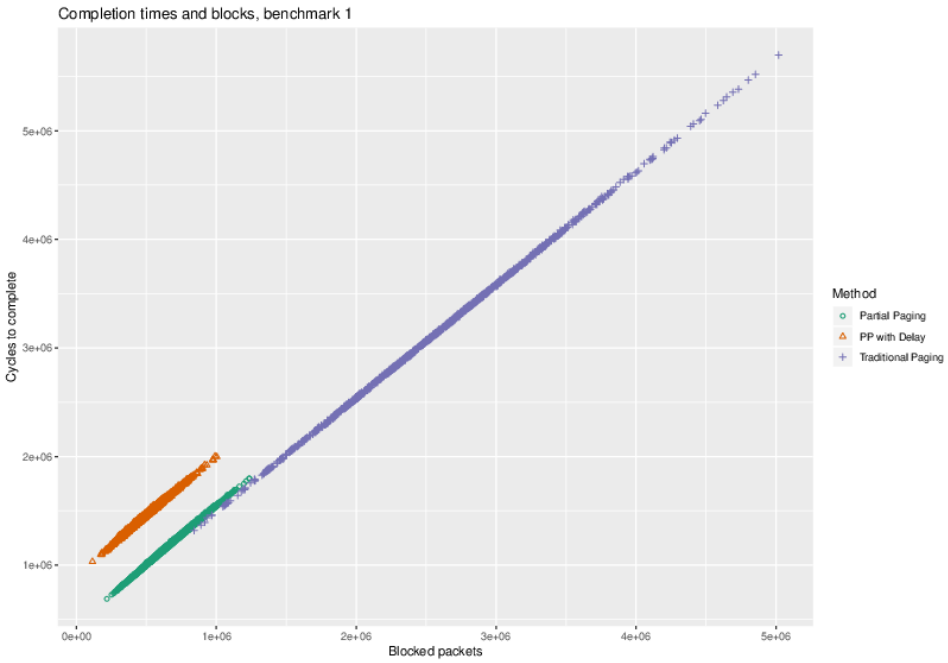

Figure 4.31: Overall cycle count and blocked packets for benchmark 1 with 1KB pages

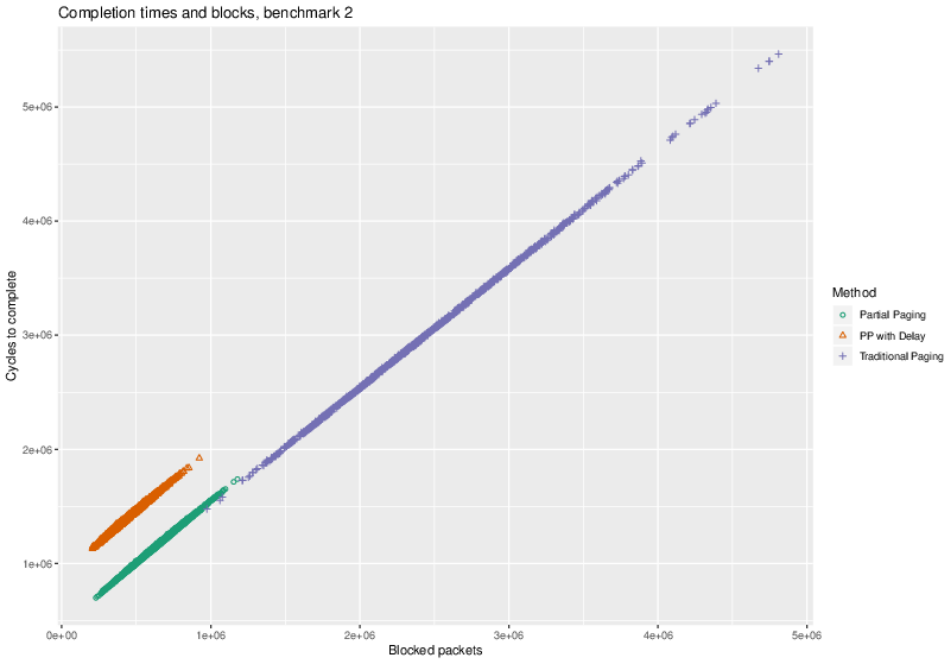

Figure 4.32: Overall cycle count and blocked packets for benchmark 2 with 1KB pages

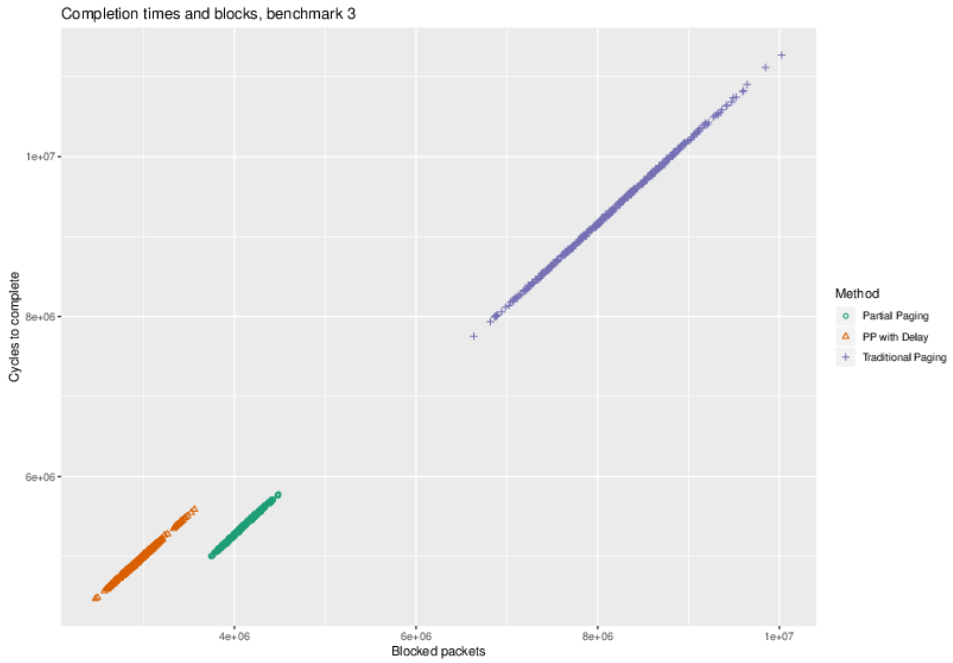

Figure 4.33: Overall cycle count and blocked packets for benchmark 3 with 1KB pages

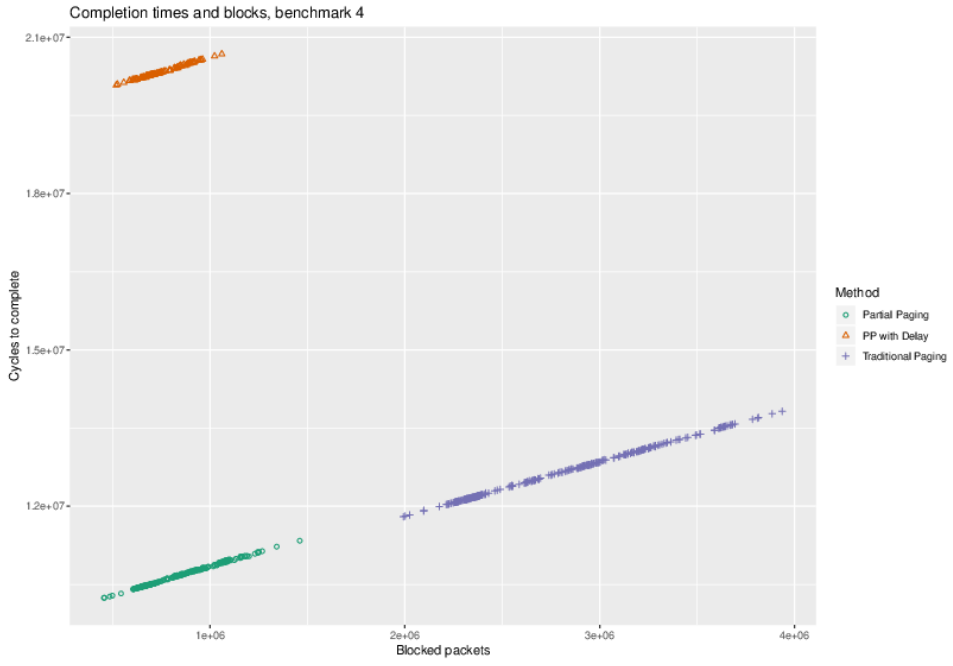

Figure 4.34: Overall cycle count and blocked packets for benchmark 4 with 1KB pages

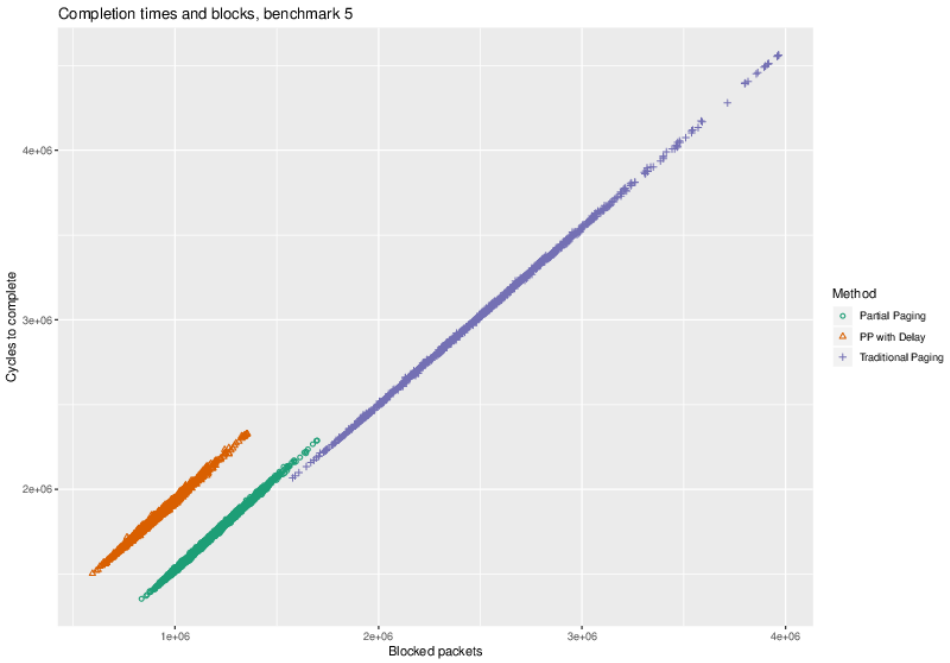

Figure 4.35: Overall cycle count and blocked packets for benchmark 5 with 1KB pages

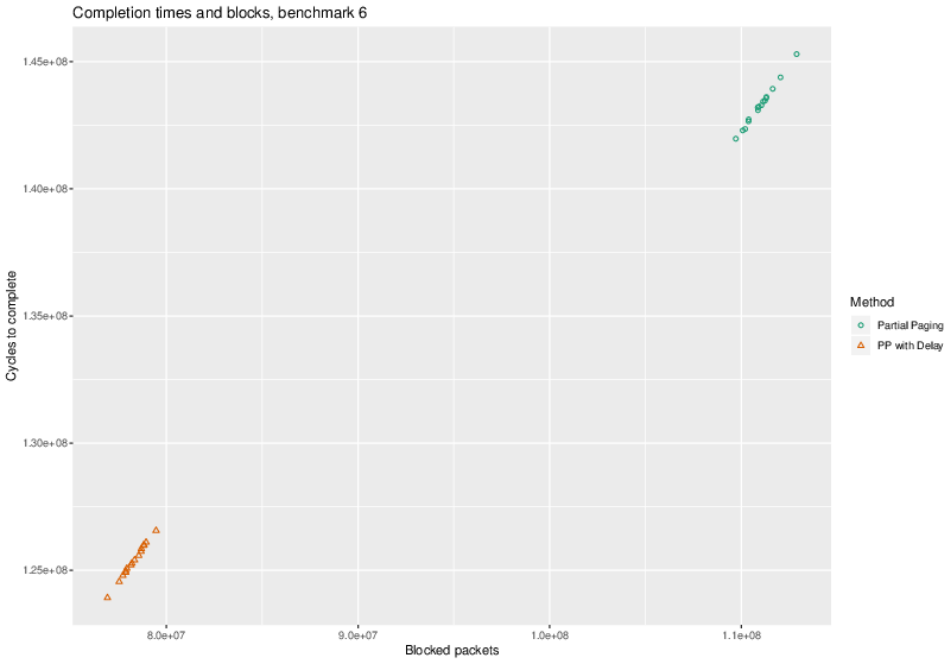

Figure 4.36: Overall cycle count and blocked packets for benchmark 6 with 1KB pages

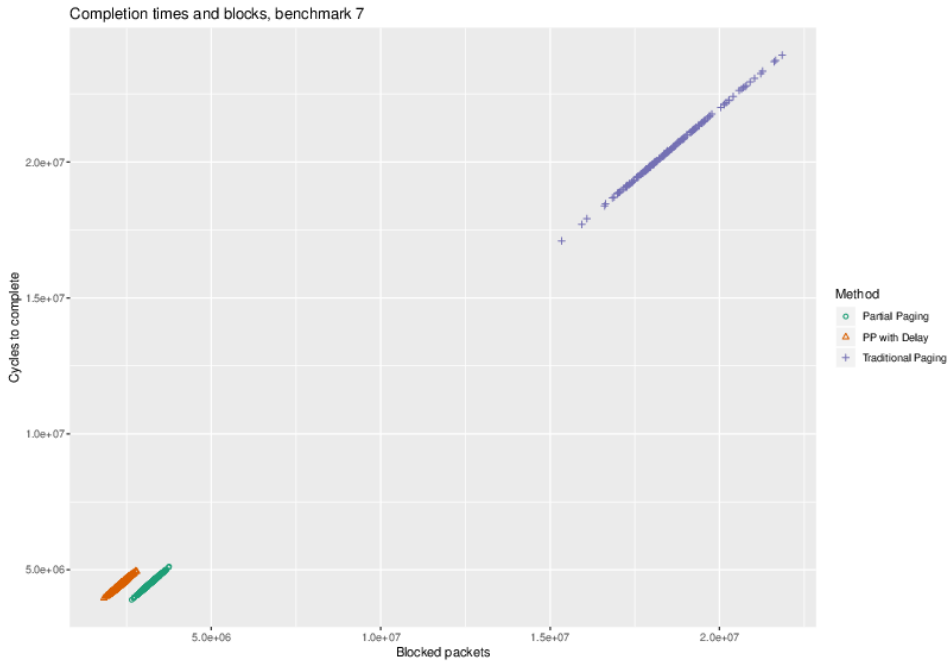

Figure 4.37: Overall cycle count and blocked packets for benchmark 7 with 1KB pages

We next model the factors behind the range of completion times as we consider how to determine or predict WCETs for real-time systems using partial paging.

4.9 PREDICTING WORST CASE EXECUTION TIMES

As noted above (1.2) a *hard* real time system fails completely if timing bounds are exceeded, while in a *firm* real time system the output of any instance that exceeds timing boundaries is useless: assuming that the observed maximum (or some simple multiple thereof or an addition to) of a small set of measurements provides a reliable figure for a safe WCET will certainly be mistaken. Instead we need to consider some form of statistical model for the possible maximum values of the execution times [76].

The systems we are examining here are complex - the timing of any individual process is not independent of others being executed and we are also seeking to estimate something which is - because of its very rarity - almost certainly beyond the range of our observations [56] in the form of an acceptable (e.g., in terms of safety) probabilistic upper value for the execution time.

In this section we consider first how uncertainty in the system state leads to uncertainty in timing outcomes before looking at how we can apply statistical methods from extreme value theory to generate probabilistic worst case execution times.

4.9.1 Modelling system entropy

We cannot be certain of our system's behaviour in terms of timing outcome. Packets compete with one another to make progress through the memory tree and experience different waiting times depending on the state of the system at any given moment. If the system's state was constant in the sense that it was fully ordered then every packet's history would be similar (depending only on a fixed relationship with other packets) and timing outcomes would fall into a very narrow band. The broad range of timing outcomes actually seen thus suggest a disordered system and the greater the disorder, the greater the likely range in outcomes.

We aim to measure this disorder through estimating the system's *entropy*. Entropy in this sense, as applied to an information-based system, is analogous to that found in thermodynamic systems (and the formula shown below as 4.1 can be derived in the same way as the Gibbs entropy formula for thermodynamic systems, though is rebased to use base 2 logarithms and to eliminate Boltzmann's constant [103]). Entropy in an information system is a scalar quantity measured in 'bits' which can be thought of as either the *average uncertainty* about system state or, alternatively, as the *average information* (in the form of answers to yes or no questions) required to describe the system state [27].

Reducing the entropy of our system is thus a way of increasing certainty about its timing outcomes (though not necessarily delivering faster outcomes) and that is the reason we consider its modelling here.

That lower entropy can potentially lead to a significant (non-linear) reduction in the randomness of timing results in our system is also suggested by the *asymptotic equipartition property*: this states that for an ensemble of N independent identically distributed random variables, $X = (X_1, X_2, \ldots, X_N)$ with sufficiently large N then the outcome $x = (x_1, x_2, \ldots, x_N)$ is almost certain to belong to a subset of outcomes having $2^{NH(X)}$ members (where $H(X)$ is the entropy of the system) where each member of the subset has a probability close to $2^{-NH(X)}$[133, 39]. Hence when $H_{observed} \ll H_{max}$the outcome set is a very small fraction of the total number of possible outcomes.

For a (distributed) computer or information system we have for entropy H [137]:

$$H = \sum_{i=1}^{k} p_i \log_2(\frac{1}{p_i}) \tag{4.1}$$

Where the system can be in one of $1..k$ states and the probability of state i is p_i. Entropy is additive[15], so in considering our system we can look at the entropy of one part of the system and add it to the entropy of others to get a system-wide value.

We can identify two major sources of uncertainty (and thus entropy) in our system: the CLOCK page replacement mechanism and blocking/queuing in the memory interconnect. The performance of the CLOCK algorithm will depend on several factors, such as the reference string of the application being run and the amount of blocking in the system. Other researchers have found that the determining the entropy of a caching process (including demand paging) may not allow us to make good predictions as to its performance [124] and we do not consider it further here. Instead we seek to model the impact of queuing/blocking in the memory connect, though we must accept this weakens the general power of our model.

If we consider the memory connection tree we can look at buffers one by one and then add the entropies together to get a figure for the connection tree as a whole. In this sense (and in a way that is analogous to thermodynamic entropy) can consider entropy to be a logarithmic measure of the number of microstates in a system that result in the same macrostate. In our case the states are differentiated by the waiting time for a packet.

In our system we can estimate that around 8.5% of requests are writes and about 91.5% are read requests (from Table 4.3). However, one write marks a whole page as dirty and requires all parts of that page in local memory to be written back when it is paged out and we find - from observation - that around 27% of all requests received by the MMU are write requests. Figure 4.38 shows write requests as a a share of all packets for 512 byte partial paging: it can be seen an initial period when all requests are reads is followed by a surge of write requests before the value moves through a narrower range.

As we are simulating a flash memory device, we assume write requests take twice as long to service as read requests (a basic read request takes 50 cycles and a write thus takes 100 cycles). We can process 4 requests simultaneously and work on the assumption that blocks are caused by queuing to access the MMU alone. Thus blocks occur when all the slots at the MMU are in use. Thus the expected frequencies and maximum service times we can expect are as in Table 4.11[16].

As Figure 4.39 illustrates variation in system performance is closely correlated to variation in the share of packets that are write requests.

15 Strictly this assumes the independence of the entropy of the systems being combined which is not fully the case here as new requests from cores cannot be issued until old ones are dealt with, but we ignore this to present this outline argument.

16 These expectation values are generated from the coefficients of the binomial expansion of $(0.27w + 0.73r)^4$.

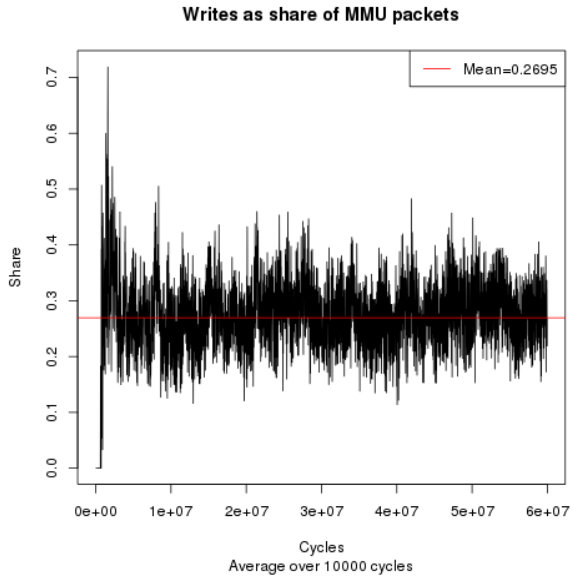

Figure 4.38: Writes as a share of all packets processed by MMU (512 byte partial paging)

Figure 4.39: Mean wait times and write packet share at root mux (512 byte partial paging)

Packet combination (in MMU)	Maximum waiting-time (cycles)	Probability
All read-only	48	0.2840
One write, three read-only	48	0.4201
Two write, two read-only	49	0.2330
Three write, one read-only	49	0.0575
Four write	98	0.0053

Table 4.11: Idealised MMU waiting times and frequencies

The MMU can accept two requests simultaneously, which has the effect of raising the theoretical maximum waiting time while lowering the mean waiting time[17] .

Figure 4.40 shows the observed frequency of inter-arrival times. We find 1.82% of packets have arrived at the root mux simultaneously, 4.39% one cycle after the previous packet and 4.42% two cycles after the previous packet, and so on.

Modelling the system is complex as waiting times are not independently distributed and are dependent on previous packets. The observed waiting times at the root mux are shown in Figure 4.41. This shows a sharp peak at around 50 cycles, reflecting the small average inter-arrival time and the fact that we expect 75% of all packets to have a service time of 50 cycles and that 99.5% of all four packet combinations will generate a maximum waiting time of 49 or 48 cycles.

As we are assuming blocks are caused only by queuing to access the MMU then blocked buffers will be concentrated towards the root of the memory connection tree. Thus, if there are 2 blocks then we assume that both those blocks are in the root mux, Thus we can generate a probability function from Figure 4.41 and so calculate an entropy from this distribution of probabilities of 5.78. As our assumption is that both buffers are blocked then the layer entropy becomes 11.56.

For the next layer down (and for each successive layer) the volume of potential states for an individual packet doubles (and the number of buffers also doubles). How this impacts entropy depends on the spread of probabilities - if the distribution was as before then the entropy of each buffer would increase by 1 (reflecting our use of base 2 logarithms).

However, what we actually see (Figure 4.42) is not just a doubling of the range of timing delays but a slight falttening of the peak probabilities and calculation shows that the per

17 E.g., if four read-only requests were accepted in two sets of two requests immediately succeeding one another then it would be 48, rather than 47, cycles before another request could be accepted.

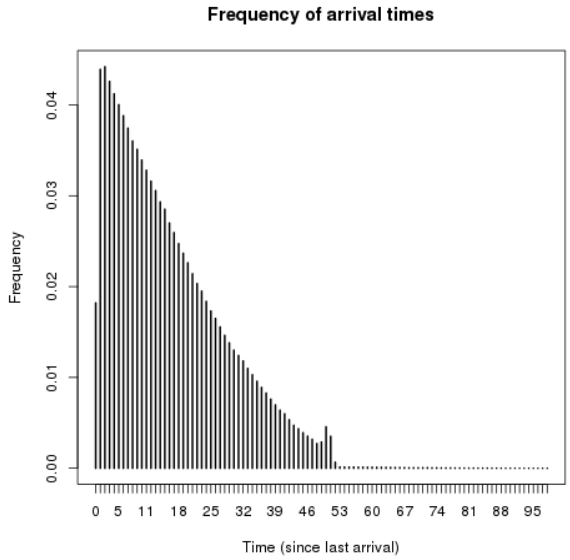

Figure 4.40: Observed frequency of inter-arrival times (with 512 byte partial paging)

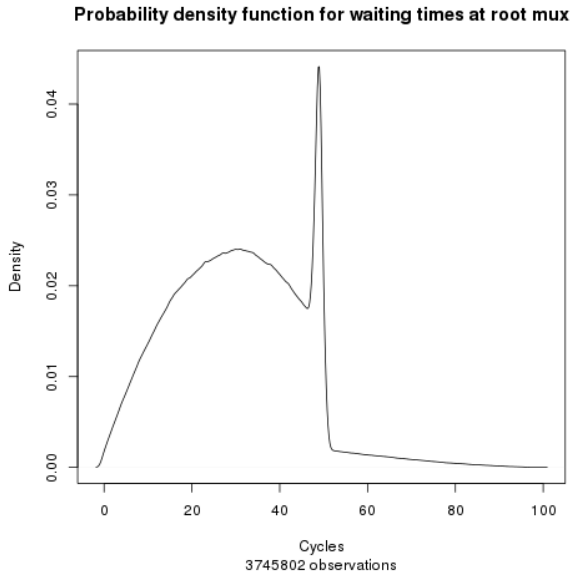

Figure 4.41: Probability density function for waiting times at root mux (for 512 byte partial paging)

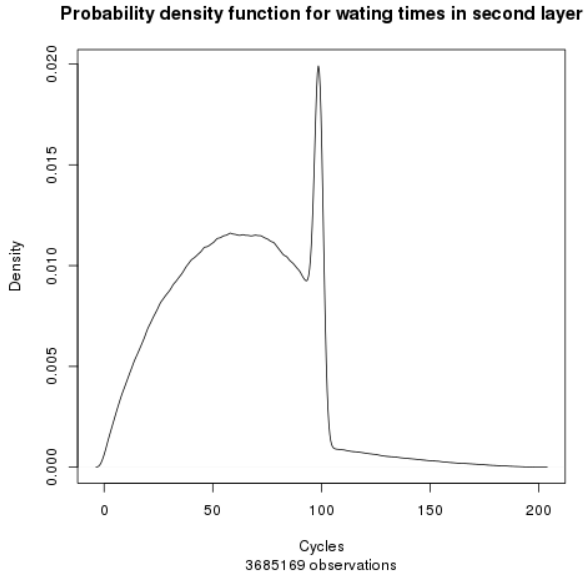

Figure 4.42: Probability density function for waiting times at second layer muxes (512 byte partial paging)

buffer entropy rises to 6.79, slightly more than 1. Assuming that the entropy per buffer rises by 1.01 at each layer we can tabulate projected entropy - Table 4.12.

The entropy projections are graphed in Figure 4.43. The red line here represents the power regression generated by R's linear model of the results[18]. The dotted lines represent the various observed mean blocking counts: blue for traditional with 97.9 blocks (with a projected entropy of 928), while the orange is for 79.6 blocks as seen with partial paging (for a projected entropy of 735 and a ratio between the two of 1.26), while purple is for 58.8 blocks and partial paging with one delay (and an entropy of 521 and a ratio with undelayed partial paging of 1.41).

Entropy here gives us an intuitive insight into the physical processes that ensure additional blocking, by increasing the volume of the number of paths through the memory a packet can 'experience' (or alternatively, increasing the phase space[19]), lead to a greater uncertainty in timing. But we must be wary of assigning it any predictive power (cf. the introductory remarks in [130]), especially as we know that the CLOCK process will inject substantial randomness into timings and our various assumptions underpinning the model have not been fully tested. Certainly Figure 4.44, which compares the range of completion times for the different methods and benchmarks (using a log scale to make the comparison of ratios clearer), suggests no clear relationship between modelled entropy and the ratios of the range of completion times.

18 This is, for blocks B, entropy $H \approx 5.19 \times B^{1.1315}$. The value of *F-statistic* for the fit is high at 5.558×10^4 and p is small at 1.942×10^{-9},suggesting a good fit

19 We take this idea from the discussion of cosmological entropy in [129]

Layer	Total blocked packets	Maximum wait	Buffer entropy	Layer entropy	Total entropy
1	2	98	5.78	11.56	11.6
2	6	196	6.79	27.16	38.8
3	14	392	7.80	62.40	101.1
4	30	784	8.81	140.96	242.1
5	62	1568	9.82	314.24	556.3
6	126	3136	10.83	693.12	1247.4

Table 4.12: Modelled entropy values for blocks in memory tree

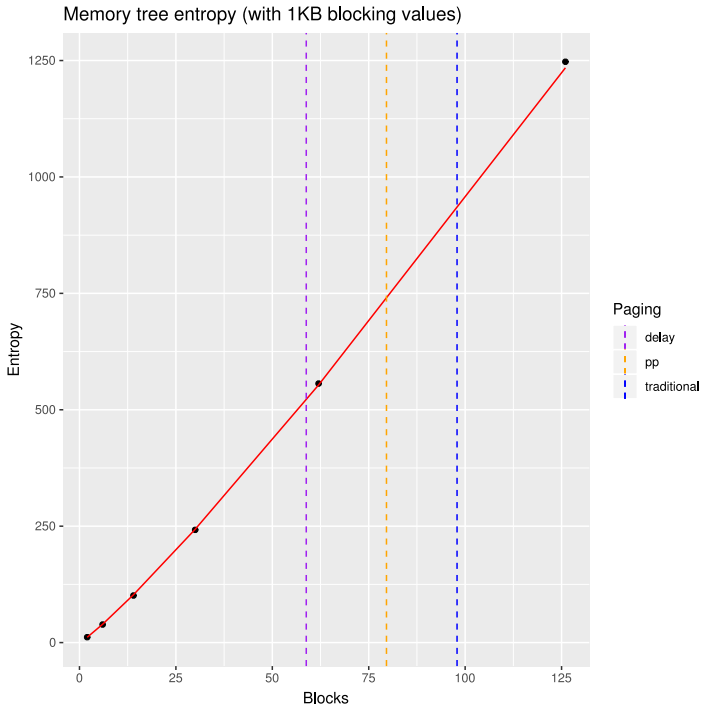

Figure 4.43: Modelling entropy in the memory tree - dotted blue line is mean blocking with traditional paging, orange is partial paging and purple is partial paging with 1 cycle delay

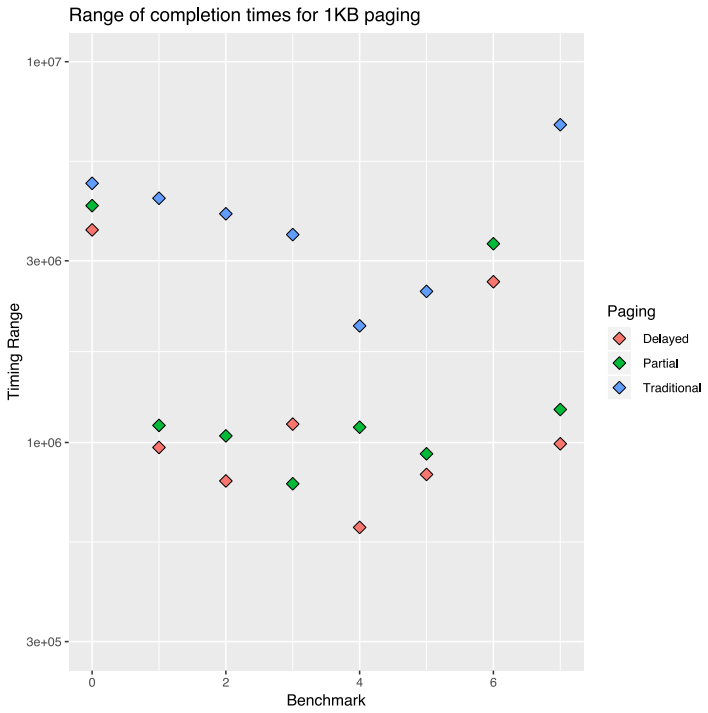

Figure 4.44: Spread of completion times of benchmarks with different paging methods

4.9.1.1 *Limitations of the entropy model*

We suggest three possible reasons for the failure of the model to offer any predictive power:

1. *The effect of entropy is systemic and attempting to measure it through individual benchmarks is flawed*. The benchmarks do not operate independently and a per benchmark measurement (as in Figure 4.44) does not capture the system-wide impact of changed entropy.

2. *Other sources of uncertainty in the system have a strong or even dominant effect* in determining the range of completion times. We have already mentioned the CLOCK page replacement algorithm as a source of uncertainty and the delay process could be another (as it injects proportionally more delay into processes that make less use of the memory interconnect and so will also be dependent on cache state and reference string). These factors combine to produce an underlying distribution of timings: this is almost certainly a factor here (see the discussion that follows in 4.9.2).

3. *The model is flawed in its assumption that blocking occurs only as a result of queuing to access the MMU* (and is thus concentrated towards the root of the tree): if blocks were spread throughout the tree then in general entropy would be lower - as a packet thus blocked might only have to wait 1 additional cycle to clear.

Benchmark	0	1	2	3	4	5	6	7
1KB with partial paging	0.3361	7.532×10^{-5}	1.311×10^{-8}	0.002641	4.758×10^{-13}	3.615×10^{-10}	0.5616	0.6434
1KB with traditional paging	9.422×10^{-6}	0.0001262	1.701×10^{-12}	0.1408	5.736×10^{-10}	8.387×10^{-12}	NA	0.000821
1KB with 1 cycle delay	0.07281	9.031×10^{-8}	1.735×10^{-12}	1.287×10^{-7}	0.001045	6.136×10^{-15}	0.9917	0.7538

Table 4.13: Shapiro-Wilk test for normality p-values for different benchmarks and paging approaches with 1KB pages

Further investigation would appear to allow us to dismiss a fourth reason that might seem feasible: *differing sampling windows*. It is reasonable to believe that, as in a typical thermodynamic system, we should expect the level of disorder to grow over time until an equilibrium is reached. Certainly longer timing paths through the tree have a lower probability and are more likely to be seen if we run the system for longer. The traditional paging system ran for for around 222 million cycles and the partial paging system for 192 million cycles and it is possible that this additional time extended the range of timings seen for the traditional system. However this is easily corrected for by restricting our measurement of the range of traditional paging timings to those benchmarks which completed inside the running length of the partial paging system. This, though, only reduces the range of one of the benchmarks, benchmark 0. It's range now falls to 4169053, a range less than that seen with partial paging (4187159). If instead we restrict the sampling window in such a way that all the benchmarks have the same number of iterations then, in general, the ratios move further away from the ratios of modelled entropy by growing larger (again with the exception of benchmark 0 where the traditional range falls to 2462860.)

4.9.2 Predicting worst case execution times

Figures 4.45 and 4.46 show the distribution of completion times for benchmark 4 and benchmark 5 shown as a quantile-quantile (QQ) plot against an expected normal (Gaussian) distribution of completion times. These results are typical in that the fit, across the range of all the completion times, is generally poor. Applying the Shapiro-Wilk test [147] suggests (Table 4.13) that a Gaussian distribution is not the most likely observation in many (or even most) cases when considering the overall distribution of completion times. In the test the null hypothesis is a Gaussian distribution and for 1KB partial paging (at the 95% confidence level) only benchmarks 0, 6 (with a very small sample) and 7 are passed as Gaussian. For traditional paging only benchmark 3 is passed as Gaussian. While for partial paging with an additional one cycle delay benchmarks 0, 6 (with a small sample) and 7 appear to exhibit normality.

However, we are not concerned here with predicting the spread of all completion times, but only with measuring, or estimating, the worst-case times. We begin by considering whether

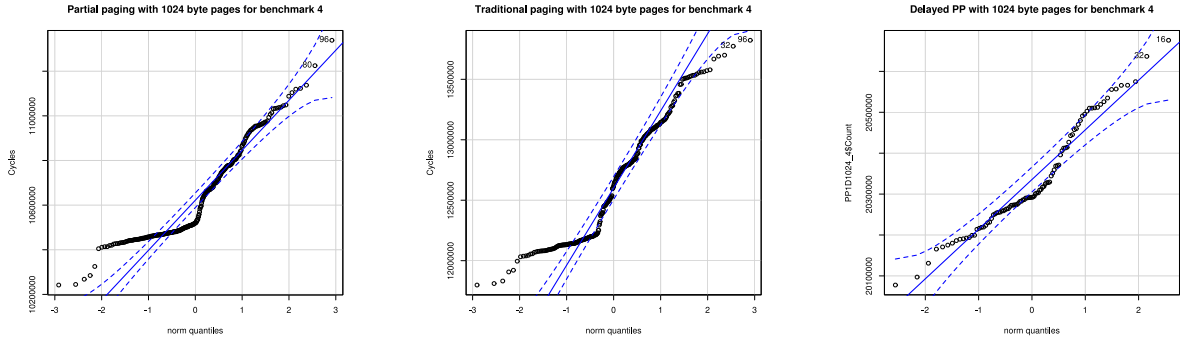

Figure 4.45: Distribution of completion times (against expected Gaussian quantiles) for benchmark 4 with partial paging, traditional paging and partial paging with delay with 1KB pages

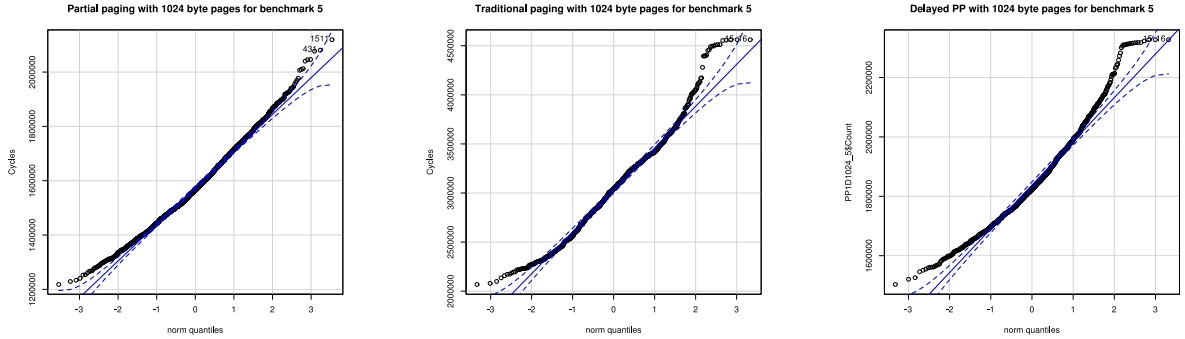

Figure 4.46: Distribution of completion times (against expected Gaussian quantiles) for benchmark 5 with partial paging, traditional paging and partial paging with delay with 1KB pages

we can model the delays in the memory interconnect, as we know from above that the level of delay appears to dominate performance.

The model discussed in 4.9.1, shows we can assume there is a maximum waiting time in the memory tree: at the top layer (for a packet in the non-priority buffer) this is 98 cycles and it will double for each layer further down the tree, so rising to 1568 cycles if layers 1 - 5 were blocked (62 blocked packets). The probability of a packet experiencing this delay at this level of blocking would be extremely small, however: perhaps as small as the order of 10^{-22} (from raising the probability of four writes in the MMU to the fifth power and multiplying this by a 0.01 chance of a zero inter-arrival time raised to the fifth power).

It can be seen that we could assign a correct worst case time for traversal of packets across the memory connect using this data but that such a delay would be unlikely to ever be observed, even on extreme timescales. Thus there is a strong argument to consider a probabilistic worst case execution time (pWCET) in this case.

Furthermore, other sources of delay in the system do not lend themselves as easily to analysis as the memory connect appears to do here, and so we have to look to other means to determine an overall WCET/pWCET. In general the growing complexity of many-core computer systems increasingly makes static analysis impossible in any practical sense [142, 93]. Moreover, a many-core system is unable to deliver the spatial and temporal partitioning that designers of hard real-time systems seek [45]. Statistical methods are thus used to derive a pWCET - and are generally based on extreme value theory (EVT) which was developed from the examination of the extreme tail events of natural phenomena (such as flooding) [148].

Extreme Value Theory is a non-parametric statistical method and so does not require a particular underlying distribution of data [31] but a practical application generally uses one of the three specific distributions for extreme values (Gumbel, Fréchet and Weibull) rather than the generalised extreme value (GEV) distribution [56].

The three limiting distributions are a particularisation of the GEV tail as signified by the shape (ξ) parameter. A Weibull distribution (negative ξ) has an absolute cut off maximum and is most useful in considering minima [56] (though could also be applied to the distribution of completion times when an absolute WCET is known [113]) and we do not consider it further here. A Gumbel distribution (zero ξ) has an exponential tail and a Fréchet distrubution (positive ξ) has a heavy tail. Figure 4.47 illustrates the cumulative density function for a Gumbel distribution and Fréchet distribution with similar parameters, illustrating the heavy-tail effect with a Fréchet distribution.

We concentrate on the Gumbel distribution below and it, as a cumulative distribution function, is of the form:

$$F(x;\mu,\beta) = e^{-e^{-(x-\mu)/\beta}}$$

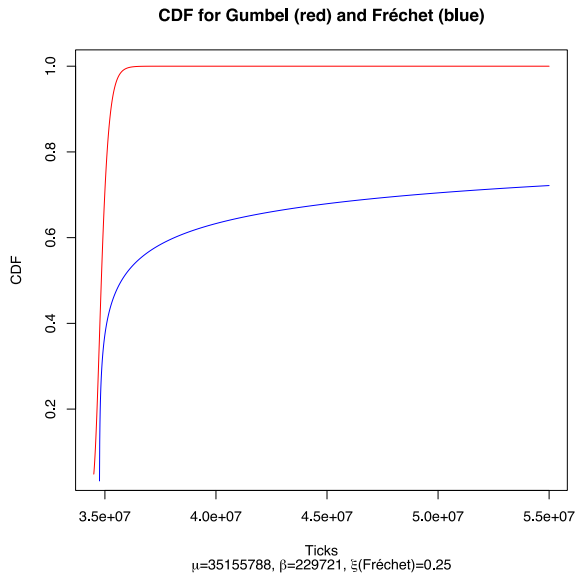

Figure 4.47: Cumulative distribution functions for Gumbel and Fréchet distributions with the same (arbitrary) values for location (μ) and scale (β) parameters

Where μ is the mode of the distribution and the mean is $\mu + \gamma\beta$, where γ is Euler's constant.

The safe application of EVT to our system is open to question. The methods of EVT assume that completion times are independently and identically distributed (i.d.d.) and that cannot be the case in a system where multiple benchmarks compete for memory and where performance depends on past reference strings.

In our system, after the initial iteration of the program subsequent behaviour and performance will depend upon which pages and parts of pages are already in memory and, of course even for the initial distributions we are dependent on the behaviour of other programs in the system (including other instances of the same benchmark). Basing a safety-critical decision on statistical extreme value theory (EVT) when this i.d.d. property is not present may well be unsafe [74]. However the i.d.d. criterion may be relaxed if the distribution is stationary [148] and a GEV based distribution may provide a good fit in such cases [139].

Table 4.14 shows the results of Augmented Dickey-Fuller tests for stationarity [1]. Here the null hypothesis is non-stationarity and it can be seen the alternative hypothesis of stationarity is met for partial paging (at statistically significant levels), with and without delay, for benchmarks 1, 2, 3, 4, 5 and 7 (the test fails on 6 because we only have one set of 16 completion times). Benchmark 0 fails in both partial paging scenarios. For traditional paging benchmark 7 fails and we have no results at all for benchmark 6.

We thus propose here to use EVT to model completion times and to derive probabilistic worst case execution times (pWCETs).

Benchmark	Partial paging: p-value	Traditional paging: p-value	Partial paging with delay: p-value
0	0.1745	<0.01	0.574
1	<0.01	<0.01	<0.01
2	<0.01	<0.01	<0.01
3	<0.01	<0.01	<0.01
4	0.03137	<0.01	0.0415
5	<0.01	<0.01	<0.01
6	NA	No results	NA
7	<0.01	0.15	0.04885

Table 4.14: Augmented Dickey-Fuller test results for stationarity for benchmarks with 1KB page sizes: with lag of 16

(The question arises of how to determine a distribution of maximum completion times for those benchmarks that do not show stationarity. Conceivably we could apply standard methods to correct for any lack of stationarity in results or perhaps collect a much larger sample size which may show stationary results over a longer range. We did not explore either option here, but these results do suggest that for some class of programs in such many-core systems where there may be no safe limit determinable by statistical methods.)

4.9.2.1 *Partial paging and traditional paging pWCETs*

We first assess our data for goodness of fit with both Gumbel and Fréchet distributions (cf. [55]): using `ev.test` from R's `goft` package [71] we tested for the null hypothesises of both a Gumbel and a Fréchet distribution. We used an algorithmic block-maxima based approach (cf.[76]): we look at the maxima inside a block of fixed size (starting with 16, but moving on to 24, 32, 48, 56 and 64) testing for a fit. If the fit test fails with the smaller block size we move on a larger block size.

Table 4.15 shows the result of the goodness of fit (GOFT) tests for partial paging (without delay). Benchmarks 0 is excluded as it failed the stationarity test and 6 has insufficient data. Benchmark 4 does not generate a large enough range of samples (17 at a block size of 16) to apply the test. Benchmark 2 does not pass the statistical test (i.e. $p < 0.05$) for either distribution and the figures here are for illustrative purposes only. For benchmark 3 a block size of 24 appears to pass for both distributions but the sample size we are left with - 23 -

Benchmark	p-value for Gumbel	Block size used	Sample size used	p-value for Fréchet	Block size used	Sample size used
1	0.05758	16	159	0.05514	24	106
2	0.03505	64	41	0.02377	64	41
3	0.9019	24	23	0.8839	24	23
5	0.2079	40	44	0.1374	40	44
7	0.3868	16	41	0.3289	16	41

Table 4.15: Goodness of fit tests for Gumbel and Fréchet distributions for partial paging with 1KB page size

is below the 30 "generally accepted" as the minimum for reliable results [76], so the results included below need also to be treated with additional caution.

In general the results - including the positive GOFT results for Fréchet distributions - suggest we need many more results for truly reliable predictions. In [113] the authors strongly discourage the use of Fréchet to model pWCETs, stating: "The Fréchet distribution is most appropriate when a maximum value does not exist, which places it outside of the core WCET problem domain." Further they state that a Fréchet distribution can be regarded as an upper bound on a more appropriate Gumbel distribution and add that "If a Fréchet distribution were to best fit a given sample of execution-time measurements, this should happen because either the sample does not contain enough tail values or some of them do not really belong to the tail of the distribution."

In our case Fréchet generally delivers a poorer fit than Gumbel with a lower p–value for a given block size but its general ability to fit at the same block size as Gumbel is probably an indicator that more robust results here would require a greater number of samples than we were able to collect in the available time. Below we concentrate on modelling Gumbel distributions of pWCETs.

Table 4.16 shows GOFT data for traditional paging. Here benchmarks 4, 6 and 7 are excluded for lack of sufficient data.

Although benchmark 5 under traditional paging appears to pass the goodness of fit test here, we exclude it from further testing because, as Figure 4.46 shows, completion times are bimodally distributed, with the completion times of the initial runs of the benchmark taking all the extreme times. To calculate a pWCET in these circumstances it is necessary to collect data from the first initial runs only, a method we apply when considering pWCETs for smaller page sizes (Sections 5.2.2 and 5.3.2).

Benchmark	p-value for Gumbel	Block size used	Sample size used	p-value for Fréchet	Block size used	Sample size used
0	0.386	16	36	0.2485	16	36
1	0.2787	16	70	0.05653	16	70
2	0.1843	16	73	0.348	24	48
3	0.9564	16	23	0.939	16	23
5	0.3552	16	72	0.2244	16	72

Table 4.16: Goodness of fit tests for Gumbel and Fréchet distributions for traditional paging with 1KB page size

Benchmark	1	2	3	5	7
Observed maximum	1798061	1739293	5779890	2288311	5120014
Computed μ (location parameter)	1542286†	1517751‡	5586817	2065689	4886545§
Standard error for computed μ	5932	NA	8389	5932	11863
Computed β (scale parameter)	72172	77520	72272	73151	67308
Standard error for computed β	4194	NA	8389	4194	8389
Computed percentage for observed maximum	$\approx$ 97.2%	$\approx$ 94.4%	$\approx$ 93.3%	$\approx$ 95.3%	$\approx$ 96.9%
Computed threshold for 10^{-1} maxima (i.e. 0.9 CDF)	1704700	1692199	5749456	2230306	5038013
ditto 10^{-3}	2040796	2053201	6086018	2570962	5351459
ditto 10^{-6}	2539379	2588729	6585292	3076307	5816439
ditto 10^{-9}	3037926	3124219	7084529	3581617	6281387
ditto 10^{-12}	3536474	3659709	7583768	4086927	6746335
ditto 10^{-15}	4035076	4195259	8083061	4592294	7211335
Memo: observed worst traditional paging time	5697371	5463362	11269496	4561929	23926071

†The block maxima for a block size of 32 was used to calculate this distribution as the smaller block size would not generate error values. The smaller size gave a μ of 1487978 and a β of 77104 which would place the observed maximum at $\approx$ 98.2% but would also generate a higher 10^{-15} threshold time of 4152117.
‡Maximising goodness-of-fit and not maximum likelihood fitting used and no error values available.
§The block maxima for a block size of 32 was used to calculate this distribution as the smaller block size would not generate error values. The smaller size give a μ of 4791571 and β of 95582, placing the observed maximum at $\approx$ 96.8% and a 10^{-15}threshold time of 8092898.

Table 4.17: Modelled Gumbel distributions of benchmark maxima completion times for 1KB partial paging

Benchmark	0	1	2	3
Computed μ	7100487	4243457	4052264	10366155
Computed β	496696	447542	356165	254569
Observed maximum	9012279	5697371	5463362	11269496
Threshold for observed maximum	$\approx 97.9\%$	$\approx 96.2\%$	$\approx 98.1\%$	$\approx 97.2\%$
Computed threshold for 10^{-15} maxima	24256156	19701333	16354052	19158860

Table 4.18: Calculated parameters for Gumbel distributions for timing extremes for traditional paging with 1KB pages

Table 4.17 shows the outcome of the modelling process for the benchmarks with partial paging. Here μ is the *location* parameter and is the mode of the distribution, while β is the *scale* parameter, with the mean of the distribution being equal to $\mu + \beta\gamma$, where γ is the Euler-Mascheroni constant. Typically a safety-critical application will demand an hourly failure rate of less than 10^{-9} which we would expect to be typically be around the 10^{-12} - 10^{-15} limit reported in 4.17. As can be seen even the most of extreme of these measures appears to be less than the observed worst traditional paging time in every case except benchmark 5 where the observed maximum with traditional paging is similar to the 10^{-15}calculated threshold time for partial paging.

In Appendix F we chart the fit of the modelled Gumbel distribution to each benchmark.

Table 4.18 shows calculated parameters for Gumbel distributions for traditional paging[20]: the significantly higher values of β, the scale parameter, compared to partial paging, show that we should expect pWCETs to stretch to very high values indeed. For benchmark 1, for instance, the calculated μ, i.e. modal values, for maxima are in a ratio of 1:2.75 between undelayed partial paging and traditional paging, but the 10^{-15} threshold times are in a ratio of 1:4.88.

4.9.2.2 *Delayed and undelayed partial paging pWCETs*

The discussion in 4.9.1 set out a basis - increased entropy - for why blocking in the system should lead to the broader spread of completion times for the density plots of traditional paging completion times in Figures E.1 - E.8.

Given that the injecting the 1 cycle delay into bitmap reading discussed in 4.7 reduces the average blocking in the system from ≈ 80 to ≈ 59 blocks per cycle and so should significantly reduce mean entropy in the memory connect, the question arises of whether this approach, despite its significant addition to the average case execution time (ACET) will lead to a lower

20 Standard errors were not available as 'maximum goodness of fit estimation' was used

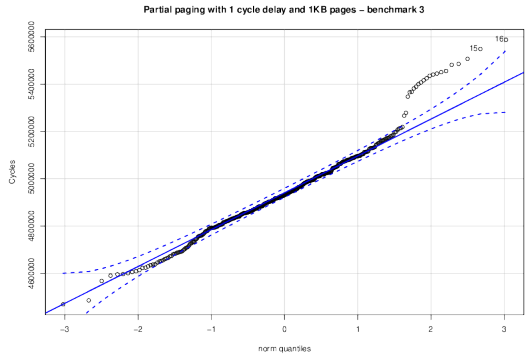

Figure 4.48: Bi-modal distribution of completion timings for benchmark 3 with delayed partial paging

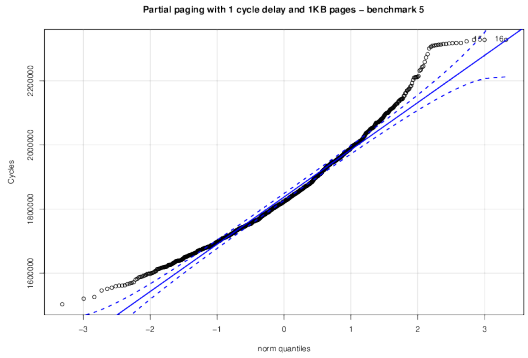

Figure 4.49: Bi-modal distribution of completion timings for benchmark 5 with delayed partial paging

Benchmark	p-value for Gumbel	Block size used	Sample size used	p-value for Fréchet	Block size used	Sample size used
1	0.2583	32	43	0.1302	32	43
2	0.1568	48	30	0.1173	48	30
7	0.0966	16	28	0.07753	16	28

Table 4.19: Goodness of fit tests for Gumbel and Fréchet distributions for partial paging with bitmap delay with 1KB page size

pWCET at low probability (and safety-critical) limits. Unfortunately we have a limited number of results on which we can test this as the timing results lack stationarity for benchmark 0 and we do not have a sufficient number of results benchmarks 4 and 6. Additionally we can also see that benchmarks 3 and 5 are bimodal under delayed partial paging (Figures 4.48 and 4.49).

Table 4.19 shows the results of the GOFT for partial paging with additional delay while Table 4.20 shows the calculated values for μ and β for partial paging with delay (cf. 4.17).

Figures 4.50 - 4.52 show plots of the probability density function (PDF) for benchmarks 1, 2 and 7 under both partial paging and partial paging with a bitmap reading delay. There is no simple pattern here - in benchmark 2 the delayed approach converges with the non-delay approach, in 1 and 7 it diverges. This shows that system-wide entropy caused by blocking cannot be the only factor that determines β and hence the range (disorder) in completion times.The plots for benchmarks 2 and 7 show that differences in typical (mean) performance may be a poor guide to which approach will perform better at safety-critical margins.

In summary, on this (limited) evidence at least, while the delay does lower blocking and so should lower entropy in the memory connect, there is no simple translation into lower

Benchmark	μ(1 cycle delay)	μ standard error	β (1 cycle delay)	β standard error	10^{-15} threshold for delay
1	1743365	5932	91770	5932	4913062
2	1701717	5932	57718	5932	3695272
7	4671040	NA†	99004	NA	8090596

†Maximal goodness-of-fit estimation(MGE) only and no error values available.

Table 4.20: μ and β for 1-cycle delay partial paging for 1KB paging

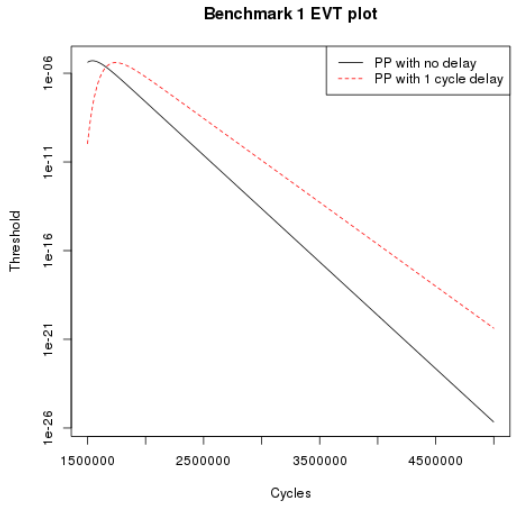

Figure 4.50: Calculated PDFs for benchmark 1 with 1KB partial paging compared

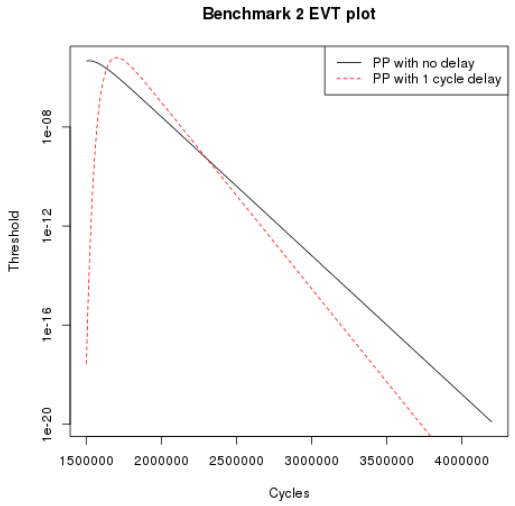

Figure 4.51: Calculated PDFs for benchmark 2 with 1KB partial paging compared

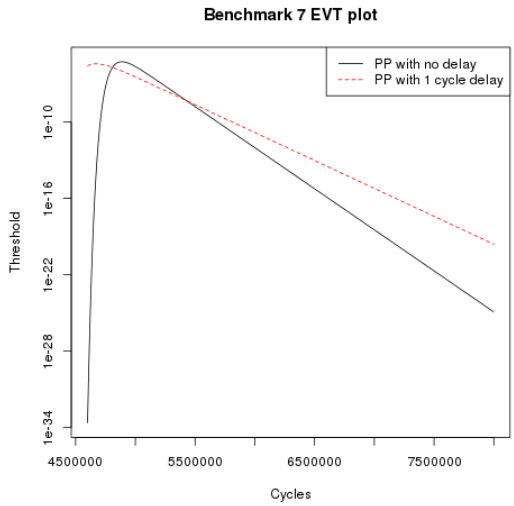

Figure 4.52: Calculated PDFs for benchmark 7 with 1KB partial paging compared

pWCETs, even at very high thresholds. Many other factors beyond entropy in the memory connect, of course, could be contributing to this picture. The marginal efficiency of the CLOCK procedure may well be important here: as remarked above the introduction of a delay comes close to halving the 'sweep' time of the CLOCK procedure and higher predicted WCETs may reflect an expected poor CLOCK performance under this constraint, for instance.

4.10 SUMMARY

The results presented in this chapter show that a partial paging system offers potential as a means of supporting virtual memory on a many-core embedded system, delivering, at least in some cases, better average and predicted WCET performance compared to traditional paging. However, it should be noted that queues were long and blocked packet counts were high: which, when we consider 3.2 and 4.9 points to a high degree of thrashing, a high level of entropy (and thus uncertainty) and a low level of overall efficiency in the system. There is considerable scope for improvement and we discuss potential ways to do this below.

In 4.2.5 we demonstrated that an enforced fairness in the memory interconnect was essential for a many-core system to avoid excessive delays.

In 4.3 we describe the various benchmarks we selected to test the simulated partial paging system we described in 4.2.2 and 4.2.6.

In fact our results suggest that, in the general case, partial paging paging delivers better results than a traditional paging system when considering the average case (4.4) and that this advantage is even more pronounced when considering the worst case execution times (4.8). Applying techniques from extreme value theory reenforces the evidence that partial paging has potential to increase system efficiency.

One of the benchmarks (0) floods the system with memory requests - we describe this as pathological behaviour - due to rapid variations in working set size around the critical value of available pages in our partial-paging system. This slows its own performance and that of all other processors - as we are able to show that the overall time taken to complete an iteration of the benchmark is a linear function of the number of blocked memory requests that a benchmark suffers.

We also show that even a simple load control mechanism can deliver better results when we consider the benchmarks as a set, though, in 4.9.2.2, we note that a better average performance does not necessarily translate into a better performance at the extreme tail (or vice versa).

In the following chapter we examine potential solutions to these problems and consider whether we can find a more robust and better performing partial paging system.

5 OPTIMISING A PARTIAL PAGING SYSTEM

Having demonstrated that a partial paging system can deliver better common-case timings and better WCET results for most of our benchmarks, we now consider ways to optimise performance and to ensure this approach delivers more efficient performance for all likely cases.

We consider two broad types of optimisation: changing the page size (to use smaller pages), and a simpler, in principle less uncertain, page replacement policy in the form of FIFO.

We use smaller pages both because our own observations (3.3) and earlier experimental work we have previously quoted ([78]) show that smaller page sizes in memory-constrained environments are likely to give better (faster) results in the general case. Here, though, we also wish to consider whether small pages also contribute to better WCETs.

We look at FIFO because our earlier results show that a substantial share of the execution time for each process (typically around 10% - see Table 4.7) is absorbed by what we describe as 'administrative' tasks which includes managing the CLOCK process. We also note that CLOCK involves a substantial degree of uncertainty (in that static analysis is practically impossible) and so replacing it with a simpler, more deterministic, page replacement process might bring other benefits.

In 5.1 we consider the difference in performance if we use smaller, 512 byte pages, which, in general, generate a higher hard fault rate and more page table lookups but, by eliminating fragmentation can generate better performance, with a lower blocking count. In 5.2 we consider worst-case timing: smaller page sizes appear to significantly improve the efficiencies of subsequent runs of each benchmark and a distinct bi-modal distribution of completion times is seen (with the initial iterations notably slower) and in 5.2.2 we apply extreme value theory statistical methods to an extended data set of the initial completion times of some of the benchmarks. A comparison of these results to those for 1KB partial paging, where available, shows that 512-byte partial paging retains its advantages even at extreme values.

In 5.3 we consider 256-byte pages and 128-byte pages. EVT-based analysis of the 256-byte pages in 5.3.2 suggests that this smaller page size may have better performance at safety critical thresholds, even where typical performance is worse: we reason this is because of lower entropy in the memory connect. In 5.3.4 we show that, in most cases, the higher fault rate for 128-byte pages leads to significantly lower performance.

We consider FIFO as a simpler page replacement policy in 5.4. Although this often gives better completion times on the first iteration, on subsequent runs performance can degrade and in 5.4.1 we show that this appears to be a result of entropy being injected into the system by memory delays - so that the uncertainty in FIFO timings becomes broadly similar to that seen with CLOCK: potentially nullifying the advantage, for real-time programmers, of having predictable patterns of page loading and replacement. EVT-based analysis of FIFO in 5.4.2 suggests that its performance is generally inferior to CLOCK at safety-critical thresholds.

In 5.5 we consider alternatives to the tree-based memory connect in the form of a crossbar and interconnected buses. In both cases we see that performance is broadly similar to that seen with Bluetree but that different arrangements of backoff timings and buffering can alter system entropy and range of completion times.

Section 5.6 is a short summary of this chapter.

5.1 SMALLER PAGES: 512 BYTE PAGES

Decreasing page size will increase the absolute number of page faults - at least on the first run of any of the benchmarks - but will also reduce external fragmentation and allow for a more efficient use of limited memory resources.

Reducing the page size to 512 bytes divides the 16KB of available per core local memory in our simulation into 32 pages, of which 2 are notionally assigned to local kernel primitives and systems software and 2 to the system stack, 2 for the local page tables but only 1 is required for the bitmaps - so making an extra 512 bytes of local memory free for general use in comparison to the 1KB system, with a total of 25 free pages. A traditional paging system will have 26 free pages.

Table 5.1 (cf., Table4.2) shows how many pages each benchmark accesses when using 512 byte pages and indicates that benchmark 1 should, after faults on loading, be able to run under partial paging without further faults on subsequent iterations. For a traditional paging system this will be true of both benchmark 1 and benchmark 2.

The change in page sizes leaves the working set of the benchmarks in a similar pattern as before. Figure 5.2 shows that, for benchmark 0, the rapid variation in size remains, but that all those variations are below the 25 available page count (cf. Figure 4.7), while for benchmark 6, Figure 5.3 shows that the working set size remains substantially greater than the number of available pages (cf. Figure 4.13).

Figures 5.4 and 5.5 show the lifetime curves [47] for benchmark 0 running under a least recently used (LRU) page replacement policy - of which CLOCK is a close analogue - when

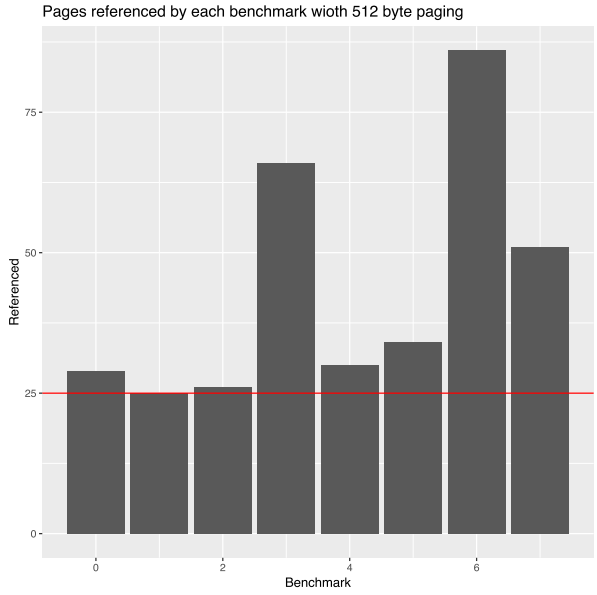

Figure 5.1: 512 byte pages referenced by each benchmark

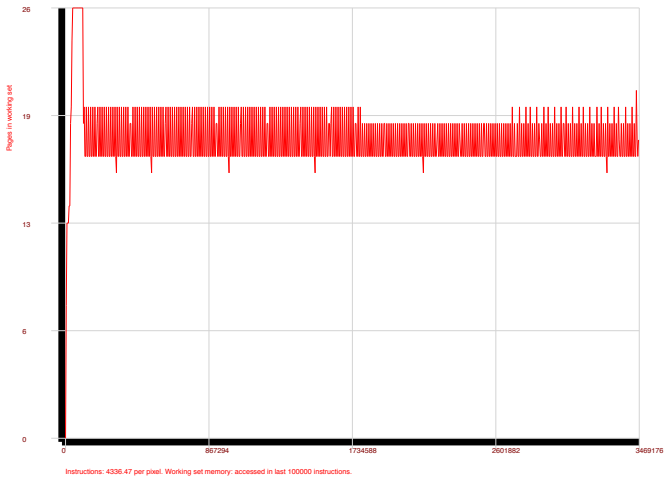

Figure 5.2: Working set size (for 100,000 instruction window) for benchmark 0 with 512 byte pages

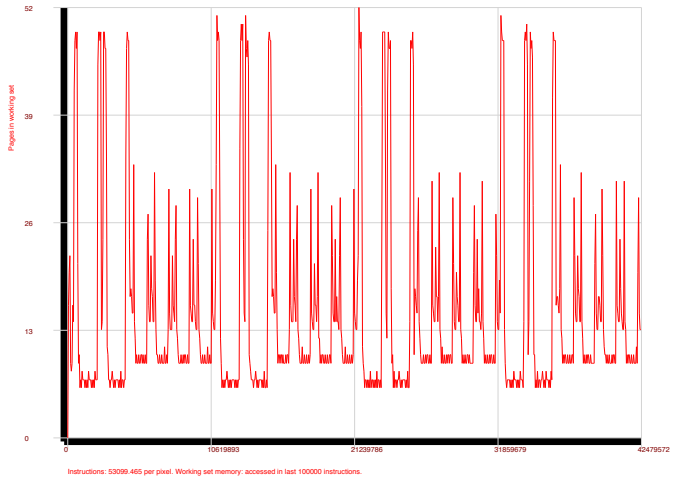

Figure 5.3: Working set size (for 100,000 instruction window) for benchmark 6 with 512 byte pages

Figure 5.4: Lifetime curves (under LRU) for benchmark 0 with 512 byte pages

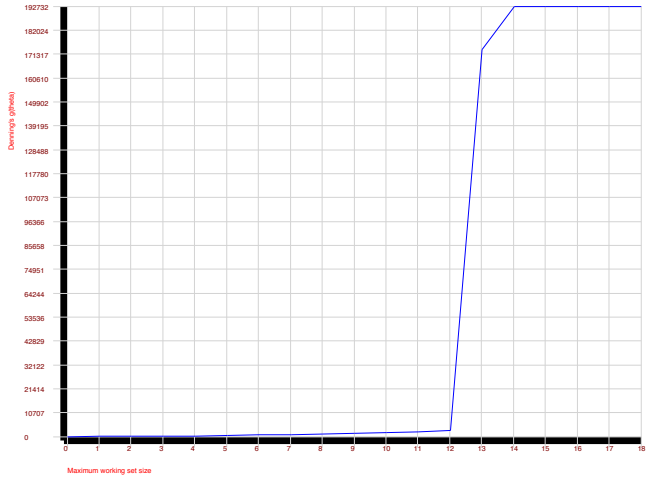

Figure 5.5: Lifetime curves (under LRU) for benchmark 0 with 1KB byte pages

using 512 byte pages and 1KB pages respectively: the y-axis plots the mean number of instructions (as measured in bytes) between hard faults, while the x-axis plots the maximum working set size. For the 1KB page sizes (where the number of pages free for general use in the partial paging system is 12), the number of instructions between faults remains very low until the working set rises to 13 pages: it is this pattern that explains the exceptionally poor performance of the partial paging system compared to the traditional paging system (where 13 pages are free). For the 512 byte page system the threshold is reached at 20 pages and so performance for the partial paging system shows a dramatic improvement as the system now has 25 free pages.

In fact we observe the average number of hard faults for benchmark 0 under partial paging falls from 908.1 with 1KB pages to 33.4 with 512 byte pages.

Using 512 byte pages the number of blocked packets falls significantly, especially after the initial run of each benchmark is completed (after which, for instance, benchmark 1 makes no further global memory requests) - with a mean number of blocks at 45.2 over the first 2.14×10^8 cycles[1]. Figure 5.6 shows a highly smoothed chart of blocking in this system and it is apparent that there is a regular underlying pattern of oscillation in a narrow band after an initial period of higher average blocking.

Tables 5.1 and 5.2 give details for mean performance values for partial paging (assuming sub-cycle bitmap reading) and Table 5.2 suggests that queuing in the memory tree is significantly

1 9,670,040,286 blocked packets over 213,901,387 cycles

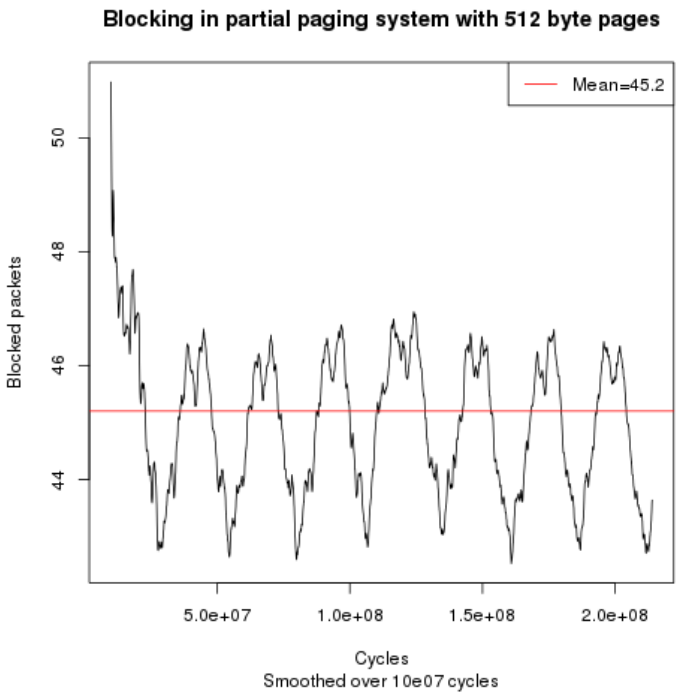

Figure 5.6: Blocking in 512 byte paged partial paging system (smoothed)

reduced (with the exception of benchmark 6) once the initial run of each benchmark is completed.

Figure 5.7 (cf. Figure 4.20) points to a lower impact of each block even if the number of faults is broadly similar to the 1KB example (as partial paging does not change the number of 16 byte lines that need to be loaded). However we can also see that, in general, the range of blocks per fault remains large - pointing to timing uncertainty. We also note that the first iterations (generally a displaced group of points to the right) show higher fault counts: this phenomenon was not so prominent for 1KB paging but is another indication that the mean performance is no guide to the WCET.

Figure 5.8 shows the bi-modal nature of the distribution of completion times with partial paging 512 byte pages for all benchmarks except benchmark 0 (here plotted against an ideal Gaussian distribution of completion times).

The linear regression plots in Figure 5.9 again generally show a linear relationship between completion times and blocking, indicating that it is the high levels of blocking on the initial run that fundamentally produces the bi-modality. For benchmarks 2, 3 and 5 the slight displacement of these initial points from the regression line is a signs that other factors may also to be important.

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	2718724	1345670	1349593	4198788	10961440	1712878	107296451	3845509
Mean	2461912	1323643	1305885	4141580	10895508	1684291	105938397	3694988
Service cost	46184	30550	30406	233875	42222	73100	6152684	154572
Blocks	760747	771355	752933	2878150	965834	1112167	73405338	2398505
Count								
Hard Faults	33.4	25.0	26.0	66.0	32.6	34.0	5172.4	126.0
Small Faults (and page table reads)	792.1	586.0	580.8	1995.5	786.2	852.0	85911.4	2089.1
Share								
Service	1.9%	2.3%	2.3%	5.6%	0.4%	4.3%	5.8%	4.2%
Blocks	30.9%	58.3%	57.7%	69.5%	8.9%	66.0%	69.3%	64.9%
Admin	16.0%	10.9%	11.0%	9.0%	14.6%	9.9%	12.2%	11.0%
Efficiency of execution	51.3%	28.5%	29.0%	15.9%	76.1%	19.7%	12.7%	19.9%

Table 5.1: Performance (means) of benchmarks with partial paging and 512 byte pages: initial execution

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	2871432	442278	1022103	3757085	10357560	1324015	107017746	3780823
Mean	2080572	441565	509549	3376961	10149337	1031102	105720853	3249095
Service cost	32279	0	3316	226620	19073	51514	6162103	158779
Blocks	448495	0	55153	2161594	326416	537015	73183478	1954439
Count								
Hard Faults	17.9	0	3.1	45.3	10.2	11.7	5173.9	118.2
Small Faults (and page table reads)	439.8	0	58.1	1708.9	282.9	396.1	85918.3	2071.8
Share								
Service	1.6%	0%	0.7%	6.7%	0.2%	5.0%	5.8%	4.9%
Blocks	21.6%	0%	10.8%	64.0%	3.2%	52.1%	69.2%	60.2%
Admin	16.2%	14.6%	14.3%	9.8%	14.9%	10.7%	12.2%	12.3%
Efficiency of execution	60.7%	85.4%	74.2%	19.5%	81.7%	32.2%	12.7%	22.7%

Table 5.2: Performance (means) of benchmarks with partial paging and 512 byte pages: continued execution

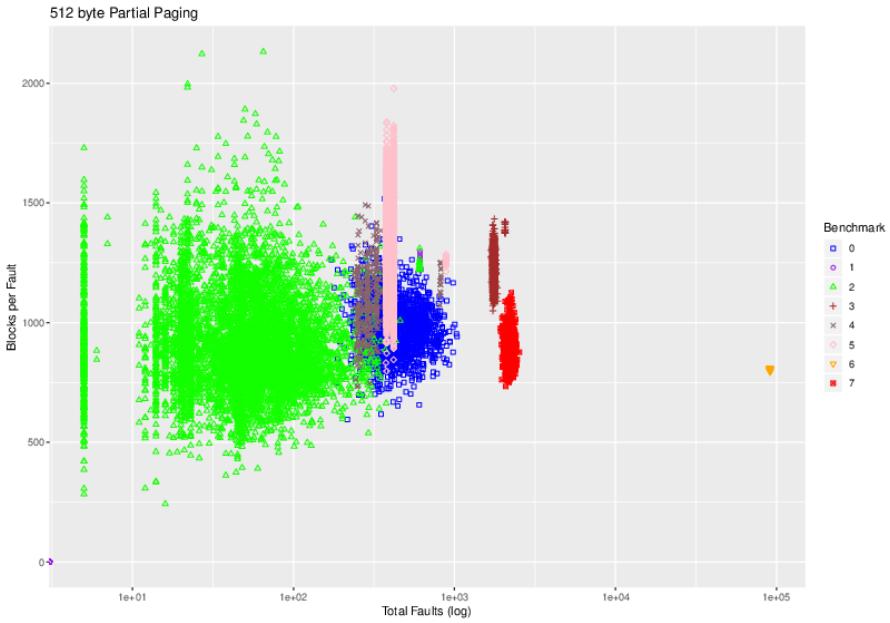

Figure 5.7: Blocks per fault and total faults compared for 512 byte partial paging

Table G.1 in Appendix G shows the calculated linear regression factors for all the benchmarks.

One important factor here may simply be that the huge improvement in benchmark 0's performance (see Figure 5.10). In general the smaller page sizes increase the number of hard faults and remote page table reads required. It should also be noted, however, that in subsequent runs 512 byte paging typically generates a smaller number of faults for all benchmarks because of the reduction in fragmentation (most obviously in benchmark 1 which produces no further faults at all). Benchmark 7 also has a lower total fault count, almost certainly because it writes fewer pages back when pages sizes are small: a further advantage of reduced fragmentation.

As the number of 16 byte 'lines' that need to be loaded in both cases is not changed by the switch in page size, the increase in fault counts shown in the table generally reflect the higher number of hard faults caused by small page sizes increasing the number of remote page table reads.

The improved performance from smaller page sizes is thus an *emergent* characteristic of the system: with the exception of benchmark 0 all the benchmarks make higher demands on the system but all show improved performance because of the lower amount of blocking in the system. After benchmarks 2 and 1 complete their first iterations (respectively the first and second benchmarks to do so, they also show a large fall in fault rates, further improving the

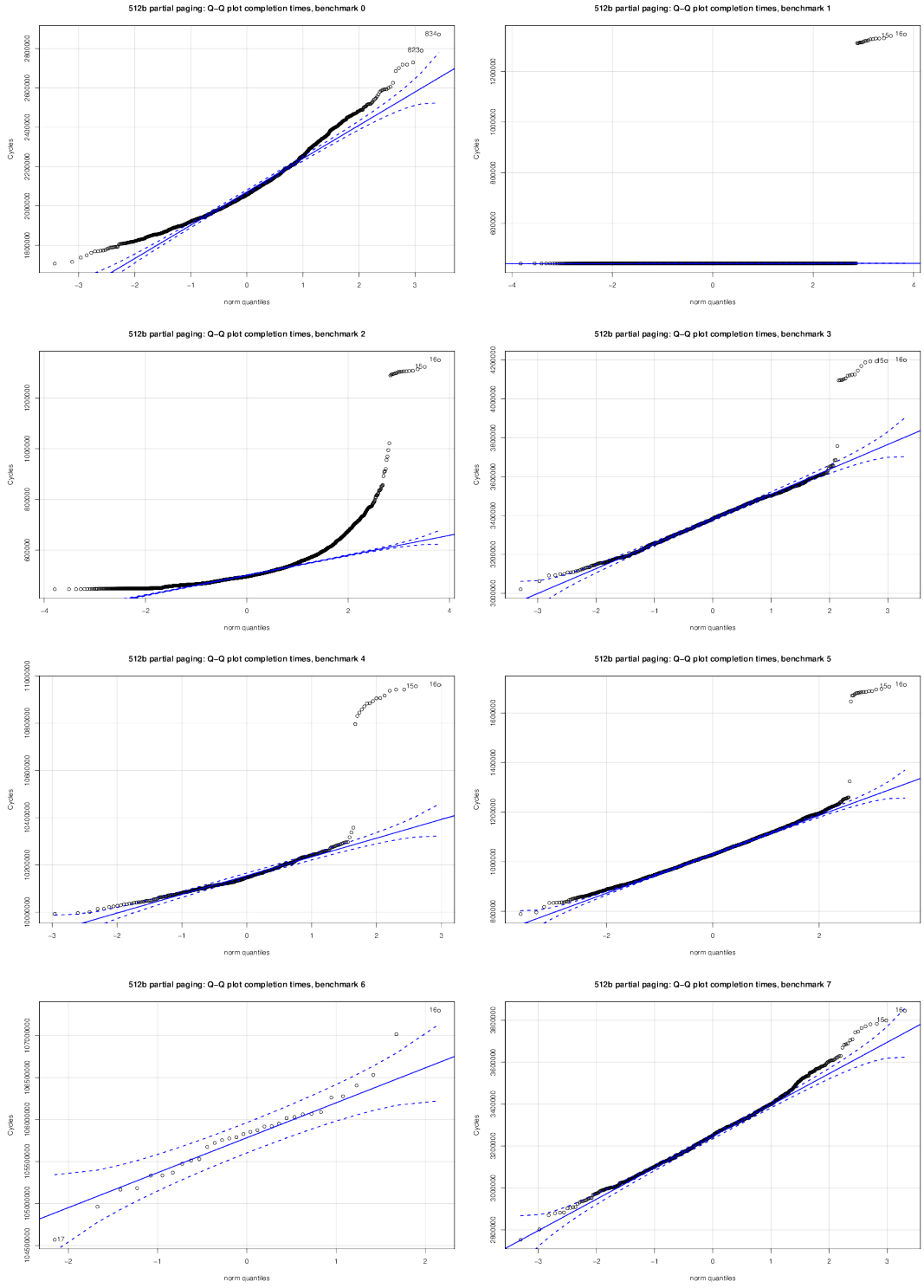

Figure 5.8: QQ Plots for 512 byte partial paging

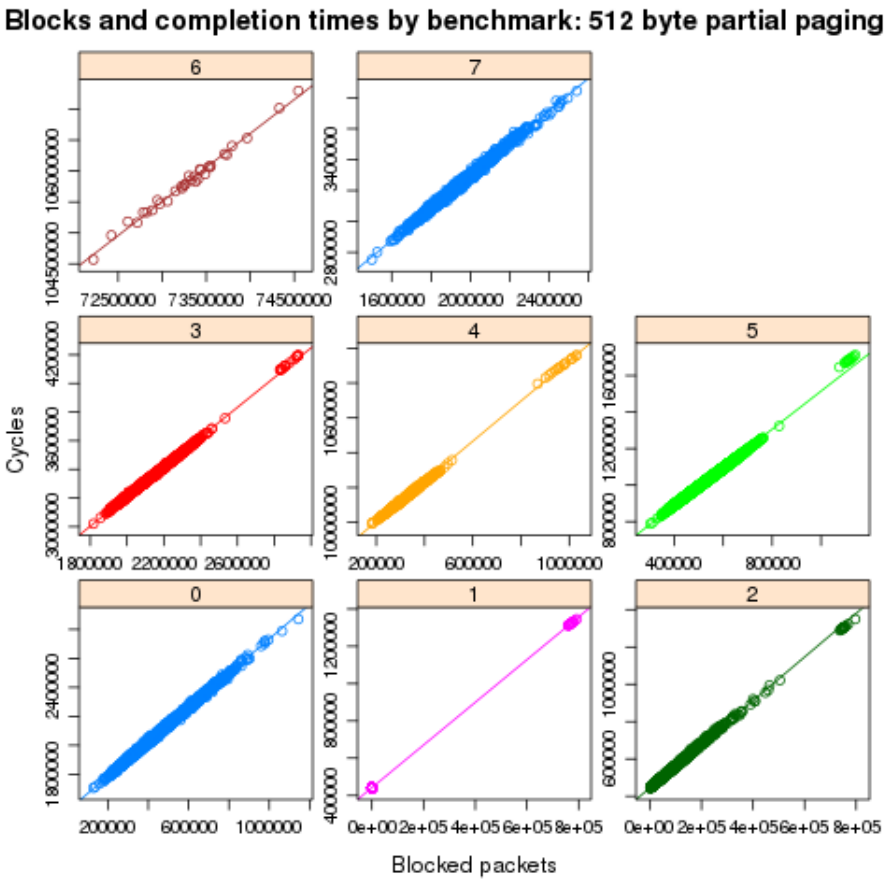

Figure 5.9: Completion times and blocked packets for all eight benchmarks with 512 byte partial paging

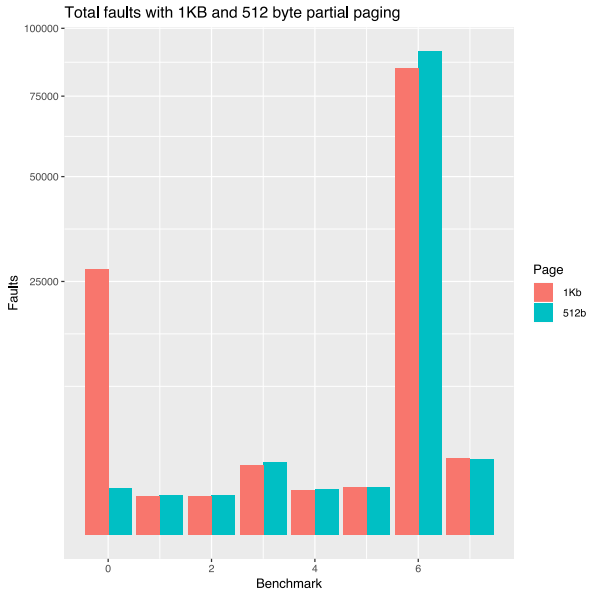

Figure 5.10: Mean total faults and remote memory requests (small and hard and remote page reads) for initial iteration with partial paging with different page sizes

environment for the other benchmarks. We can see a similar, if smaller, effect for 256 byte pages (see Section 5.3).

5.1.1 Traditional and delayed partial paging with 512-byte pages

Tables 5.3 and 5.4 show the performance of traditional paging with 512-byte pages: once again partial paging generally turns in better average performance than a whole-page system, though the traditional system can run benchmark 2, after the initial run, without the need to load any further pages. Average blocking, at 51.60 (12,206,041,433 blocks over 236,573,866 cycles) remains higher than for partial paging, though the gap is substantially smaller (Figure 5.11).

Tables 5.5 and 5.6 show mean performance data for the system for 512-byte pages with an additional cycle for bitmap checking. Once more benchmarks 3 and 6 show better performance (as measured by observed maximum timings) compared to the undelayed partial paging system.

Table 5.7 compares the extremes of performance for the different approaches: here (with benchmark 2 requiring no page loads after the initial run), traditional paging delivers the best mean performance and partial paging with a delay, the worst. But when considering the minimum efficiencies - in other words the slowest performance - likely to be of essential

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	3109766	1748323	1780333	4798261	11687305	2262785	236307671	7848475
Mean	2970309	1743186	1773689	4695856	11596715	2243299	233994214	7242201
Service cost	62013	45000	46800	254488	64400	88400	14765738	384350
Blocks	1331239	1233974	1259921	3593065	1725603	1730835	197934946	5830618
Count								
Hard Faults	31.5	25.0	26.0	66.1	32.0	34.0	5030.6	120.5
Remote Page Table Reads	132.3	100.0	104.0	423.8	136.0	168.0	26840.8	686.5
Share								
Service	2.1%	2.6%	2.6%	5.4%	0.6%	3.9%	6.3%	5.35
Blocks	44.8%	70.8%	71.0%	76.5%	14.9%	77.2%	84.6%	80.5%
Admin	10.6%	5.0%	5.0%	4.0%	13.0%	4.1%	3.4%	4.0%
Efficiency of execution	42.5%	21.6%	21.3%	14.0%	71.5%	14.8%	5.7%	10.2%

Table 5.3: Performance (means) of benchmarks with traditional paging and 512 byte pages: initial execution

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	2812052	441948	443433	3922287	10533013	1407555		7835679
Mean	2127548	441210	442664	3541281	10132798	1076417	NA	6640068
Service cost	34969	0	0	220490	16066	46733		405857
Blocks	543354	0	0	2489060	354755	623116		5211118
Count								
Hard Faults	13.3	0	0	43.1	7.3	10.2		112.8
Remote Page Table Reads	66.2	0	0	340.5	32.6	74.2		689.7
Share								
Service	1.6%	0%	0%	6.2%	0.2%	4.3%		6.1%
Blocks	25.5%	0%	0%	70.3%	3.5%	57.9%		78.5%
Admin	13.5%	14.6%	14.5%	4.8%	14.5%	6.9%		4.3%
Efficiency of execution	59.3%	85.4%	85.5%	18.6%	81.9%	30.9%		11.1%

Table 5.4: Performance (means) of benchmarks with traditional paging and 512 byte pages: continued execution

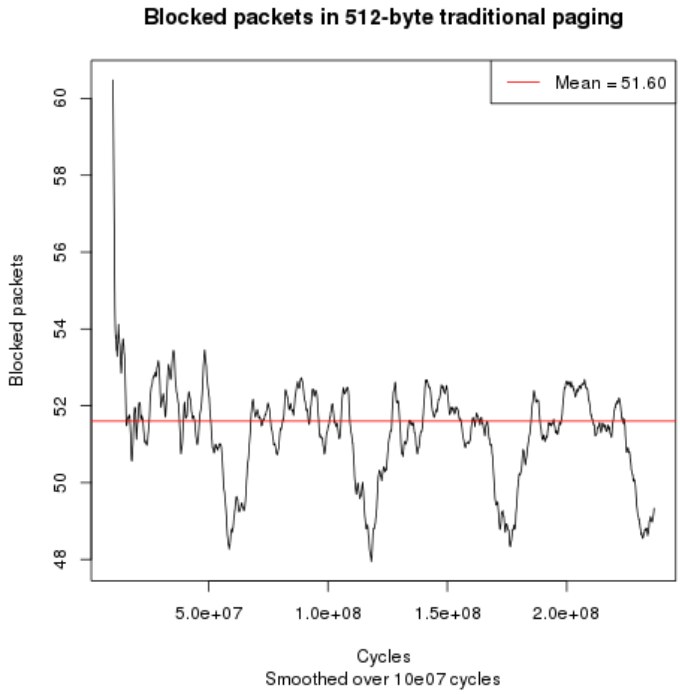

Figure 5.11: Blocking in 512 byte paged traditional paging system (smoothed)

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	3929618	1678454	1648156	4164409	20415399	1920967	86411477	3965871
Mean	3865328	1661561	1611810	4112368	20376230	1883279	85389130	3816706
Service cost	43950	30550	30200	234238	41550	73100	6005459	143978
Blocks	621048	666191	614128	2086484	671239	919640	37241795	1650193
Count								
Hard Faults	31.7	25.0	26.0	66.0	32.0	34.0	5038.5	116.6
Small Faults (and page table reads)	749.7	586.0	578.0	1996.0	779.0	852.0	84134.9	1971.6
Share								
Service	1.1%	1.8%	1.9%	5.7%	0.2%	3.9%	7.0%	3.8%
Blocks	16.1%	40.1%	38.1%	50.7%	3.3%	48.8%	43.6%	43.2%
Admin	50.1%	35.4%	36.6%	27.6%	55.8%	29.6%	33.6%	33.7%
Efficiency of execution	32.7%	22.7%	23.5%	16.0%	40.7%	17.6%	15.7%	19.3%

Table 5.5: Performance (means) of benchmarks with partial paging and 512 byte pages and 1 cycle delay for bitmap reads: initial execution

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	3801318	881422	1221198	3561582	19882062	1416608	85507815	3640310
Mean	3398690	880655	932802	3040383	19728385	1141262	85389130	3101858
Service cost	27492	0	4053	222107	17184	47750	6010116	148293
Blocks	226493	0	35489	1066360	156159	266072	35984701	941083
Count								
Hard Faults	14.2	0	3.5	44.0	9.5	11.0	5032.6	107.8
Small Faults (and page table reads)	376.3	0	72.3	1674.0	259.0	368.0	84034.5	1936.4
Share								
Service	0.8%	0%	0.4%	7.3%	0.1%	4.2%	7.1%	4.8%
Blocks	6.7%	0%	3.8%	35.1%	0.8%	23.3%	42.8%	30.3%
Admin	55.4%	57.2%	55.2%	36.0%	57.1%	43.4%	34.1%	41.2%
Efficiency of execution	37.1%	42.8%	40.6%	21.7%	42.1%	29.1%	16.0%	23.7%

Table 5.6: Performance (means) of benchmarks with partial paging and 512 byte pages and 1 cycle delay for bitmap reads: continued execution

Paging approach	Max Efficiency	Mean Efficiency	Min Efficiency	Max blocks per hard fault	Mean blocks per hard fault	Min blocks per hard fault
Traditional	0.855	0.709	0.057	87263	53732	25645
Partial (no delay)	0.854	0.660	0.125	69430	26893	1417
Partial (1 tick delay)	0.428	0.356	0.156	52274	16277	182

Table 5.7: Range of performance parameters for different paging approaches with 512 byte pages

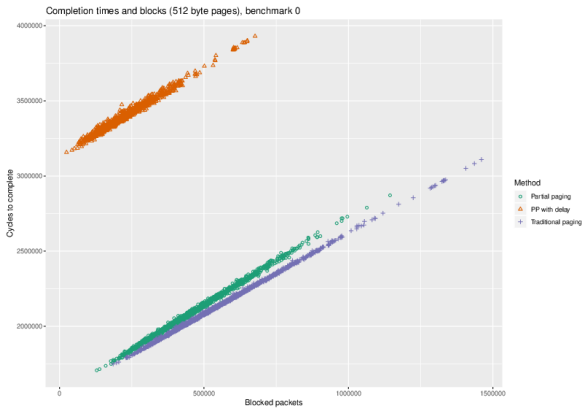

Figure 5.12: Blocks and performance compared for 512 byte paging for benchmark 0

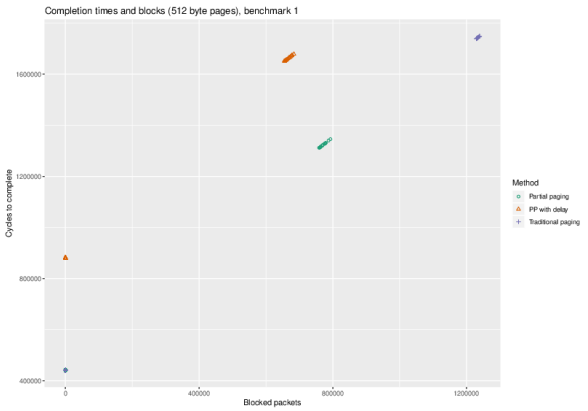

Figure 5.13: Blocks and performance compared for 512 byte paging for benchmark 1

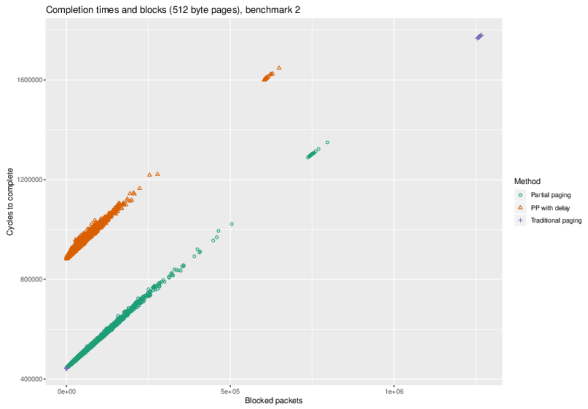

Figure 5.14: Blocks and performance compared for 512 byte paging for benchmark 2

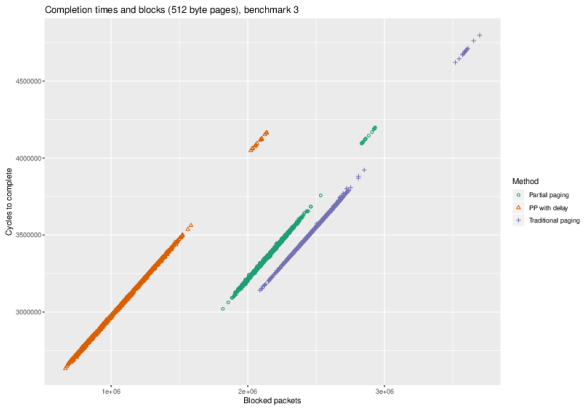

Figure 5.15: Blocks and performance compared for 512 byte paging for benchmark 3

interest when designing a real-time system, the opposite is the case and least worst minimum efficiency is seen with the delayed approach.

Blocking has fallen to an average of just 23.3 (4,314,191,126 blocks over 184,849,513 cycles).

5.2 WORST CASE EXECUTION TIMES WITH 512-BYTE PAGES

5.2.1 Maximum observed completion times for 512-byte pages

Figures 5.12 - 5.19show the range of completion timings for the different benchmarks and the bimodal distribution of completion times is clearly seen for each method.

Figure 5.20 plots the observed worst execution times for each benchmark for both 1KB and 512 byte paging and across the different approaches used.

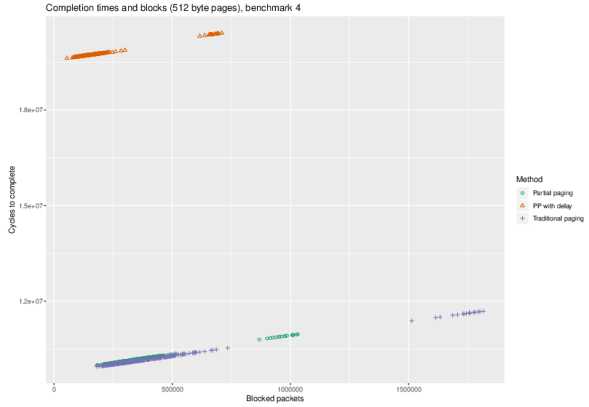

Figure 5.16: Blocks and performance compared for 512 byte paging for benchmark 4

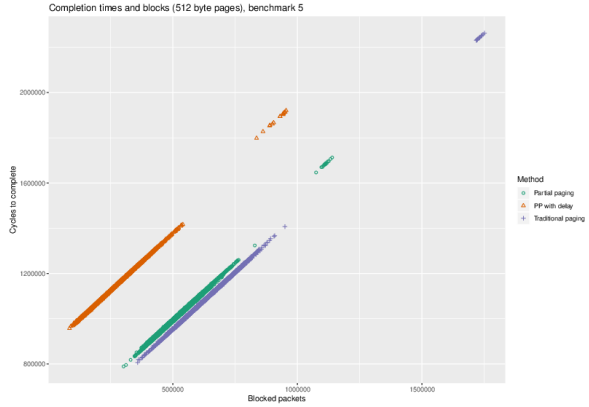

Figure 5.17: Blocks and performance compared for 512 byte paging for benchmark 5

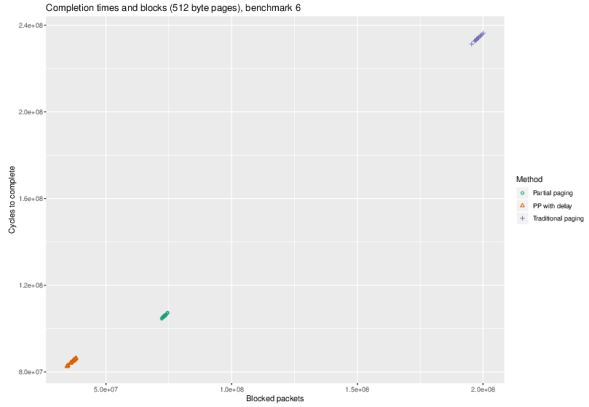

Figure 5.18: Blocks and performance compared for 512 byte paging for benchmark 6

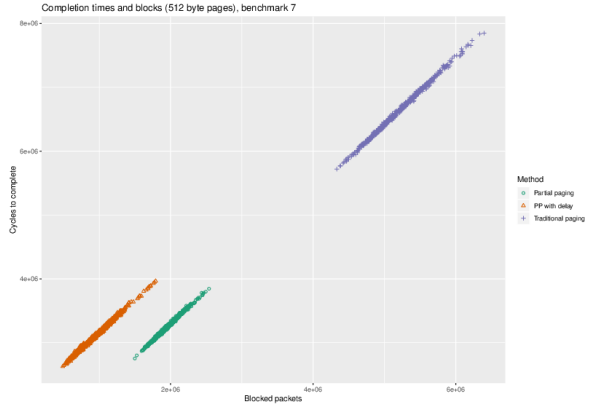

Figure 5.19: Blocks and performance compared for 512 byte paging for benchmark 7

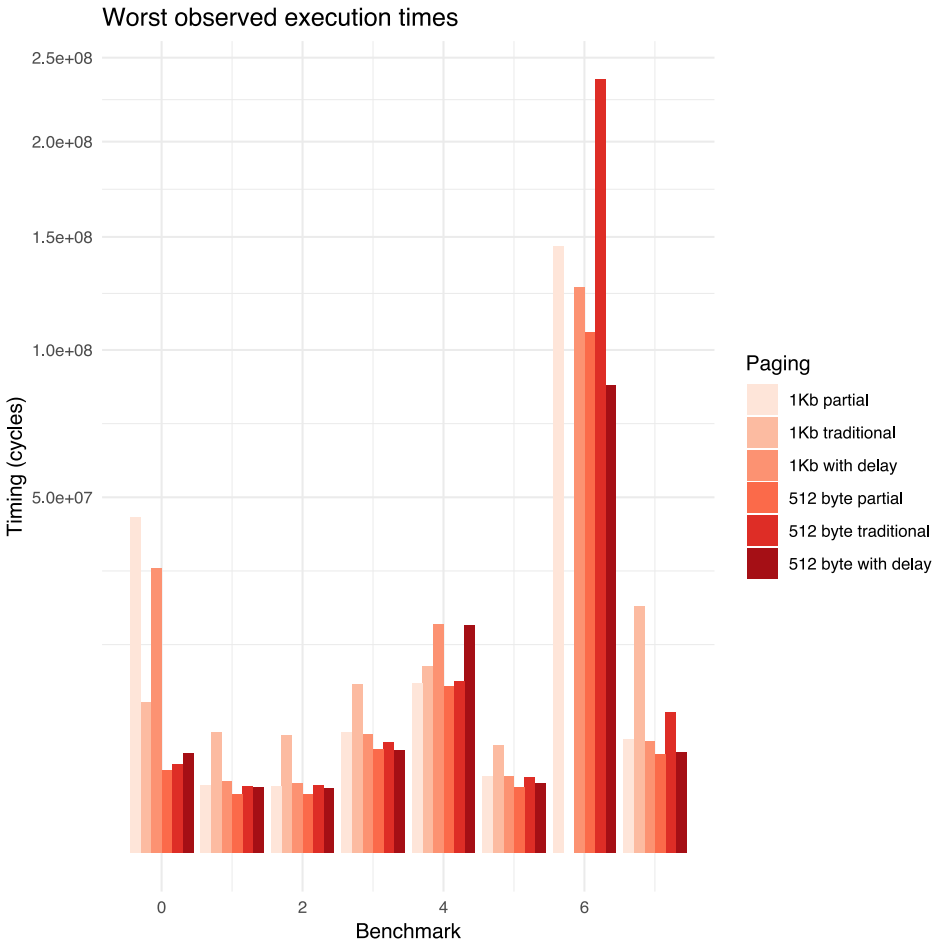

Figure 5.20: Observed worst execution times compared for different page sizes and paging paradigms

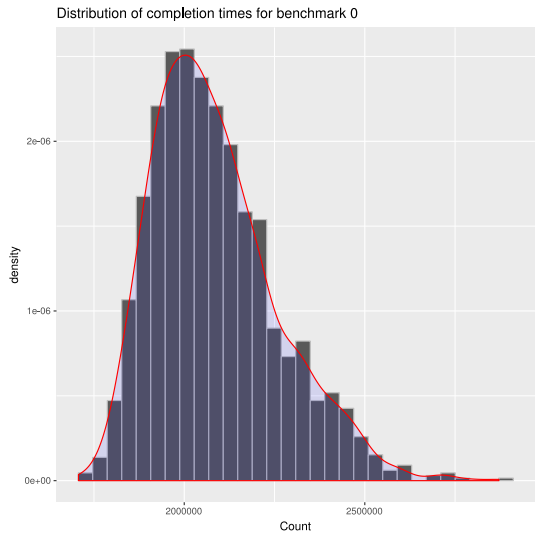

Figure 5.21: Distribution of completion times for benchmark 0 with 512-byte pages under partial paging

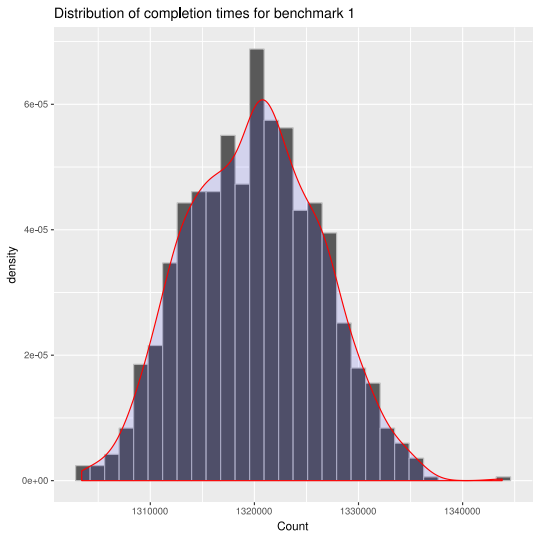

Figure 5.22: Distribution of completion times for benchmark 1 initial iteration with 512-byte pages under partial paging

5.2.2 EVT analysis of 512 byte page maxima

The quantile-quantile plots for completion times for the benchmarks under partial paging (Figure 5.8) and the bimodal distribution of completion times they reveal show that to undertake an EVT analysis we need to collect data from multiple initial runs of the system for every benchmark except benchmark 0. Accordingly we ran the system 75 times and collected data for benchmarks 1, 2, and 5 (each generating 1200 completion times). We also used the data for benchmark 0 from the general test run.

Figures 5.21 - 5.24 show that completion times are (with the exception of benchmark 5) skewed right suggesting a long tail of maxima.

Although our sample times do not have the i.d.d. property - they are inherently dependent on whole system behaviour, including that of other benchmarks - we can assume stationarity as we are only considering the initial runs of the system. The exception to this is benchmark 0, where we are using a wider set of results. Here an ADF test does report stationarity ($p < 0.01$ where the alternative hypothesis is stationarity).

We again tested the collected values for fit against both the Gumbel and Fréchet distributions (Table 5.8).

In fact, in the case of benchmark 1 we were forced to make a fit on a sample size of just 21 - below the "generally accepted" limit of 30 for reliable results [76] and so the results shown here need to be treated with caution and regarded as for illustrative purposes only. For

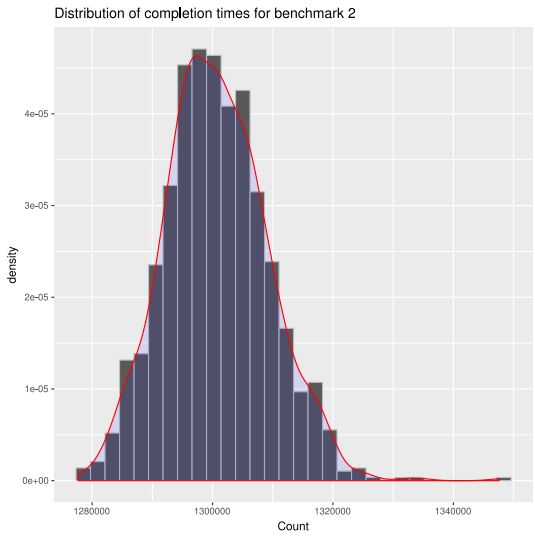

Figure 5.23: Distribution of completion times for benchmark 2 initial iteration with 512-byte pages under partial paging

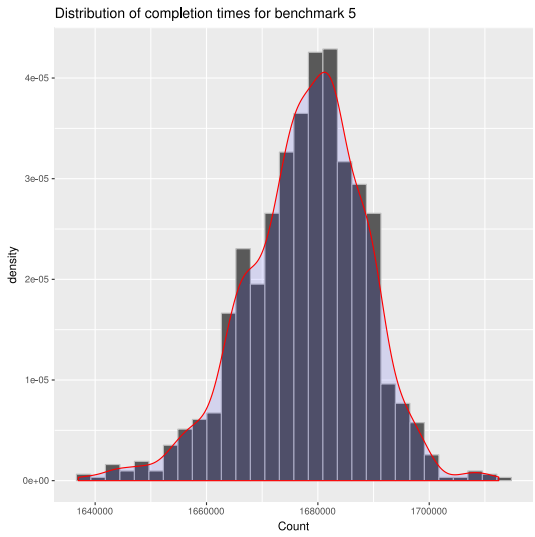

Figure 5.24: Distribution of completion times for benchmark 5 initial iteration with 512-byte pages under partial paging

benchmark 2 the Fréchet fit is better than that for Gumbel and this also suggest more results are needed (cf. Section 4.9.2).

Appendix H contains the plots for each of the fitted Gumbel distributions for 512 byte partial paging.

The results suggest that small page sizes comfortably retain their advantage even at extreme probabilities. Indeed, the computed values of β - the shape parameter that indicates the spread of completion times - is smaller for 512 byte partial paging than for 1024 byte partial paging in every case where a comparison exists.

Benchmark	p-value for Gumbel	Block size used	Sample size used	p-value for Fréchet	Block size used	Sample size used
0	0.3155	32	51	0.2282	32	51
1	0.1705	56	21	0.169	56	21
2	0.2012	32	37	0.203	32	37
5	0.09187	16	75	0.08866	16	75

Table 5.8: Goodness of fit tests for Gumbel and Fréchet distributions for partial paging with 512-byte page size

512-byte partial paging	0	1	2	5
Observed maximum	2871432	1343799	1347581	1712485
Computed μ	2452575	1333297	1317823	1692572
Standard error for computed μ	5931	426	739	538
Computed β	99198	1858	4285	4418
Standard error for computed β	4194	314	544	391
Computed percentage for observed maximum	≈98.5%	≈ 99.6%	≈ 99.9%	≈ 98.9%
Computed threshold for 10^{-1} maxima (i.e. 0.9 CDF)	2675807	1337478	1327466	1702514
ditto 10^{-3}	3137761	1346131	1347421	1723088
ditto 10^{-6}	3823046	1358966	1377022	1753609
ditto 10^{-9}	4508282	1371801	1406622	1784127
ditto 10^{-12}	5193519	1384635	1436222	1814646
ditto 10^{-15}	5878832	1397472	1465825	1845168
Observed maximum for traditional paging	3109766	1748323	1780333	2262785
Threshold equivalent for partial paging (% CDF)	≈ 99.9%	≈ 100%	≈ 100%	‘ ≈ 100%

Table 5.9: Estimated Gumbel distribution for benchmarks 0, 1, 2 and 5 with 512-byte partial paging

Interestingly the figures also suggest that our CLOCK is still relatively poorly tuned to benchmark 0 at this page size as we see a large β and a long tail.

In terms of our simple entropy model (cf., 4.9.1), at a mean blocking of 45.2 we calculate an entropy of 387, 53% of that seen for 1KB partial paging.

5.3 SMALLER PAGES: 256 BYTE PAGES AND 128 BYTE PAGES

We have shown how reducing page size from 1024 bytes to 512 bytes appears to significantly enhance performance across the range of benchmarks even though available memory is generally still smaller than that required to accommodate programs' working sets. This appears to be an emergent characteristic of the system: although the number of hard faults increases for all benchmarks except benchmark 0, that benchmark's dramatic fall in fault rates lowers system congestion enough to both lower completion times and - by lowering system entropy - limits the predicted WCETs for the benchmarks yet further.

Smaller pages will not, however, continue indefinitely to increase the efficiency of the system. Smaller pages may reduce fragmentation but they also increase the fault rate and require larger page tables, taking away time and space which could otherwise be used for program execution. We tested this effect by reducing page size to first 256 bytes and then to 128 bytes.

5.3.1 256 byte pages - mean performance with partial paging

For 256 byte pages we have 64 page frames available locally, and we need 4 page frames for notional kernel routines and 4 page frames for a local stack (keeping each of these as 1KB in total as with previous simulations). As the bitmap size is fixed (1024 bits are needed to map 16KB of local memory as 16-byte lines) and as a 256 byte page can accommodate 2048 bits, this can fit inside 1 page, so releasing 256 bytes compared to 512-byte paging. But the local page tables are made up of fixed sized entries (28 bytes long) and so we now require 7 page frames to map our 64 pages - leaving us with $\frac{48}{64}$ (75%) of local page frames for general use - a smaller proportion than with 512 byte pages, where the fraction of usable space was $\frac{25}{32}$ or 78.125%: effectively we have lost two 256-byte page frames[2].

2 In a traditional approach we have 49 free page frames (76.5625% of all space) for 256-byte pages and 26 (81.25%) for 512-byte pages, so the loss here is the equivalent of three 256-byte page frames, though we still have one more page frame than we with partial paging.

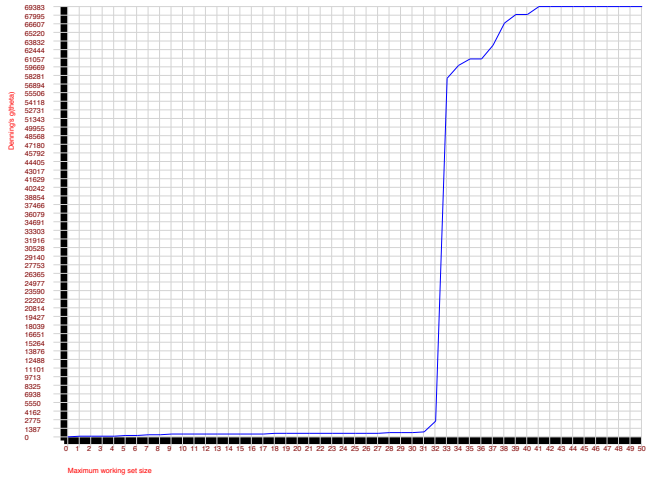

Figure 5.25: LRU lifetime curve for benchmark 0 with 256 byte pages

Using smaller page sizes will significantly increase the fault rate for an individual benchmark on its initial run: for benchmark 0 the mean hard fault count now rises to 50.0 and each extra hard fault also generates 4 additional remote page table reads.

The smaller page size does lead to less blocking in the system - an average of 34.03 blocked packets over 2.03×10^8 cycles[3]: with around 27% of all computing time being lost to blocked packets in the memory tree (see Figure 5.26). And, as is shown below (Tables 5.10 and 5.11) the smaller page size can deliver a better fit with limited memory resources (here benchmark 2 fits inside 42 256-byte pages and so generates no faults after the initial run). It is this fall in the system-wide fault count (along with a small contribution from benchmark 7) that appears to drive the overall lower blocking and better average (mean) performance.

The typical or average case is worse for 256-byte pages in four cases though the worst-case observed times are better for 256-byte pages for all cases except benchmark 1 (see Figure 5.27). This is consistent with the impact of entropy in the system: the lower amount of blocking in the 256 byte paging system lessens the range of possible and observed timing outcomes. The distribution of timings is again bimodal (see Figures 5.28 - 5.35) and so a closer examination of the maxima with EVT methods is required.

3 6,922,322,157 blocked packets over 203,390,630 cycles.

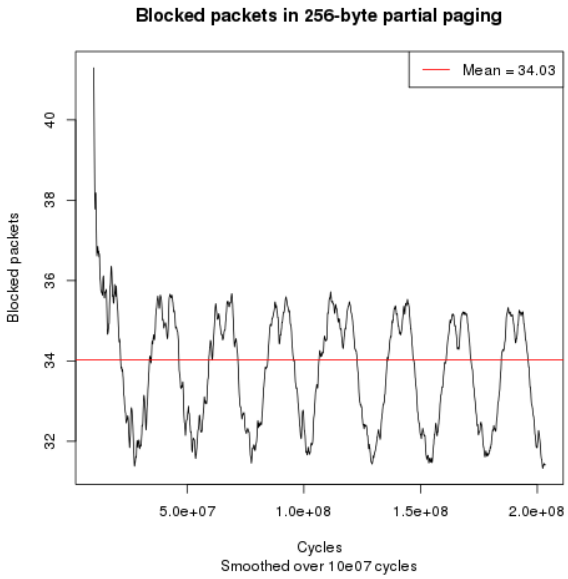

Figure 5.26: Blocked packets with 256-byte pages under partial paging (smoothed over 10,000,000 cycles)

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	2514678	1357224	1316858	3840749	10929818	1528350	102077157	3202193
Mean	2501839	1348234	1309466	3806664	10896744	1520754	101277686	3026640
Service cost	43575	33950	33400	230900	43763	59300	7243922	122413
Blocks	730514	775433	735726	2524815	806768	952774	64982307	1775537
Count								
Hard Faults	50.0	42.0	42.0	118.0	52.0	55.0	9628.3	126.8
Small Faults (and page table reads)	769.5	637.0	626.0	2267.3	821.3	907.0	104054.0	1712.1
Share								
Service	1.7%	2.5%	2.6%	6.1%	0.4%	3.9%	7.2%	4.0%
Blocks	29.2%	57.5%	56.2%	66.3%	7.4%	62.7%	64.2%	58.7%
Admin	18.6%	12.0%	12.4%	10.3%	16.1%	11.6%	15.4%	13.0%
Efficiency of execution	50.5%	28.0%	28.9%	17.3%	76.1%	21.9%	13.3%	24.3%

Table 5.10: Performance (means) of benchmarks with partial paging and 256 byte pages: initial execution

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	2049874	451757	454168	3136864	10096896	880703	101418175	2907958
Mean	1781897	449512	451767	2825495	9933229	687082	100751205	2349550
Service cost	11813	0	0	221260	7009	3.9%	7254988	120270
Blocks	130736	0	0	1612131	97209	232131	64450433	1117599
Count								
Hard Faults	9.9	0	0	82.1	8.0	11.2	9624.6	108.2
Small Faults (and page table reads)	158.0	0	0	1832.6	130.7	229.9	104102.6	1564.0
Share								
Service	0.7%	0%	0%	7.8%	0.1%	3.9%	7.2%	5.1%
Blocks	29.2%	0%	0%	57.1%	1.0%	33.8%	64.0%	47.6%
Admin	21.2%	16.2%	16.3%	11.8%	15.4%	13.9%	15.5%	16.0%
Efficiency of execution	70.8%	83.8%	83.7%	23.3%	83.5%	48.4%	13.3%	31.3%

Table 5.11: Performance (means) of benchmarks with partial paging and 256 byte pages: continued execution

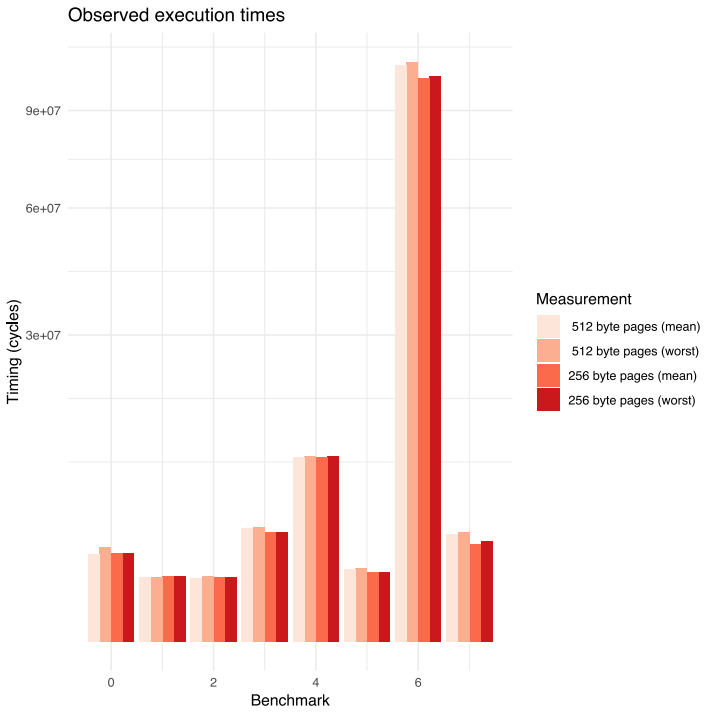

Figure 5.27: Timings and faults for 256 byte and 512 byte partial paging (first iterations only for mean)

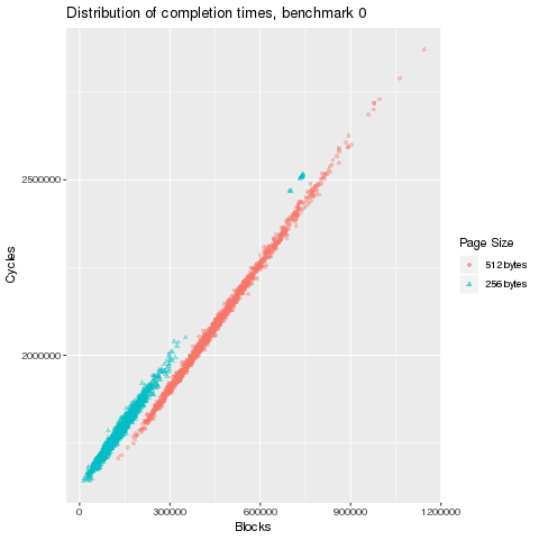

Figure 5.28: The range of completion times for benchmark 0 with 512-byte and 256-byte partial paging

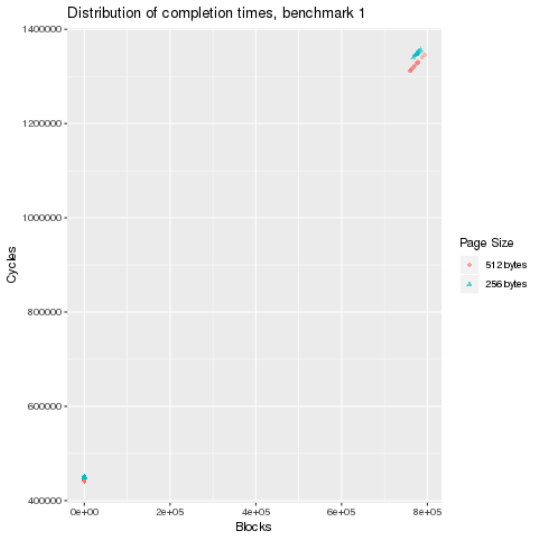

Figure 5.29: The range of completion times for benchmark 1 with 512-byte and 256-byte partial paging

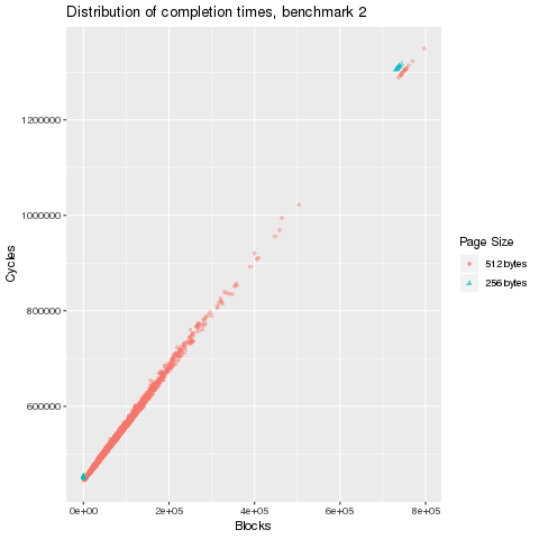

Figure 5.30: The range of completion times for benchmark 2 with 512-byte and 256-byte partial paging

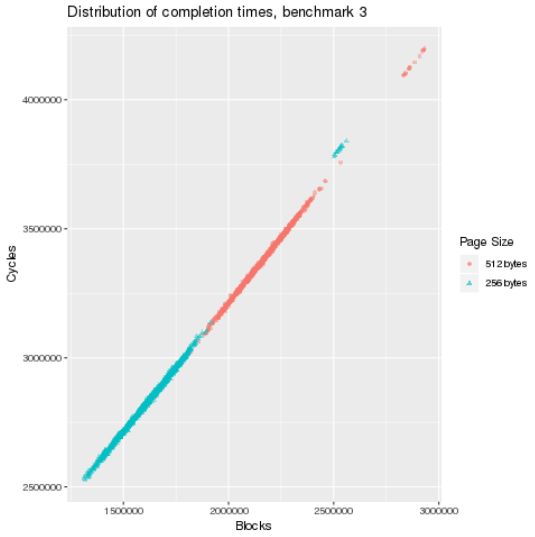

Figure 5.31: The range of completion times for benchmark 3 with 512-byte and 256-byte partial paging

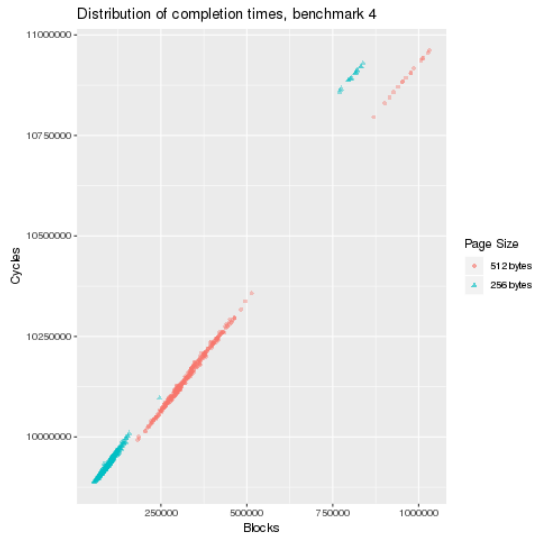

Figure 5.32: The range of completion times for benchmark 4 with 512-byte and 256-byte partial paging

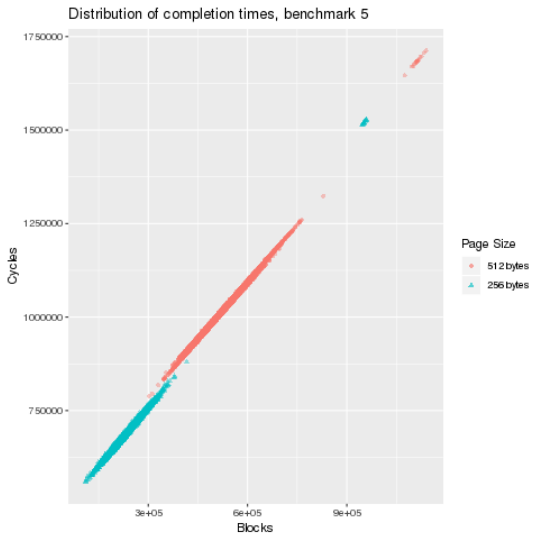

Figure 5.33: The range of completion times for benchmark 5 with 512-byte and 256-byte partial paging

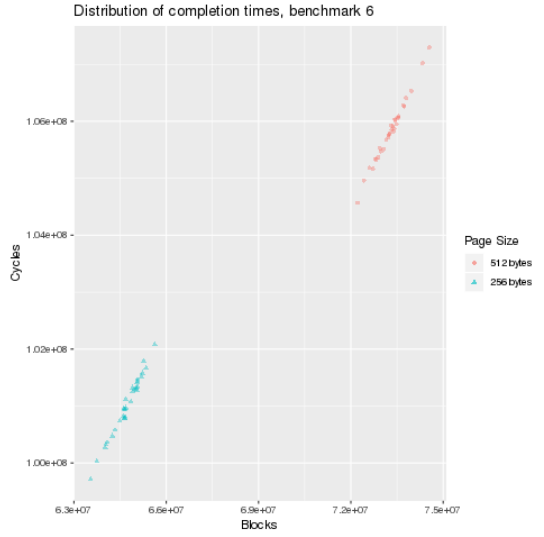

Figure 5.34: The range of completion times for benchmark 6 with 512-byte and 256-byte partial paging

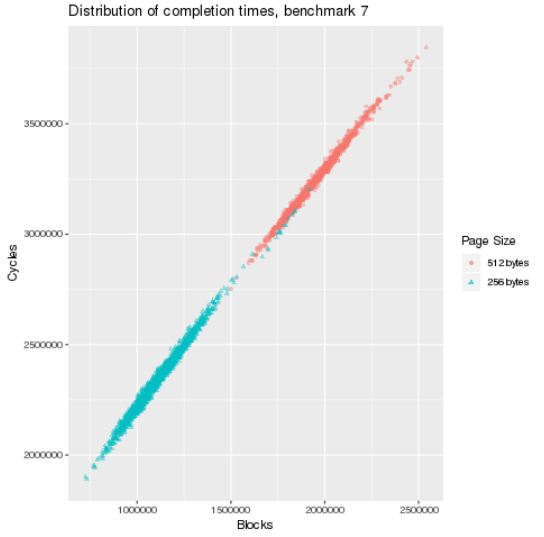

Figure 5.35: The range of completion times for benchmark 7 with 512-byte and 256-byte partial paging

Benchmark	p-value for Gumbel	Block size used	Sample size used	p-value for Fréchet	Block size used	Sample size used
0	0.08031	16	77	0.07879	16	77
1	0.2123	32	38	0.2104	32	38
2	0.9001	24	51	0.9054	24	51
5	0.1196	16	77	0.1179	16	77

Table 5.12: Goodness of fit tests for Gumbel and Fréchet distributions for partial paging with 256-byte page size

Benchmark	Computed μ	SE for μ	Computed β	SE for β	Memo: μ for 512-byte pages	Memo: β for 512-byte pages	10^{-9} timing for 256-bytes	10^{-12} timing for 256-bytes	10^{-15} timing for 256-bytes	Memo: 10^{-15} timing for 512-bytes
0	2516976	335	2783	237	2452575	99198	2574649	2593873	2613100	5878832
1	1357058	210	1223	149	1333297	1858	1382403	1390851	1399300	1397472
2	1314504	217	1461	138	1317823	4285	1344781	1354873	1364966	1465825
5	1526161	196	1629	142	1692572	4418	1559919	1571172	1582426	1845168

Table 5.13: Computed completion times, based on Gumbel distribution, for 256-byte pages

5.3.2 EVT analysis with 256-byte pages

Similarly to above (cf. Section 5.2.2) we collected initial timings from 77 separate runs of the 256-byte paging system for benchmarks 0, 1, 2 and 5. Figures 5.36 - 5.39show the distribution of completion times.

Table 5.13 shows the computed μ and β for the benchmarks. The values of β are significantly smaller for 256-byte partial paging than for 512 byte partial paging (though the error margins are large as a proportion of calculated value) and would seem reasonable to assert that, in this case, the lower entropy in the system as a result of lower blocking makes the smaller page size a good choice for real-time applications even where the typical (mean) performance is slower.. Appendix I contains the plots for each of the fitted Gumbel distributions for 256 byte partial paging.

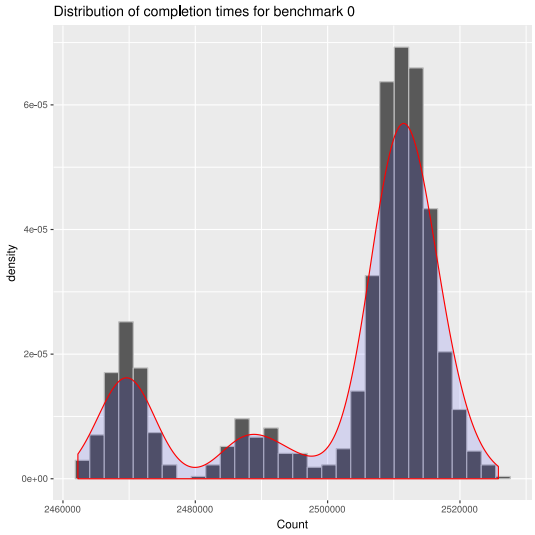

Figure 5.36: Distribution of completion times for benchmark 0 initial iteration with 256-byte pages under partial paging

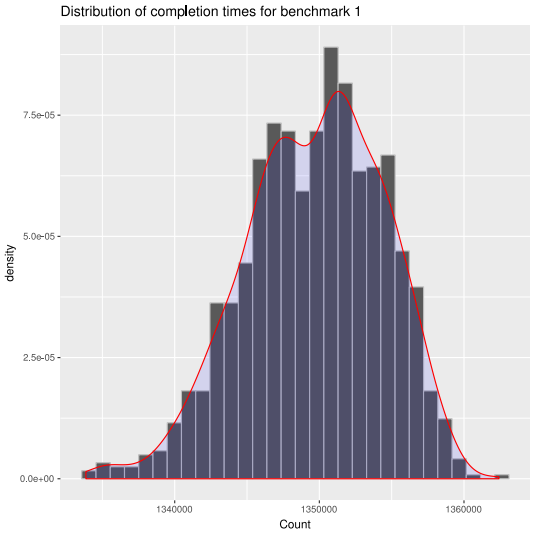

Figure 5.37: Distribution of completion times for benchmark 1 initial iteration with 256-byte pages under partial paging

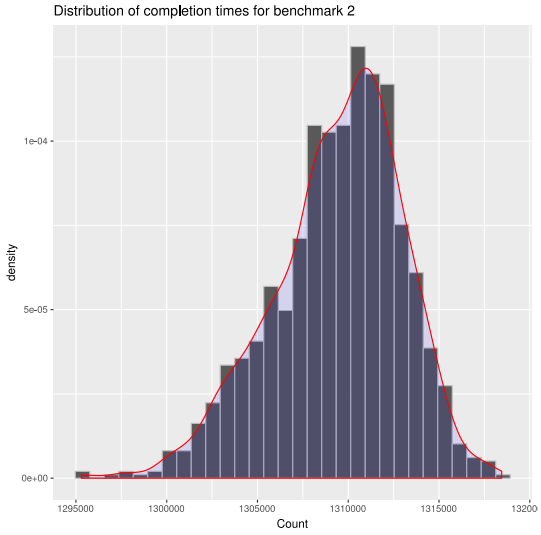

Figure 5.38: Distribution of completion times for benchmark 2 initial iteration with 256-byte pages under partial paging

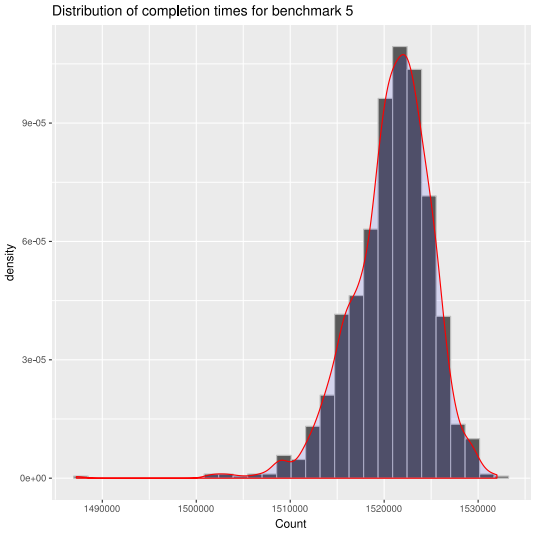

Figure 5.39: Distribution of completion times for benchmark 5 initial iteration with 512-byte pages under partial paging

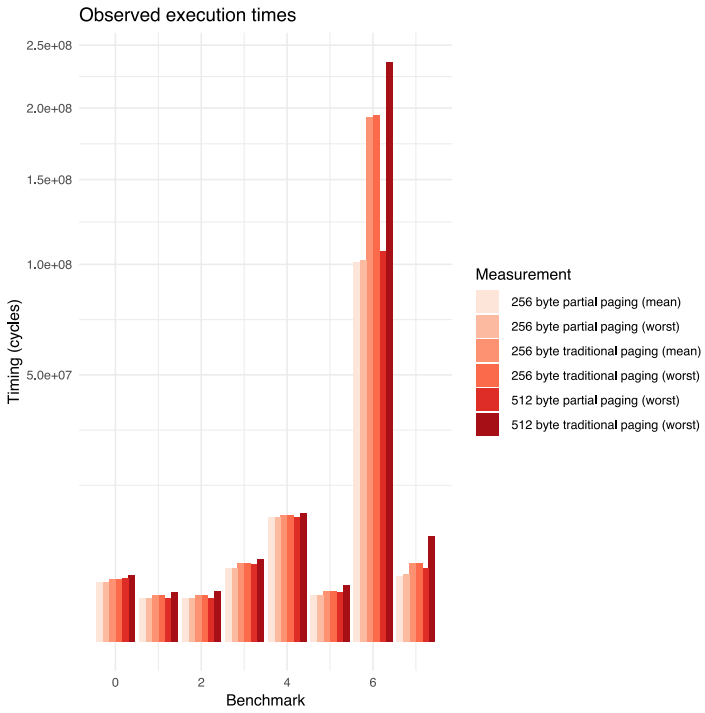

Figure 5.40: Traditional and partial paging timings, in cycle times, compared for 256-byte pages (initial iteration only for averages)

5.3.3 256-byte pages with traditional paging

Figure 5.40 (cf. Figure 5.27) shows that traditional paging under-performs partial paging for 256-byte pages[4]. The average amount of blocked packets, at 41.1 (8,027,129,005 blocked packets over 195,222,155 cycles) is lower than for 512-byte partial paging and 256-byte traditional paging outperforms 512-byte partial paging for benchmark 0, though has an observed worst case execution time for benchmark 6 that is 82% slower than for partial paging. That gap is so large we think it unlikely that, even at safety-critical thresholds, the smaller-paged traditional system would out-perform the larger-paged partial paging system, though we have not tested this assumption.

5.3.4 128-byte pages with partial paging

With 128-byte partial paging we have 128 local page frames but require 28 of these to accommodate the page tables. Taking into account space we set aside for notional kernel routines and a local stack, as well as for bitmaps, we are left with $\frac{83}{128}$ of page frames for general

4 The comparison here is of the first iteration of each benchmark as timings are again bi-modally distributed.

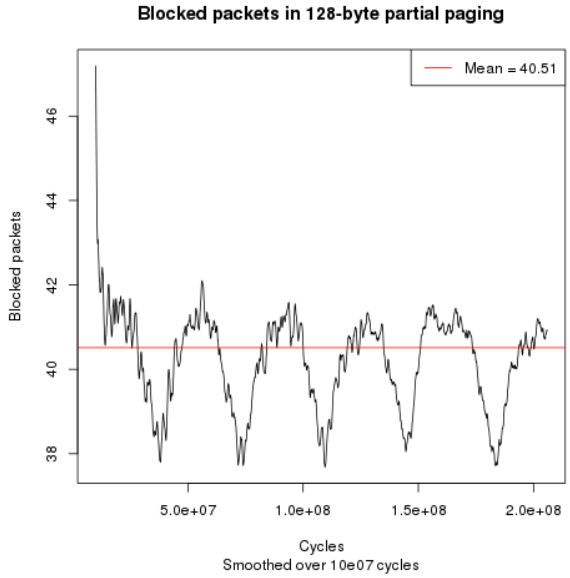

Figure 5.41: Blocked packets with 128-byte pages under partial paging (smoothed over 10,000,000 cycles)

use (64.8% of all local memory) with a partial paging model, while for a traditional approach we would have $\frac{84}{128}$ (65.6% of all local memory)- a substantial loss of capacity compared to either 512 byte pages or 256 byte pages.

Blocking now rises compared to 256-byte partial paging: with an average of of 40.51 blocked packets (8,341,797,822 blocked packets over 205,930,640 cycles) - see Figure 5.41.

Tables 5.14 and 5.15 also indicate the rising cost of maintaining a paging system with 128 pages: with the "admin" component taking a bigger share of a bigger total time cost: this element includes the costs of walking the page tables when handling faults. Costs of servicing memory requests and small fault counts (which include remote page table reads) also rise - reflecting the additional costs of managing a system with such a large number of pages: for instance for benchmark 0 the number of hard faults on initial execution has risen from 50.0 (see Table 5.10) with 256 byte pages to 101.1 with 128 byte pages (more or less reflecting the halving of page size), these additional (mean) $\approx$ 51.1 hard faults will generate at least 204 additional reads of the remote page tables - probably more because of additional page write-backs - and indeed the number of small faults (which include table reads) recorded rises on average by 217.

Figure 5.42 compares the observed worst execution times for partial paging with differently-sized pages and in every case the performance is worse for 128-byte pages than for 256-byte pages, and while benchmark 7 still has a lower observed worst execution time with this size of page than with 512-byte pages, in other cases it is worse.

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	3068840	1550763	1497607	4567862	11612715	1897730	148349057	3633858
Mean	3023533	1541446	1488820	4529518	11572521	1875484	145932294	3537339
Service cost	59072	40750	40400	264869	52450	75100	9228416	135153
Blocks	981194	904479	849438	3098090	981710	1231761	98770449	2132330
Count								
Hard Faults	101.1	76.0	77.0	222.1	96.0	102.0	16799.3	190.4
Small Faults (and page table reads)	986.5	739.0	731.0	2855.9	953.0	1096.0	136859.6	1925.2
Share								
Service	2.0%	2.6%	2.7%	5.8%	0.5%	4.0%	6.3%	3.8%
Blocks	32.5%	58.7%	57.1%	68.4%	8.5%	65.7%	67.7%	60.3%
Admin	23.8%	14.2%	14.8%	11.2%	19.4%	12.6%	16.8%	15.1%
Efficiency of execution	41.7%	24.5%	25.4%	14.5%	71.7%	17.7%	9.2%	20.8%

Table 5.14: Performance (means) of benchmarks with partial paging and 128 byte pages: initial execution

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	2951335	494889	497402	4022633	10585418	1281317		3542333
Mean	2391844	483565	485401	3722582	10405716	1032079	NA	3537339
Service cost	36023	0	0	268992	15978	50114		135763
Blocks	447162	0	0	2336904	258092	492560		1495933
Count								
Hard Faults	52.2	0	0	174.1	26.7	33.2		173.4
Small Faults (and page table reads)	527.6	0	0	2552.6	285.6	477.2		1814.2
Share								
Service	1.5%	0%	0%	7.2%	0.2%	4.8%		4.7%
Blocks	18.7%	0%	0%	62.8%	2.5%	47.7%		51.6%
Admin	27.0%	22.1%	22.1%	12.3%	17.6%	15.2%		18.4%
Efficiency of execution	52.8%	77.9%	77.9%	17.7%	79.7%	32.2%		25.4%

Table 5.15: Performance (means) of benchmarks with partial paging and 128 byte pages: continued execution

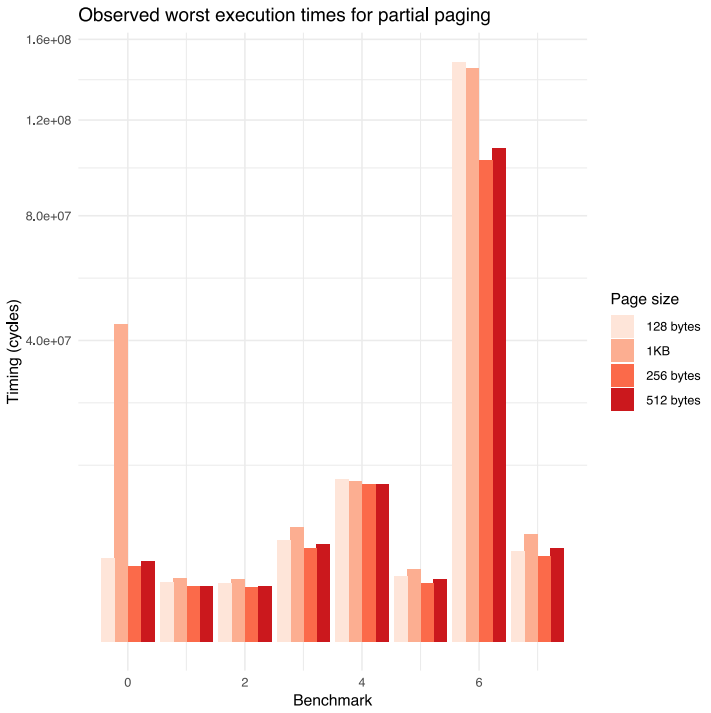

Figure 5.42: Observed worst execution times for each benchmark for partial paging with differently-sized pages

As our exploration of EVT-based timings has shown, lower blocking in the memory tree means that at safety critical margins, 128-byte pages might still be a better option for some benchmarks than both 512-byte pages and 1024-byte pages. We did not collect the data to formally test this but note that the range of completion times for different page sizes[5] (Figures 5.43 - 5.50) shows that the gap between the observed performance of 512-byte pages and 128-byte pages is generally (all cases except benchmark 7) larger than that between 512-byte pages and 256-byte pages.

5.4 FIFO PAGE REPLACEMENT

The increasing proportional cost of maintaining a semi-ordered list of page frames under a CLOCK policy, as we eliminate blocks on the memory tree or use smaller page sizes, leads

5 We have excluded 1024-byte pages from the plot of results of benchmark 0 because the large gap between those results and all others. In all cases except benchmark 0 we only plot the density for the initial runs for page sizes other than 1024 bytes because these contribute the observed worst case. For benchmark 0 we show all instances of 512-byte paging.

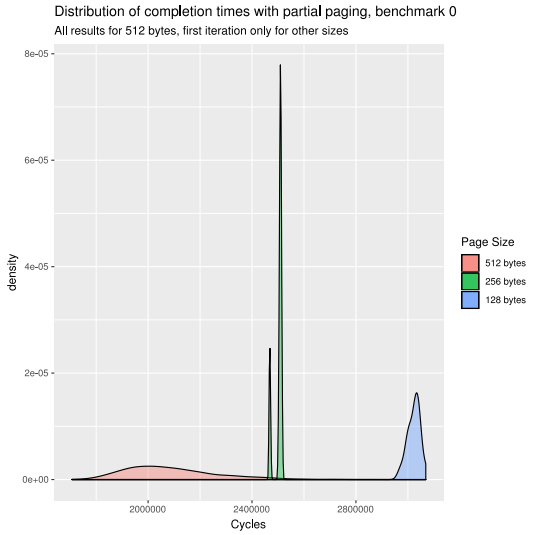

Figure 5.43: Density plots of observed timings for different page sizes with partial paging (benchmark 0)

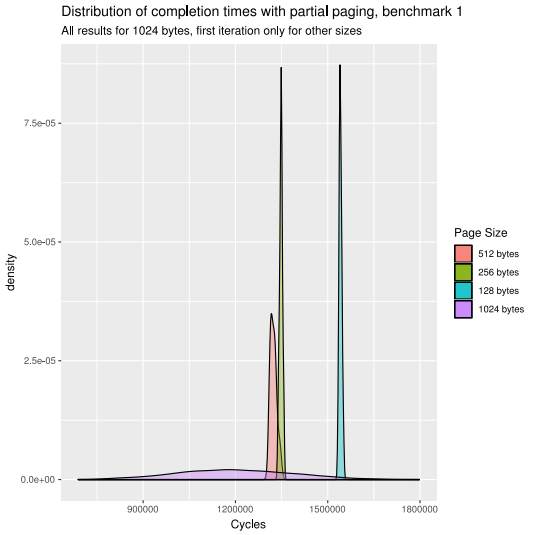

Figure 5.44: Density plots of observed timings for different page sizes with partial paging (benchmark 1)

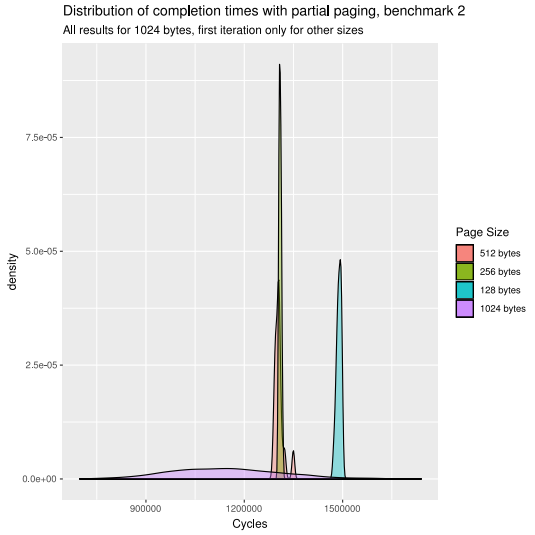

Figure 5.45: Density plots of observed timings for different page sizes with partial paging (benchmark 2)

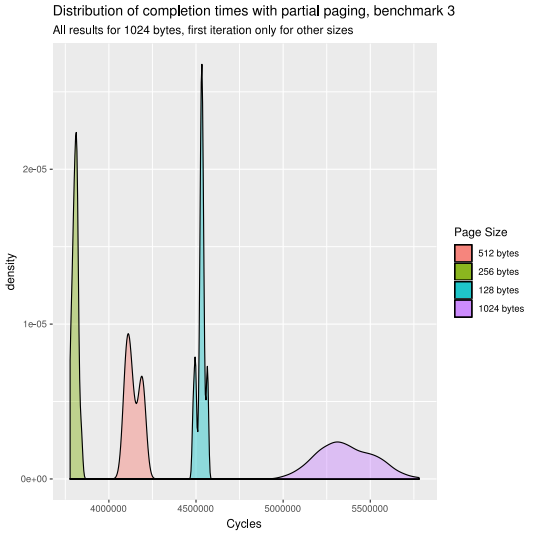

Figure 5.46: Density plots of observed timings for different page sizes with partial paging (benchmark 3)

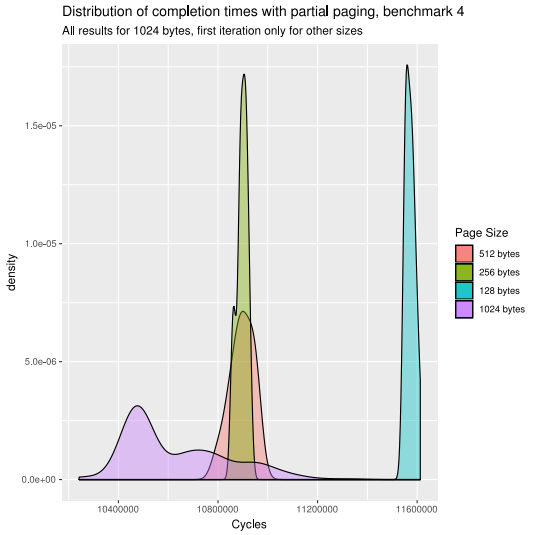

Figure 5.47: Density plots of observed timings for different page sizes with partial paging (benchmark 4)

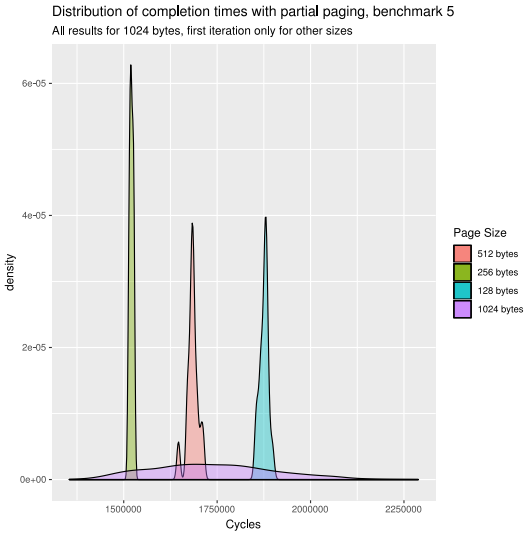

Figure 5.48: Density plots of observed timings for different page sizes with partial paging (benchmark 5)

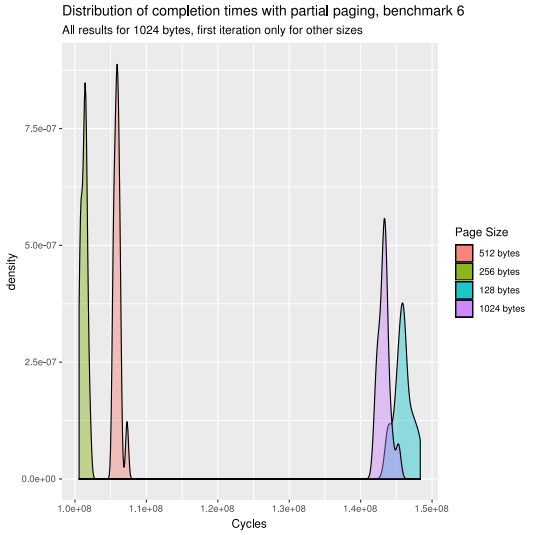

Figure 5.49: Density plots of observed timings for different page sizes with partial paging (benchmark 6)

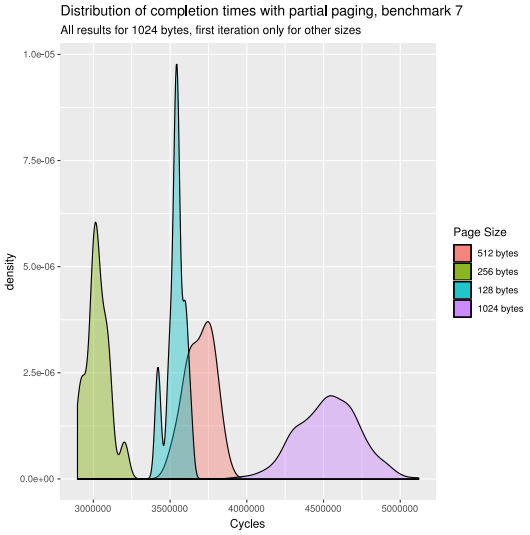

Figure 5.50: Density plots of observed timings for different page sizes with partial paging (benchmark 7)

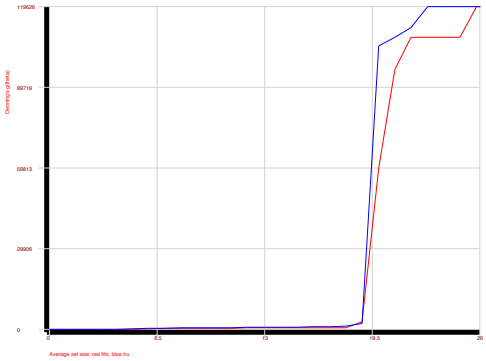

Figure 5.51: FIFO (red) and LRU (blue) lifetime curves compared for benchmark 0 with 512 byte pages

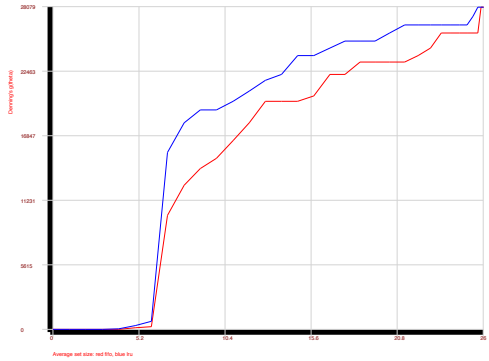

Figure 5.52: FIFO (red) and LRU (blue) lifetime curves compared for benchmark 5 with 512 byte pages

us to consider whether a simpler first-in, first-out (FIFO) replacement policy could be a reasonable alternative.

We would expect policies based on ordering pages by access time, such as LRU (of which CLOCK is an analogue) to outperform FIFO [44] but perhaps not by much for small buffers. Static analysis (see Figures 5.51 and 5.52 for illustrative examples) shows that the life curves (i.e. the mean amount of code executed between faults when a given mean number of pages are cached locally as a working set) for the benchmarks under FIFO are generally lower but often broadly similar to those seen under LRU - further suggesting that while we should expect a higher fault rate with FIFO the difference may not be great. If the simpler page replacement logic means less time is lost to administrative tasks then FIFO could match or surpass CLOCK performance for at least some benchmarks. However, this we might expect this only to apply to the very first run: as CLOCK would (or at least should) preserve in memory those pages that are most likely to be needed (as so require faulting back in if removed) on subsequent runs.

We simulated FIFO with 512-byte pages using partial paging. The mean level of blocking was higher than with CLOCK at 67.26 blocked packets on each cycle (10,853,620,898 blocked packets over 161,375,684 cycles). Figure 5.53 (cf. 5.6) shows the average level of blocking in the system varies through a small range when we smooth the count over 10,000,000 cycles.

Tables 5.16 and 5.17 show the average performance of the FIFO system for each benchmark. Benchmark 1 rises to 100% efficiency after the initial run: with no page replacements required and no administrative costs of an unneeded CLOCK interrupt, the system simply runs through the code. Benchmarks 3 and 7 show a small improvement on subsequent runs but all the other benchmarks show a decline in mean performance - reflecting both the lack of the memory

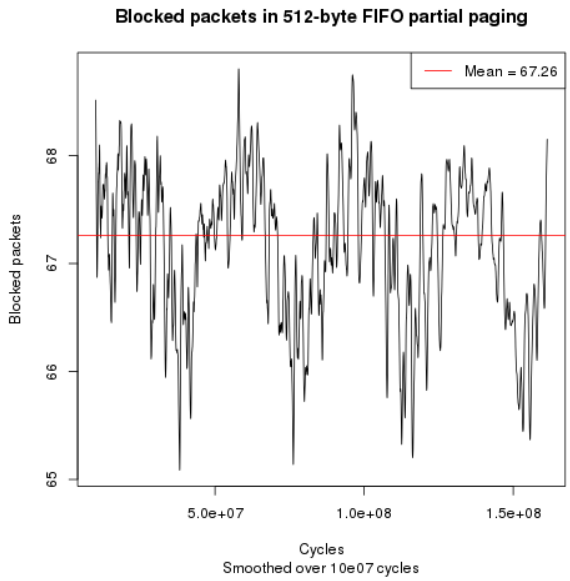

Figure 5.53: Blocking in a partial paging system with 512-byte pages and FIFO page replacement policy - plot is smoothed over 10,000,000 cycles

tuning that CLOCK delivers and the impact of entropy in the system, which we discuss further below (in 5.4.1).

Table **??** compares observed maximum execution times for partial paging with FIFO page replacement to partial paging with CLOCK-based page replacement. The figures suggest FIFO may be a good choice for benchmarks which do not put too much pressure on the memory system - delivering a lower maximum for benchmarks 0, 1 and 4. (It should be noted that with benchmark 1 no pages are ever replaced, so the page replacement code is never called with FIFO). However, in more complex cases it under-performs CLOCK. It also should be noted that, with FIFO, the benefit of CLOCK seen in subsequent iterations (that pages in high demand are held in memory) will not apply unless the program can fit in memory (as with benchmark 1) - and as Figure 5.54 illustrates, the worst cases with FIFO are typically found after the first run of the benchmark. We examine this further in 5.4.1.

5.4.1 The impact of entropy on completion timings under FIFO

As we noted above (5.2.2) with 512 byte pages under partial paging the worst completion times are generally seen with the initial iteration and this pattern is also seen with 256-byte and 128-byte paging systems.

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	2146245	1165088	1123676	4406631	9639727	1367198	156993072	4408034
Mean	2142056	1158976	1121688	4388787	9630301	1355769	156672843	4393091
Service cost	39900	30550	30200	221550	42850	51300	6965750	169550
Blocks	747691	676100	639283	3242145	1191934	863923	125088358	3236443
Count								
Hard Faults	32.0	25.0	26.0	75.0	33.0	37.0	5986.0	135.0
Small Faults (and page table reads)	724.0	586.0	578.0	2036.0	782.0	849.0	97135.0	2214.0
Share								
Service	1.9%	2.6%	2.7%	5.0%	0.4%	3.8%	4.4%	3.9%
Blocks	34.9%	58.3%	57.0%	73.9%	12.4%	63.7%	79.8%	73.7%
Admin	4.3%	6.5%	6.6%	6.1%	1.0%	8.0%	7.1%	5.7%
Efficiency of execution	58.9%	32.5%	33.7%	15.0%	86.1%	24.5%	8.6%	16.8%

Table 5.16: Performance (means) of benchmarks with partial paging and 512 byte pages using FIFO replacement policy: initial execution

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	2639271	376891	1418667	4896644	10213377	2558750		4785509
Mean	2482276	376891	1220420	4611916	10007962	2115834	NA	4410733
Service cost	55781	0	37700	257408	65197	101281		176754
Blocks	1069836	0	729001	3423385	1543679	1564971		3241789
Count								
Hard Faults	29.1	0	26.0	68.5	30.1	34.3		132.0
Small Faults (and page table reads)	734.1	0	598.0	2062.0	801.4	910.6		2279.0
Share								
Service	2.2%	0%	3.1%	5.6%	0.7%	4.8%		4.0%
Blocks	43.1%	0%	59.7%	74.2%	15.4%	74.0%		73.5%
Admin	3.8%	0%	6.2%	5.9%	1.0%	5.5%		5.8%
Efficiency of execution	50.9%	100%	31.0%	14.3%	82.9%	15.7%		16.7%

Table 5.17: Performance (means) of benchmarks with partial paging and 512 byte pages using FIFO replacement policy: continued execution

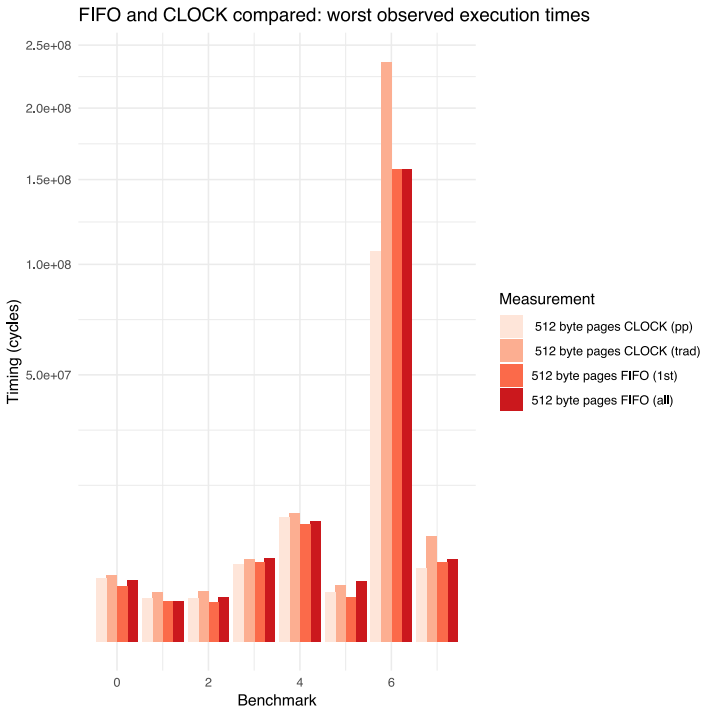

Figure 5.54: FIFO observed worst execution times compared to CLOCK for 512 byte pages

But as Figure 5.54 shows, with FIFO the worst execution times can occur after the first iterations and Figure 5.55 demonstrates[6] that timing uncertainty grows over time with FIFO - it can be seen the timings are tightly packed for initial iterations but then become spread out as the benchmarks interact by blocking each other's memory requests and so inject greater randomness into the timing: in other words it is a product of entropy in the memory connection.

This instability in completion times is despite the regularity in the pattern of FIFO faults - for instance, as Table 5.18 shows, benchmark 5, after the initial iteration, repeats a pattern of hard and small faults with a period of seven iterations, but the range of timings grows, as shown by σ, the standard deviation of completion times. Though the figures also suggest that the system approach some form of equilibrium.

If we consider the two major sources of entropy in the CLOCK-based partial paging systems to be the CLOCK mechanism itself and the blocking and queuing in the connection to memory, then the FIFO system allows us to consider which of these might be the dominant factor. Figures 5.56 - 5.61 show the variation in completion times for several benchmarks using partial-paging FIFO, traditional paging and partial paging with a CLOCK replacement policy -

6 In Figure 5.55 the boxes show the interquartile range (IQR) between the first and third quartile, with the black line as the median, while the whiskers extend 1.5 IQR at either side. We have suppressed the display of outliers to make what are already crowded charts a bit easier to read.

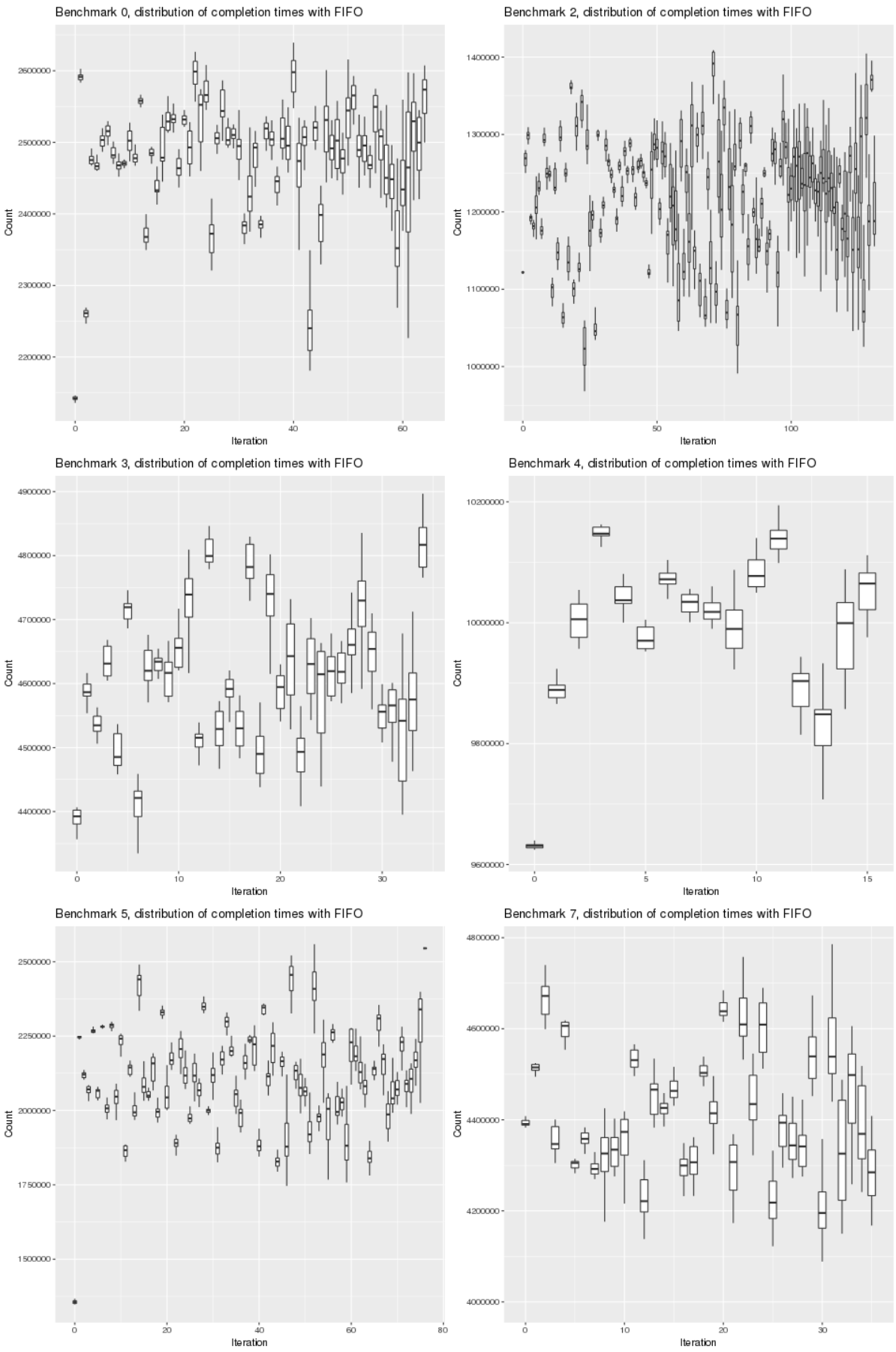

Figure 5.55: Range of performance timings with partial paging and FIFO page replacement

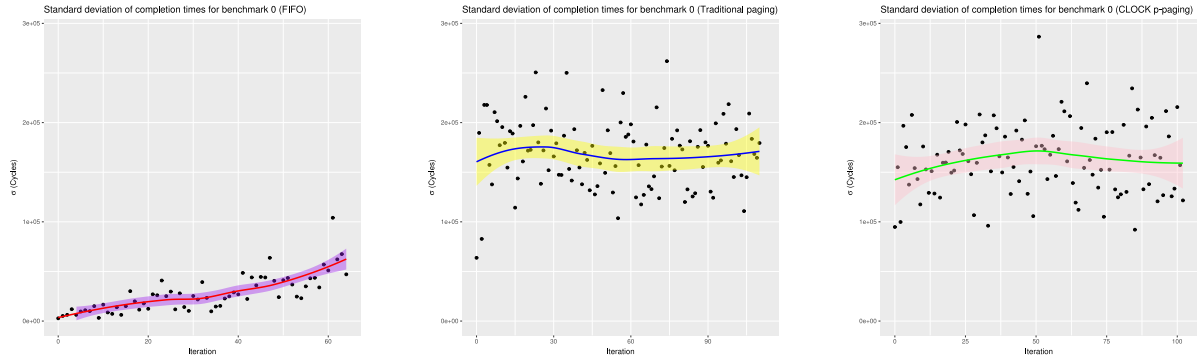

Figure 5.56: Standard deviation (σ) of completion times for benchmark 0, for partial-paged FIFO (left), traditional CLOCK (centre) and partial-paged CLOCK (right)

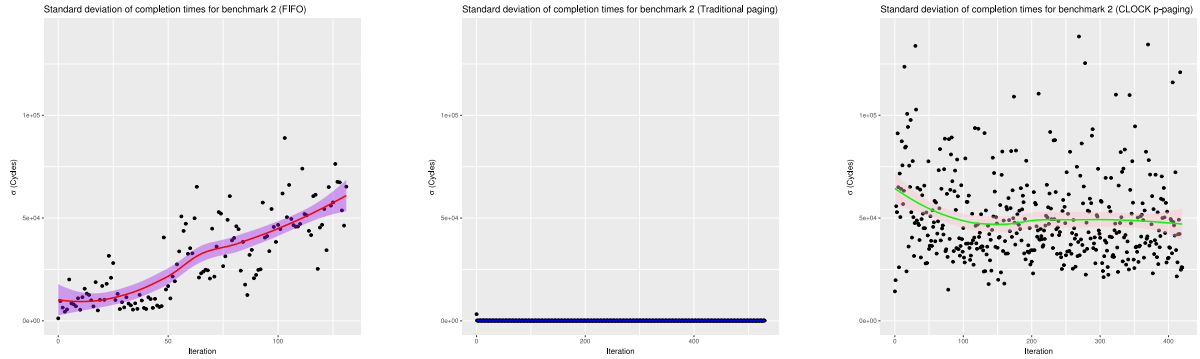

Figure 5.57: Standard deviation (σ) of completion times for benchmark 2, for partial-paged FIFO (left), traditional CLOCK (centre) and partial-paged CLOCK (right)

along with a local regression line. The general pattern is that, for CLOCK-based partial paging, uncertainty starts at a high level but is broadly stable, while for FIFO initial low variation quickly disappears. The pattern varies for different benchmarks (for instance, for benchmark 0 it appears CLOCK is poorly tuned to match the rapid variation in the working set that we know occurs) but overall there is little to suggest that avoiding the uncertainty that comes from using a CLOCK policy delivers greater stability if the price of that is more blocking and queuing in the system. We test this further in 5.4.2 when we consider EVT analysis of the FIFO results.

Figure 5.62 shows the standard deviation of completion times for three of the benchmarks when using CLOCK and 256-byte paging: these results show that instability in timing under FIFO rises towards values which are similar to those seen with the partial-paging CLOCK system.

Iteration	Hard Faults	Small Faults	μ Completion time	σ Completion Time
0	34	905	2246187	3337
1	34	904	2119739	9961
2	35	918	2067552	17146
3	34	905	2268026	7250
4	34	920	2058722	15778
5	35	916	2282312	5916
6	34	906	2007399	18714
7	34	905	2284146	10350
8	34	904	2041751	35884
9	35	918	2227193	34124
10	34	905	1875559	40829
11	34	920	2140236	18925
12	35	916	2003849	29758
13	34	906	2425214	45621
14	34	905	2088478	37584
15	34	904	2053084	37584
16	35	918	2143496	43957
...				
30	35	918	1874172	28347
37	35	918	2235360	30256
44	35	918	2162238	25336

Table 5.18: Cycle in FIFO faults for benchmark 5

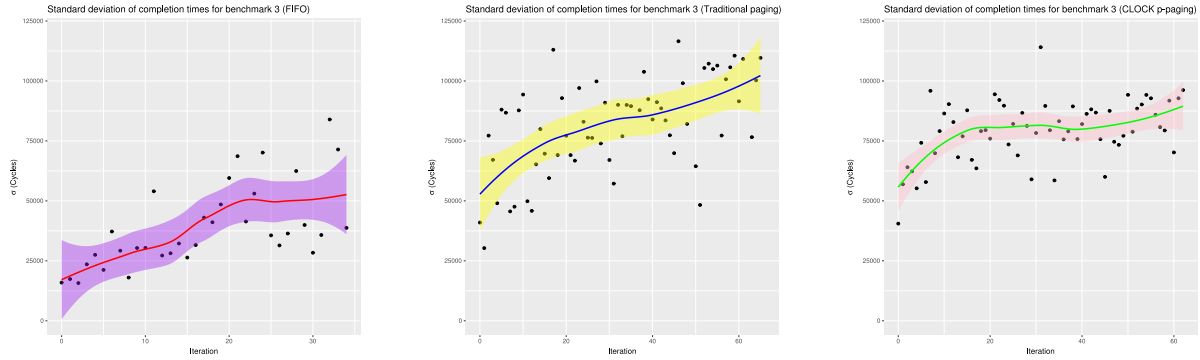

Figure 5.58: Standard deviation (σ) of completion times for benchmark 3, for partial-paged FIFO (left), traditional CLOCK (centre) and partial-paged CLOCK (right)

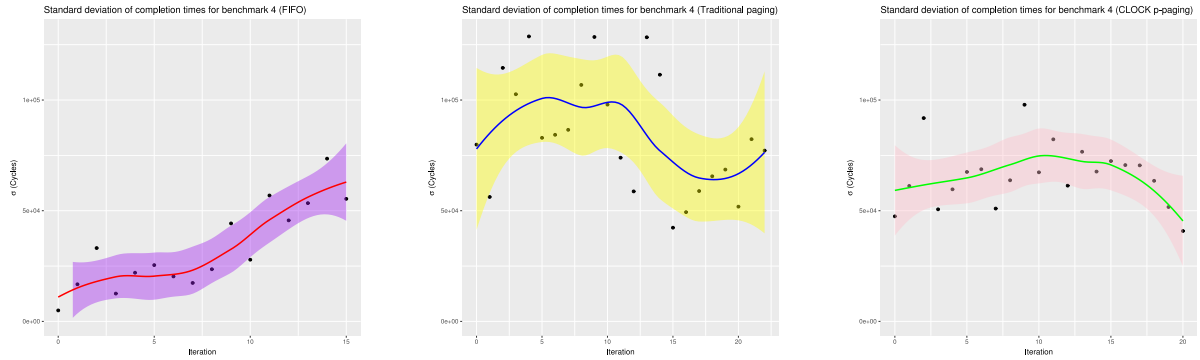

Figure 5.59: Standard deviation (σ) of completion times for benchmark 4, for partial-paged FIFO (left), traditional CLOCK (centre) and partial-paged CLOCK (right)

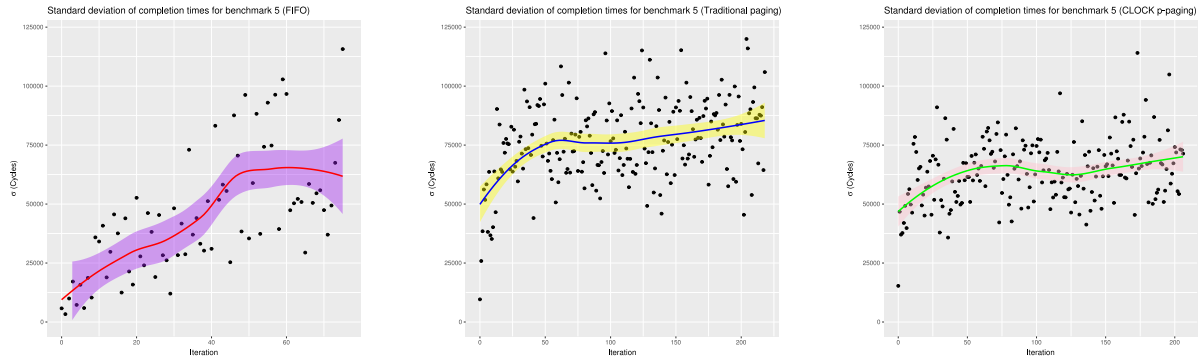

Figure 5.60: Standard deviation (σ) of completion times for benchmark 5, for partial-paged FIFO (left), traditional CLOCK (centre) and partial-paged CLOCK (right)

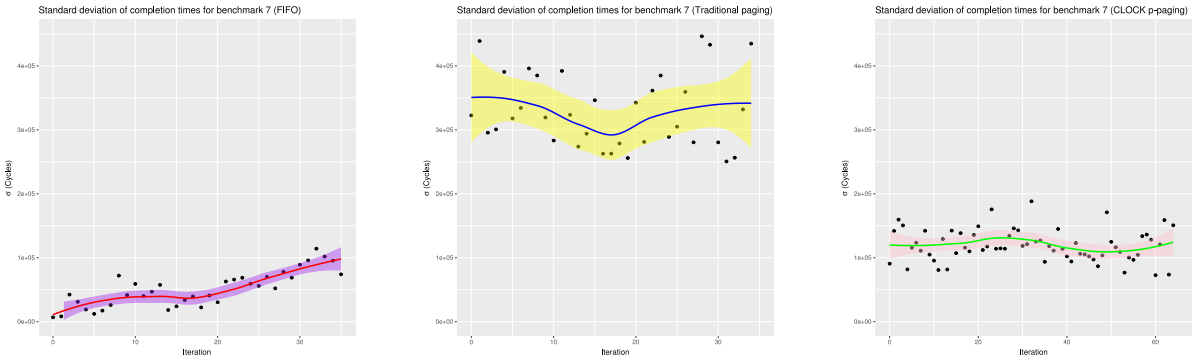

Figure 5.61: Standard deviation (σ) of completion times for benchmark 7, for partial-paged FIFO (left), traditional CLOCK (centre) and partial-paged CLOCK (right)

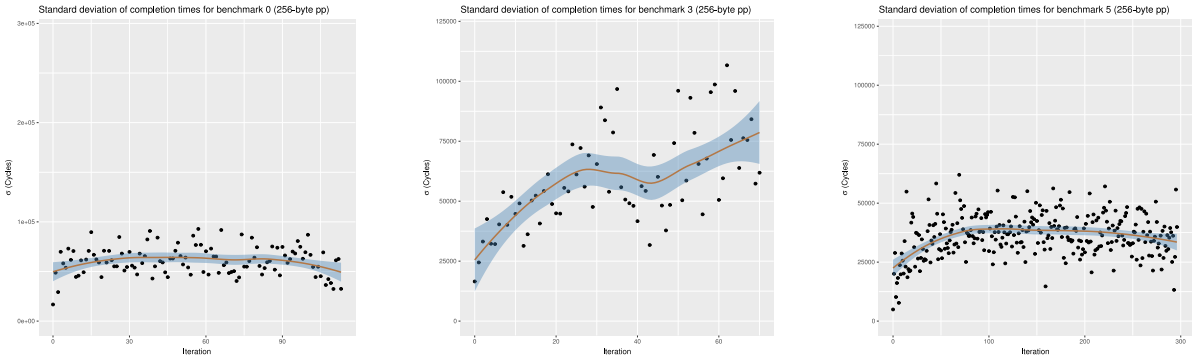

Figure 5.62: Standard deviation (σ) of completion times for benchmarks with 256-byte pages and CLOCK-based page replacement

Benchmark	ADF test p-value
0	<0.01
1	NA
2	<0.01
3	<0.01
4	0.0663
5	<0.01
6	NA
7	<0.01

Table 5.19: Augmented Dickey-Fuller test results for 512-byte FIFO partial paging (alternative hypothesis is stationary values)

Benchmark	p-value for Gumbel	Block size used	Sample size used	p-value for Fréchet	Block size used	Sample size used
0	0.6304	32	32	0.5845	32	30
2	0.07518	48	43	0.05375	48	43
3	0.98022	24	23	0.5856	24	23
5	0.7269	24	50	0.6956	24	50
7	0.3782	24	24	0.3391	24	24

Table 5.20: Goodness of fit tests for Gumbel and Fréchet distributions for partial paging with FIFO 512-byte page size

5.4.2 EVT-based analysis of FIFO timings

We can apply the EVT methods discussed in Section 4.9.2 to the FIFO results. Again, caveats apply - particularly that we do not have the i.d.d. property.

Here we compare, where possible, the pWCETs for the 512-byte page FIFO instances with those for the CLOCK-based partial paging systems considered in Section 5.2.2.

The first question is to what extent we can consider the distributions of completion times as stationary. Results in Table 5.19 show that results exhibit stationarity for all but benchmark 1 (which is fully bi-modal), 4 and benchmark 6 (where insufficient data is available).

	0	2	3	5	7
Observed maximum	2639271	1418667	4896644	2558750	4785509
Computed μ	2539555	1305316	4685243	2237134	4510743
Standard error for computed μ	5932	5932	8389	5932	8389
Computed β	35184	41029	75234	70913	110038
Standard error for computed β	4843	4194	8389	4194	8389
Memo: computed μ for partial paging	2452575	1317823	-	1692572	-
Memo: computed β for partial paging	99198	4285	-	4418	-
Computed percentage for observed maximum	$\approx$ 94.3%	$\approx$ 93.9%	$\approx$ 94.2%	$\approx$ 98.9%	$\approx$ 92.1%
Computed threshold for 10^{-1} maxima (i.e. 0.9 CDF)	2618732	1397646	4854547	2396714	4758369
ditto 10^{-3}	2782580	1588714	5204903	2726948	5270804
ditto 10^{-6}	3025640	1872153	5724639	3216833	6030974
ditto 10^{-9}	3268682	2155571	6244337	3706683	6791090
ditto 10^{-12}	3511726	2438990	6764037	4196534	7551208
ditto 10^{-15}	3754795	2722440	7283793	4686439	8311409
Memo: 10^{-15}threshold for CLOCK (pp)	5878832	1465825	-	1845168	-
Memo: Observed maximum for traditional paging	3109766	1780333	4798261	2262785	3845509
Computed threshold equivalent for FIFO (with pp)	$\approx 9.1 \times 10^{-8}$	$\approx 9.4 \times 10^{-5}$	$\approx$ 80%(CDF)	$\approx$ 50%(CDF)	$\lll$ 0.1%(CDF)

Table 5.21: Estimated Gumbel distributions for Benchmarks 0, 2, 3, 5 and 7 with 512-byte FIFO partial paging compared

Table 5.21 shows the modelled Gumbel distributions of pWCETs for 512-byte partial-paging FIFO for benchmarks 0, 2, 3, 5 and 7. For benchmarks 0, 2 and 5 we also show the available comparisons for CLOCK-based partial paging with the same page size[7].

The results suggest that FIFO appears to be a generally poor choice for a page replacement algorithm, with performance comparable to traditional paging in some cases. With benchmark 0, where we have already noted that our CLOCK algorithm appears poorly tuned to handle the large number of page replacements required, FIFO does well. In every other case, though, it gives a poor performance which appears to reflect a failure to hold required pages in memory (so increasing the fault rate and delays caused by blocking) - leading to a high value of μ and that blocking then adding to entropy (and hence extending the timing range) - as shown by a high β.

Figures J.1 - J.5 in Appendix J on page 243 plot the quality of the fit to the Gumbel distribution and the Q-Q plots suggest we may be over-estimating the length of the tail: a much larger collection of data is undoubtedly required to give more robust answers but the general pattern seems clear.

7 For the "observed maximum" here we have used the maximum seen on the first iteration of the runs tabulated in Tables 5.1 and 5.2 rather than the results of the tests for EVT.

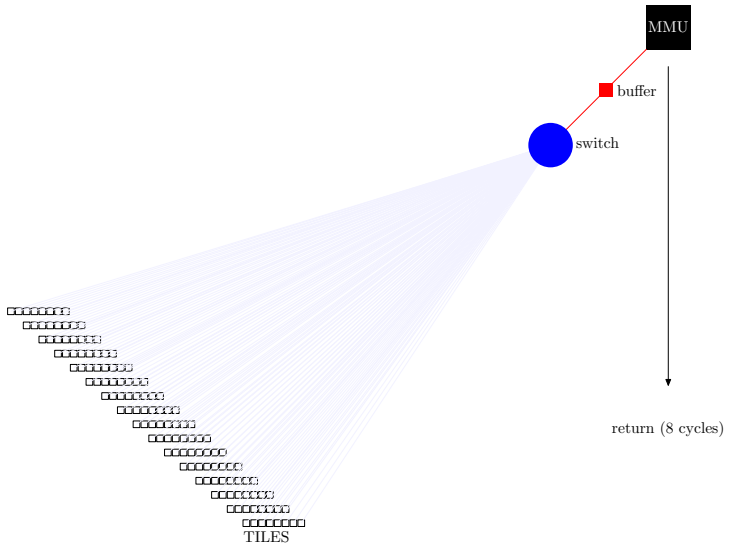

Figure 5.63: Block model of a simple crossbar/switch connection

5.5 ALTERNATIVE CONNECTIONS TO MEMORY

The advantage of the tree-like design, certainly for realtime uses, is that all routes from the processors to the MMU are of equal length and CPU to CPU traffic cannot block memory requests [65]. But, as discussed in Section 4.2.4, there are alternatives to the tree that preserve predictability and maintain the general fairness characteristic we identify as important in 4.2.5. We test both a crossbar-based and a multiple-bus-based approach in this section.

As with the tree-based approach these methods aim to preserve a constant distance between the memory-requesting cores and external memory and operate a fair (and generally stochastic) arbitration process for requests. Access time to external memory is governed by a probability distribution that is independent of the location (or number) of the requesting processor and is the same for all processors.

5.5.1 A crossbar memory connect

A crossbar (switch) mechanism, allowing all processors to communicate directly a (one-packet) buffer connected to the MMU maintains equidistance between cores and memory. In this arrangement one packet can be buffered waiting for the MMU to become free (with a maximum of four memory requests being serviced simultaneously by the MMU). Figure 5.63 presents a simple block model of this arrangement.

Here processors request access to the buffer. If the buffer is empty then the packet is immediately accepted and the buffer marked as full. A packet has to wait at least one cycle

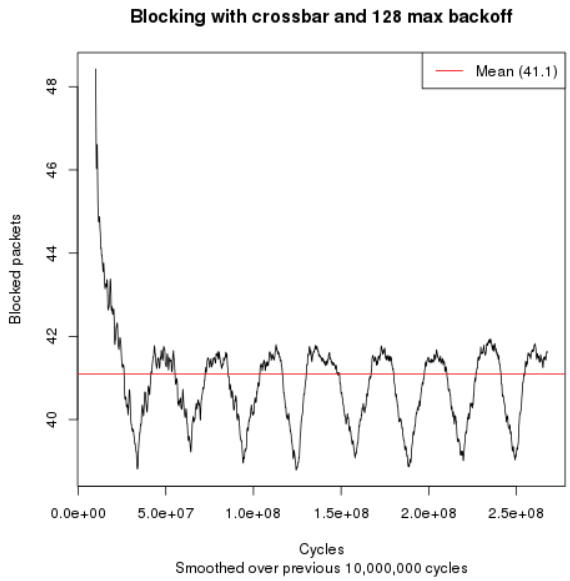

Figure 5.64: Blocked packets with simulated crossbar connect for 512-byte partial paging

in the buffer but a packet can move from the buffer into the MMU and a new packet can be accepted in the buffer in the same cycle. As before the MMU can process four packets simultaneously and read requests are served in 50 cycles and write requests in 100 cycles (each request is 16 bytes long). On being released by the MMU a packet is returned to the requesting processor in 8 cycles. We did not model a return mechanism, though conceivably this could be a delay-free tree.

If there is a packet buffered and waiting for service then a requesting processor makes repeated requests after a backoff period which varies from 2^1 to 2^7 (i.e. from 2 to 128) cycles (if this has failed after 2^7 cycles the backoff reverts to 2^1), with the backoff increasing by a power of two each time.

This system (with 512-byte pages and using partial paging) has an average blocking count of 41.1 (10,998,635,238 blocked packets over 267,598,998 cycles) compared to 45.2 for the tree-based system. Figure 5.64 shows a smoothed plot of the number of blocked packets and this looks much like Figure 5.6.

Tables 5.22 and 5.23 show the broad performance of the different benchmarks under this scheme, again these are not hugely different from the tree-based results (as shown in Tables 5.1 and 5.2). It seems that those benchmarks which make heavy calls on memory (3 and 6) are a little slower while the others are slightly faster. It is noticeable, though that the gap between maximum and average times seems smaller and we consider an entropy model of the system below.

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	2485800	1343184	1332485	4259658	10619277	1622117	125430329	3882840
Mean	2422466	1316339	1295115	4190638	10528907	1594327	124451968	3604490
Service cost	43328	30550	30444	233756	41816	73100	6155759	154378
Blocks	736239	768815	746804	2941809	604814	1034550	92752773	2331183
Count								
Hard Faults	31.6	25.0	26.0	66.1	32.3	34.0	5188.3	126.1
Small Faults (and page table reads)	748.5	586.0	581.3	1995.7	782.1	852.0	85936.4	2080.4
Share								
Service	1.8%	2.3%	2.4%	5.6%	0.4%	4.6%	4.9%	4.3%
Blocks	30.4%	58.4%	57.7%	70.2%	5.7%	64.9%	74.5%	64.7%
Admin	15.7%	10.6%	10.8%	8.5%	15.1%	9.7%	9.7%	10.6%
Efficiency of execution	52.1%	28.6%	29.2%	15.7%	78.8%	20.8%	10.8%	20.4%

Table 5.22: Performance (means) of benchmarks with partial paging and 512 byte pages with crossbar connection to MMU: initial execution

Benchmark	0	1	2	3	4	5	6	7
Memo: minimum instruction time	1262292	376891	378280	658567	8295863	332299	13437582	736058
Ticks								
Maximum	2344953	442278	621986	3933637	9945240	1039688	125746015	3643035
Mean	1716808	441564	459934	3665154	9855577	908597	124735079	3193231
Service cost	31653	0	3242	223419	19107	52399	6160144	159066
Blocks	91206	0	6305	2460394	35033	421786	93031267	1919892
Count								
Hard Faults	17.6	0	3.1	44.5	10.2	11.9	5177.2	118.5
Small Faults (and page table reads)	429.9	0	56.6	1684.5	281.7	402.8	85881.4	2075.7
Share								
Service	1.8%	0%	0.7%	6.1%	0.2%	5.8%	4.9%	5.0%
Blocks	5.3%	0%	1.4%	67.1%	0.4%	46.4%	74.6%	60.1%
Admin	19.3%	14.6%	15.7%	8.8%	15.3%	11.2%	9.7%	11.8%
Efficiency of execution	73.5%	85.4%	82.2%	18.0%	84.2%	36.6%	10.8%	23.1%

Table 5.23: Performance (means) of benchmarks with partial paging and 512 byte pages with crossbar connection to MMU: continued execution

Attempt	Total Wait	Expected wait (partial)	Probability	Entropy contribution	Total entropy
1	0	0	$\frac{1}{37}$	0.14	0.14
2	2	0.05	$\frac{36}{37} \times \frac{1}{37}$	0.14	0.28
3	6	0.21	$(1 - (\frac{36}{37} \times \frac{1}{37})) \times \frac{1}{37}$	0.14	0.41
...	...		...	...	...
9	256	16.86	0.0217	0.12	1.17
10	260	22.35	0.0211	0.12	1.29
...	...		...	...	...
100	3558	919.89	0.00179	0.016	5.95

Table 5.24: Simplified model of entropy for crossbar system

5.5.1.1 *An entropy model of the crossbar system*

We can see that the system delivers similar average (mean) timings as the tree-based system and to consider pWCETs we can model the entropy of the system. For benchmark 1, which has no memory write-backs, the average blocking per packet is 1256 cycles (in comparison to 1262 cycles for 512-byte partial paging with CLOCK, and 1107 cycles with FIFO using partial paging). The average blocking over the first 1,316,339 cycles is 73.2 blocked packets. If we assume that (at this point) 90% of all packets are read-only requests then the average time to serve a packet will be 55 cycles and to maintain a constant queue length a new packet has to enter the system every 13.75 cycles on average. We will round this up and assume that it means a packet sits in the buffer for 14 cycles before being serviced and so the time blocked waiting to get to the buffer becomes 1232 cycles.

We can model the acceptance process as a Markov process (i.e. memory-less in that each request has a fixed probability of succeeding) and by trial and error find a probably for packet acceptance that gives an expected waiting time that matches the observed time. In this case a probability of 1 in 37 gives a good fit, converging towards 1233.67 cycles which is close to our observed value.

Some values for timing expectation and packet entropy generated using this approach are shown in Table 5.24. Using these values the entropy converges towards 6.63 per packet. With 73.2 blocked processors this gives a total entropy of 485, substantially lower than the 668 our model would predict for a tree-based system. This is despite the backoff system being unbounded in a way the tree-based system is not.

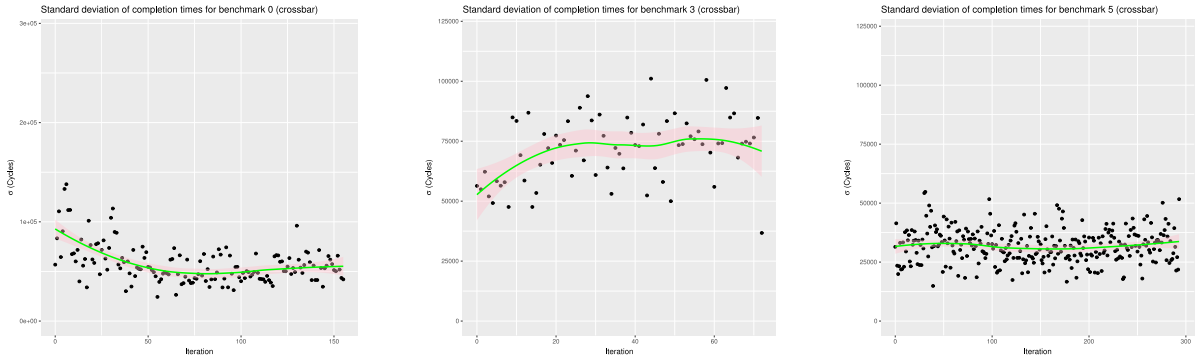

Figure 5.65: Standard deviation (σ) of completion times for benchmarks with 512-byte pages and CLOCK-based page replacement using crossbar memory connect

Benchmark	0	1	2	3	4	5	6	7
Crossbar	2485800	1343184	1332485	4259658	10619277	1622117	125746015	3882840
Tree	2871432	1345670	1349593	4198788	10961440	1712878	107296451	3845509

Table 5.25: Maximum observed execution times (cycles) compared for 512 byte pages with crossbar and memory tree with CLOCK and partial paging

The lower entropy in the system might still allow it to deliver better performance at extreme and safety-critical limits despite under-performing compared to the tree-based system for some benchmarks. We did not record data here in a way that would allow us to use EVT analysis (the results are bimodal again), but Figure 5.65 does suggest that the variation in completion times is indeed somewhat lower than with the tree-based design (cf. Figure 5.56 for benchmark 0, for instance). The gap between the performances for benchmark 6 for the two systems remains large however.

We did not model the use of a crossbar/switch with traditional paging (below in 5.5.3, though, we do consider traditional paging with a bus-based arrangement which generally performs similarly to this crossbar model). However it seems reasonable to expect that using a traditional paging approach will increase the amount of blocking in the system and thus increase the length and range of waiting times, so increasing entropy in the system.

5.5.1.2 *Changing the back off range*

The system is sensitive to the backoff ranges chosen. If we change the maximum backoff range to 2^9 cycles then blocking rises: if the average number of attempts to gain access to the buffer is substantially greater than the number of elements in the backoff cycle then this is almost certainly going to happen, even if the additional waiting time creates opportunities for other processors to grab control of the buffer.

Benchmark	0	1	2	3	4	5	6	7
Max backoff of 2^9 cycles	2722913	1420524	1389744	4960962	10735543	1886881	134563137	4151912
Max backoff of 2^7 cycles	2485800	1343184	1332485	4259658	10619277	1622117	125746015	3882840

Table 5.26: Maximum observed execution times (cycles) compared for 512 byte pages wi7th crossbar using different maximum back off times

Switching to this backoff regime increases mean blocking to 44.7 blocked packets (12,472,693,515 blocks over 278,989,309 cycles) , while Table 5.26 shows how this is manifested in the observed worst execution times.

Considering the requests and waiting time, in this case benchmark 1 has an average blocking count of 1366 blocked packets per fault and again we assume that 14 of these are spent waiting in the buffer. By fitting the wait time to a Markov chain model we approximate the success probability per request, with a modelled wait time of 1355 cycles, as $\approx \frac{10}{157}$ - in other words we should expect that a packet would require fewer requests to be accepted: so increasing the backoff range cuts the projected mean number of requests needed to succeed by more than half but slows down overall performance.

In our entropy model per-packet entropy falls to 5.37, while the mean blocking over the 1,381,891 cycles that benchmark 1 takes to complete (on average) in this regime rises to 77.3, giving a projected total entropy of 415, lower than for the smaller-range backoff system. However Figure 5.66 (cf. Figure 5.65) does not support this conjecture of lower entropy: in fact suggesting that disorder in completion times increases somewhat.

We are forced to conclude the entropy model is flawed in that it relies on targeting a fixed blocking count - it will be the case that typically the larger backoff system requires fewer requests to succeed, but as we are considering the range of values when we look at entropy we must consider the distribution of blocking (over time) and not just its mean value. The marginal cost of additional requests in the larger-backoff system is likely to be higher, perhaps substantially so, and so that system will display a larger range of completion times.

In a real world system a decision on what range of backoff timings to use may be dependent on various factors: for instance a higher maximum backoff might lead to lower power consumption if waiting processors consume less energy. But these results do suggest that backoff tuning has performance and predictability implications.

5.5.2 Multi-layered bus-based connectors

Large crossbars and switches are generally avoided because of their energy requirements and electronic complexity, so we consider bus-based alternatives.

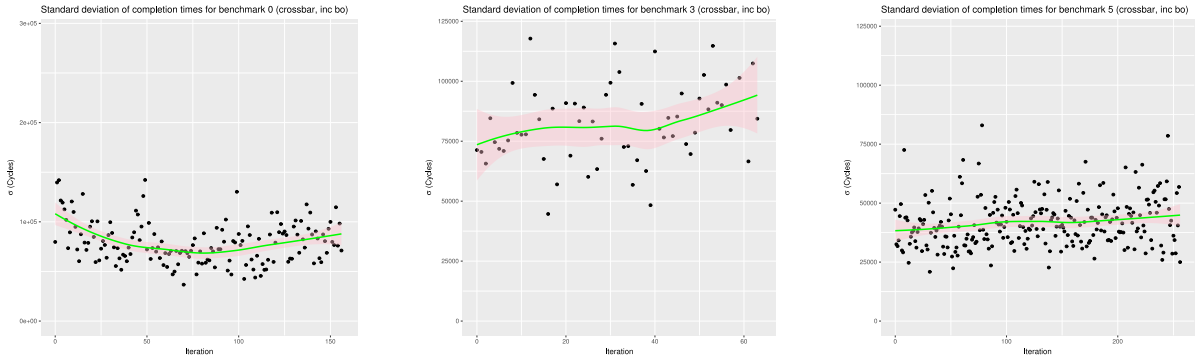

Figure 5.66: Standard deviation (σ) of completion times for benchmarks with 512-byte pages and CLOCK-based page replacement using crossbar memory connect with maximum backoff of 2^9cycles

As noted above (Section 2.3), buses are generally not an effective means of connecting large numbers of processors: signals attenuate and can be slow to propagate and the method is generally inefficient, though are generally relatively simple to implement. Here we tested a simulation of a multi-layered bus arrangement rather than modelled a single bus connecting all cores.

Each processor core was connected to a bus that was also connected to 15 other cores and which could buffer a single packet. We reason that without some sort of buffering capacity no such benchmark arrangement would be likely to be feasible for a large number of processors. These eight buses were then connected to a single bus which is in effect at the bottom of a chain of six further buses (or chained buffers), each of which can buffer a single packet (so the total number of packets that can be buffered in this arrangement is 8 + 7 = 15 (in contrast in the tree-based connect there are more buffers than processors).

Figure 5.67 shows the essentials of the arrangement being tested (though shows the minimal buffering version of this experiment described below): a group of 16 processors is chained to a single bus and then these 8 buses are themselves connected by a single bus.

Only one processor can master each of the buses at the bottom level and access is again governed by a backoff algorithm: with each processor cycling through a delay of 2^0 - 2^7 cycles if an initial attempt to master the bus fails. This bottom layer of buses then attempts to master the next point in the chain, again applying a backoff, though the cycle here is between 2^0 and 2^3 cycles. Above this point packets can move forward every cycle should the chained buffer in front become empty. The minimum journey time between the core and the MMU thus remains 8 cycles, as with the tree.

We again assume an 8 cycle return time to the requesting processor when the request is completed.

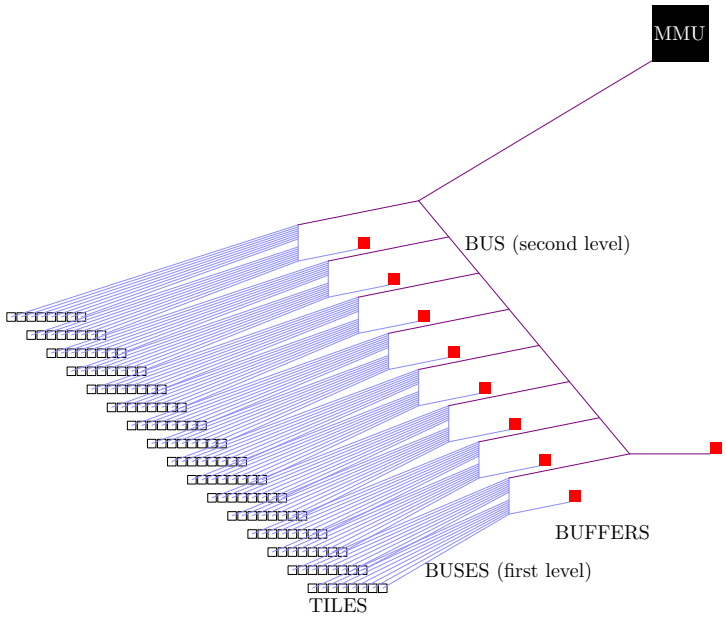

Figure 5.67: Simplified representation of multi-layered bus connection

	0	1	2	3	4	5	6	7
Same benchmark	2611484	1340816	1343941	4181388	10599044	1614364	123106007	4040913
Range of benchmarks	2809824	1335642	1295518	4250247	10845018	1724218	108333976	3916552

Table 5.27: Observed maximum execution times (cycles) compared for 512 byte pages with different bus arrangements

We attached the processors to the bottom of the chain of buses in two different ways - either with every processor running the same benchmark attached to the same bus or each bus being attached to 2 rows of processors where each processor in a row was running a different benchmark.

For the case where the benchmarks are not grouped together the average blocking was 45.0 (5,600,624,321 blocks over 124,440,980 cycles). When grouped the blocking is lower at 41.3 (5,194,104,541 blocks over 125,881,844 cycles). The maximum observed completion times for both arrangements are shown in Table 5.27.

The results suggest that where benchmarks which make heavy demands of the memory system (particularly benchmark 6) compete against only against the same code they are slowed, while when all the benchmarks compete against one another, the highly demanding benchmarks benefit at the cost of the others.

REWORKING THE BUS DESIGN TO MINIMISE BUFFERING AND LAYERING We reconfigure the system so to use just two bus layers and so only be capable of buffering 9 packets and so

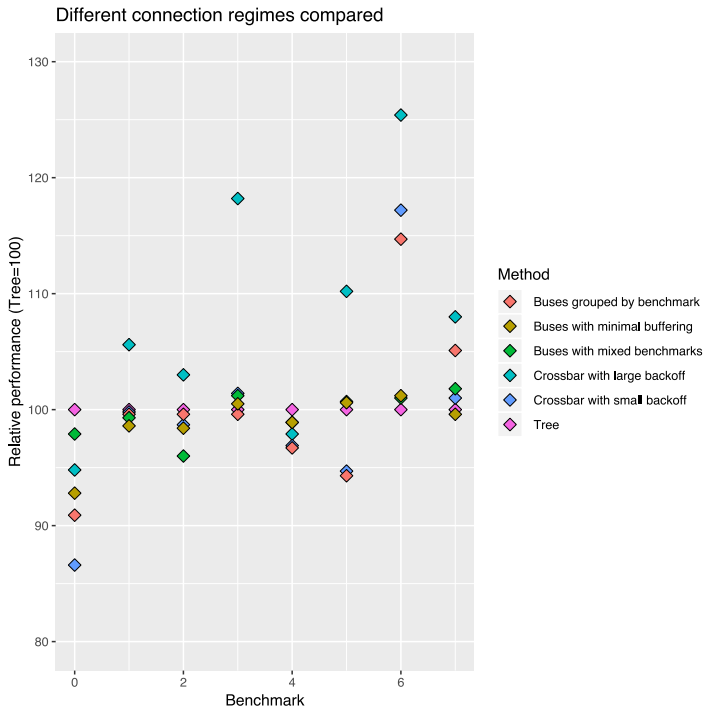

Figure 5.68: Indexed observed maximum execution times for different connection arrangements (tree = 100.0)

also reducing the minimal bus traversal time to 2 cycles, (the return time was also reduced to 2 cycles). This does not result in a uniform change in benchmark performance. Table 5.28 (top row) shows the observed maximum execution times (when buses are attached to processors running a range of benchmarks). Blocking rises to 45.9 (5,177,101,890 blocks over 112,910,216 cycles) reflecting the loss of buffering capacity (and so forcing processors to spend longer in a backoff cycle as they compete for mastery of the buses).

Once again (cf. Table 5.27) the performance results show a similar pattern to the tree (and thus the crossbar) and this suggests that the bigger trade-off here is between how the different benchmarks are grouped, rather than the choice between the various connection patterns tested. When other benchmarks compete directly against the benchmarks with the highest fault rates they tend to take longer and vice versa, but the differences are often not great for everything except benchmark 6 (see Figure 5.68).

5.5.3 Traditional paging with a bus connection

To complete the picture we consider the behaviour of a bus-based system using traditional paging. In this case we use a bus system with minimal buffering (as described in 5.5.2 above).

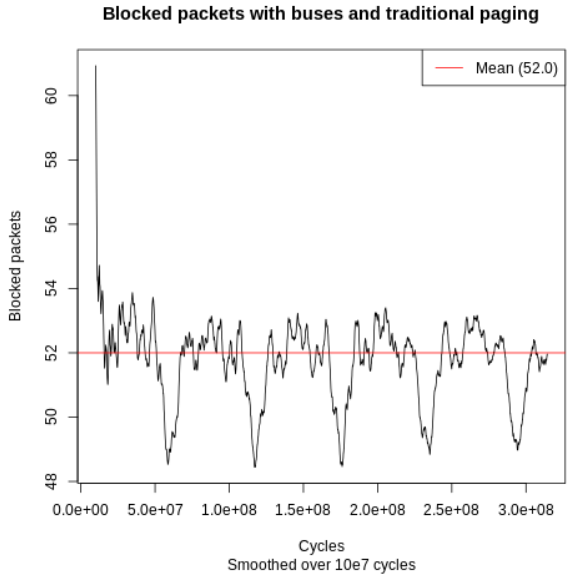

Figure 5.69: Blocks for 512 byte page sized traditional paging using interlinked buses (smoothed)

	0	1	2	3	4	5	6	7
Partial paging	2665386	1326565	1327555	4221647	10846057	1722333	108569056	3829709
Traditional paging	3217417	1768072	1825805	4958499	11486664	2244862	236293254	7992146

Table 5.28: Observed maximum execution times (cycles) compared for 512 byte pages with different bus arrangements

As Figure 5.68 suggests, across all the benchmarks this model most closely matches the performance of the tree-based connect and so offers a good basis for comparison with our wider set of results. The mean blocking is 52.0 (16,343,646,381 packets over 314,300,325 cycles) - somewhat higher than the partial paging system with the same bus arrangement (Figure 5.69).

A higher blocking count indicates higher entropy and so we see a greater range of completion times (eg., Figures 5.70 - 5.72) as well as higher maximum observed completion times (cf. Table 5.28). In fact the maximum observed timings for this system are very similar to those seen for traditional paging when using the Bluetree-like connection (cf. Table 5.3).

5.5.4 An entropy model for buses

Multiple factors will determine the level of entropy in the different bus-based systems. Multiple buffers will work to increase entropy as each adds an element of uncertainty to the length

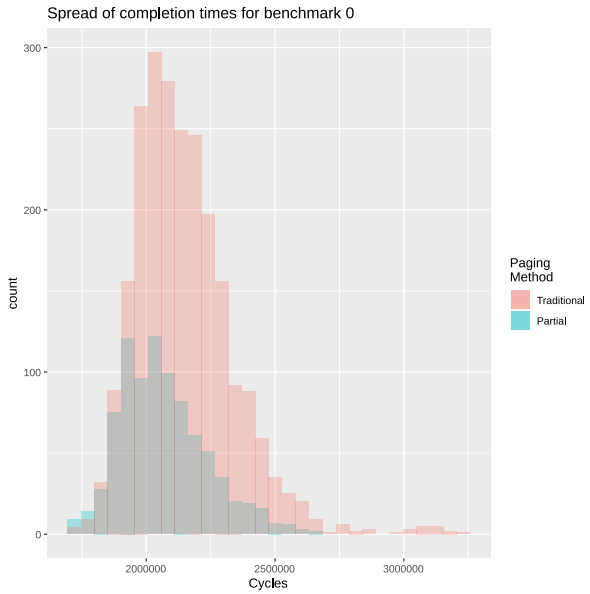

Figure 5.70: Spread of completion times for benchmark 0 compared for traditional and partial paging using buses.

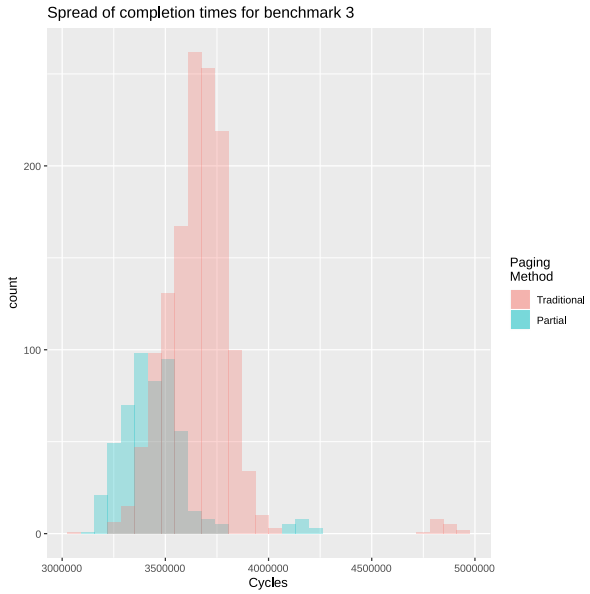

Figure 5.71: Spread of completion times for benchmark 3 compared for traditional and partial paging using buses.

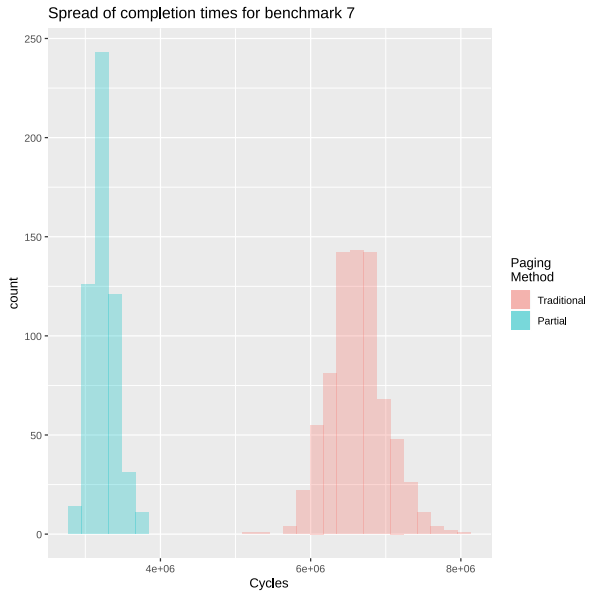

Figure 5.72: Spread of completion times for benchmark 7 compared for traditional and partial paging using buses.

of time a packet will wait to be serviced. In contradistinction the additional blocking seen with the minimally buffered system shows that packets are typically spending longer in the backoff procedure and so exposed to a wider range of delays.

Figure 5.73 shows the spread (as standard deviation) of completion times for benchmarks 0, 3 and 5 for the grouped bus system, the ungrouped bus system and for the minimally-buffered system. (cf. Figure 5.65 and Figure 5.66). These charts strongly suggest that the grouped system delivers the lowest entropy.

With traditional paging the higher level of blocking increases entropy and, as illustrated in Figures 5.70 - 5.72, increases the range of (as well as the mean for) completion times.

5.6 SUMMARY

We find (Section 5.1) that 512-byte pages perform better than 1024-byte pages and similarly 256-byte pages generally perform better than 512-byte pages (Section 5.3). Despite increasing the fault rate for most benchmarks the smaller page sizes can lower fragmentation for some benchmarks and in a system where all processors share the connection to main memory, this reduction in traffic lowers blocking and, by reducing system entropy, also improves worst case execution times.

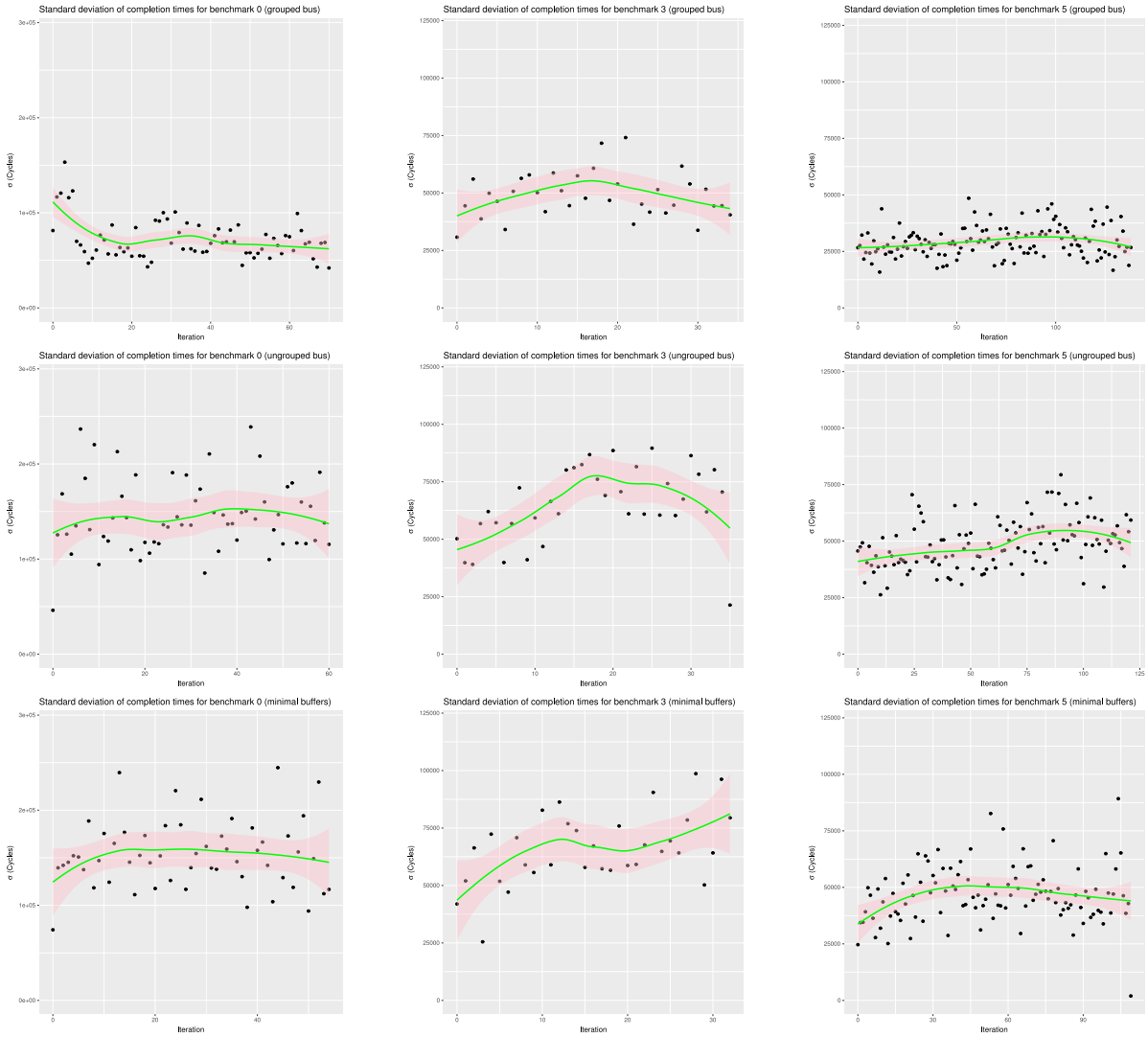

Figure 5.73: Standard deviation (σ) of completion times for benchmarks 0, 3 and 5 with 512-byte pages and CLOCK-based page replacement using grouped buses with multiple buffers (top row), ungrouped buses with multiple buffers (middle row), and ungrouped buses with minimal buffering (bottom row)

For 128-byte pages, though, the process goes into reverse, with a rising fault rate and the loss of free memory space to larger page tables causing execution times to rise (Section 5.3.4).

We find that FIFO, as a simpler alternative to CLOCK-based page replacement, while often given better typical-case performance (Section 5.4) is likely to be a poor choice for safety-critical systems as its high level of sustained blocking extends timing limits at such critical thresholds (Section 5.4.2). FIFO is also subject to growing entropy as execution proceeds (Section 5.4.1).

Alternatives to the tree-based memory connect based on either a crossbar switch (Section 5.5.1) or a network of buses (Section 5.5.2) are shown to give similar timing results to the tree-based system, though examining system entropy and uncertainty shows there are multiple factors to be considered (Sections 5.5.1.1 and 5.5.4). Using a traditional paging approach with the bus-based alternative connection generates very similar results to using traditional paging with the tree-based memory connect: with higher blocking, lower performance and greater uncertainty in completion times than partial paging (Section 5.5.3).

These results show that partial paging is likely to have a general applicability and is not dependent on the connection medium used. By reducing traffic to the MMU partial paging lowers blocking and queue lengths for memory service and in systems with a large number of cores, restricted local memory and a memory connection bottleneck limiting such traffic may be an effective or the most effective way of both improving mean and worst-case performance. Picking an effective and well-tuned page replacement mechanism will also be essential to deliver the benefits that partial paging could offer.

6 CONCLUSIONS AND FUTURE WORK

In Section 1.4 we set ourselves the task of proving a research hypothesis:

Paged virtual memory can be implemented in an effective manner for a many-core real-time system in a way that minimises the worst case execution times for real-time tasks: not through the traditional means of a better page replacement algorithm, but through a more efficient and real-time-appropriate 'partial page' loading algorithm. Furthermore, this partial paging approach is most effective with smaller page sizes and, by limiting queuing for memory service, partial paging reduces entropy in the system and thus increases certainty about timing.

In this thesis we have firstly described the observed behaviour of programs in execution, running on many-core systems with limited local memory, that we believe indicates that a better page loading algorithm desirable as programmers seek to handle larger and more complex workloads. We have then set out what we believe to be such an approach in the form of a workable partial paging virtual memory algorithm. We then report the results of tests in software - no hardware implementation being available - of a simulated many-core NoC system using this partial paging approach while running benchmarks designed to test embedded real-time systems. We have compared partial paging's performance to a traditional whole-paging approach, suggested a simple load control mechanism and developed an entropy model for the system and considered probabilistic worst case execution times.

Subsequently we have considered ways in which a partial paging system could be optimised, examining smaller page sizes and using a simpler (FIFO-based) page replacement policy.

This thesis provides valuable signposts to those seeking to implement virtual memory in many-core systems, particularly those using network-on-chip designs. We demonstrate that, despite severe resource constraints, partial paging offers a mechanism to improve performance and to provide predictable worst-case execution times at extreme or safety-critical limits. Further we develop a model of system entropy that helps explain the factors that shape the variability of WCETs and so improve understanding of the impact of the design choices that hardware implementors will face.

6.1 CONTRIBUTIONS

In Chapter 4 we propose a better page loading algorithm via partial paging and demonstrate results that show, if this could be implemented in hardware, it has the potential to allow a paged virtual memory system to offer a significant improvement in both typical execution timings and worst case execution timings compared to a traditional whole-page-loading paging system. In this chapter we also discuss how a simple load control mechanism might be implemented and develop a model of system entropy that offers an insight into uncertainty in execution timings.

In Chapter 5 we show how smaller page sizes are likely to deliver both better average performance and, by limiting system entropy, better worst case performance. We show, though, that there are limits to how small pages can go before the page tables needed to manage them become unwieldy. We present evidence that whilst a simple page replacement policy in the form of FIFO may match or better the performance of an LRU-analogue such as CLOCK in the typical case, at extreme, safety critical, limits it may show significantly poorer performance compared to a well-tuned CLOCK system. We also show how FIFO is subject to increased timing uncertainty over multiple executions. Finally we show that our results are likely to be generally applicable to any system that seeks to buffer and arbitrate memory requests.

These contributions show it is possible, with appropriate hardware, to construct a system that delivers effective and efficient use of a many-core NoC system with predictable timing bounds, answering the need to provide embedded real-time computing services for larger and more complex computing tasks, so proving the hypothesis.

6.2 CONCLUSIONS

Our first major conclusion is that many-core NoCs can be developed as viable devices to handle a large number of complex concurrent computing tasks. Typically partial paging is 50 - 100% faster than traditional paging for average execution times for demanding code, and our results show that lowering blocking in the system does not just improve average execution times but limits the range of execution times and so delivered better probabilistic worst case execution times.

We would point out that these are conclusions about asynchronous concurrent tasks rather than the *parallelism* defined in [80] as "harnessing multiple processors to work on a single task" and requiring synchronisation across processors or cores. The results outlined in Appendix

A suggest that in such systems the memory gap effect overwhelms the advantage of having large numbers of cores.

Our work does show there are many parameters that can be tuned to deliver the best performance. In software these include the page size, the page replacement policy to be used (and subsequently any parameters for the chosen policy, such as the rate at which a CLOCK policy marks pages for replacement and how many pages are marked).

Many of these choices look like options inside a zero-sum game. A further option - whether to implement a simple load control mechanism - illustrates the point: slowing down the system lowers the speed of most benchmarks but because it also reduces traffic rates in the memory connect it speeds up the slowest (and most demanding) benchmark.

Designers and implementors of partial paging hardware and systems will need to consider how best to tune their system for the task in hand.

6.3 FURTHER WORK

Ou work in this thesis suggests several areas for further research.

HARDWARE FOR PARTIAL PAGING For partial paging to be viable it needs to be supported in hardware. Our mechanism relies on CPUs trapping address references in parallel to TLB lookups. It then requires a high-speed lookup of a bitmap. The second of these could be done in software but only slowly. Projects such as RISC-V [11] make it feasible to consider building such hardware as a research project without substantial commercial risk.

POWER MANAGEMENT: DIM AND DARK SILICON In 2.8 we mention the issue of "dark silicon" and the likely need to actively manage power on a NoC device to avoid overheating, but we have not reported on any experiments here. In fact we did consider one alternative to dark silicon, so-called "dim silicon". In [83] the authors suggest that ideally future generations of chips could be operated at 0.5× nominal voltage for the best results as "dim" silicon, whilst in [153] the author suggests a ratio of 5:8 between power savings and frequency fall for such near-threshold operations: pointing to a 4×saving on power and a 6.4× drop in the frequency of operation.

Here we looked to operate the overall system at (approximately) 50% of maximum design power. In doing so we estimate that the on-chip networks and routers, including the tree-based memory connect use 30% of total system power: basing this assumption on the data collated

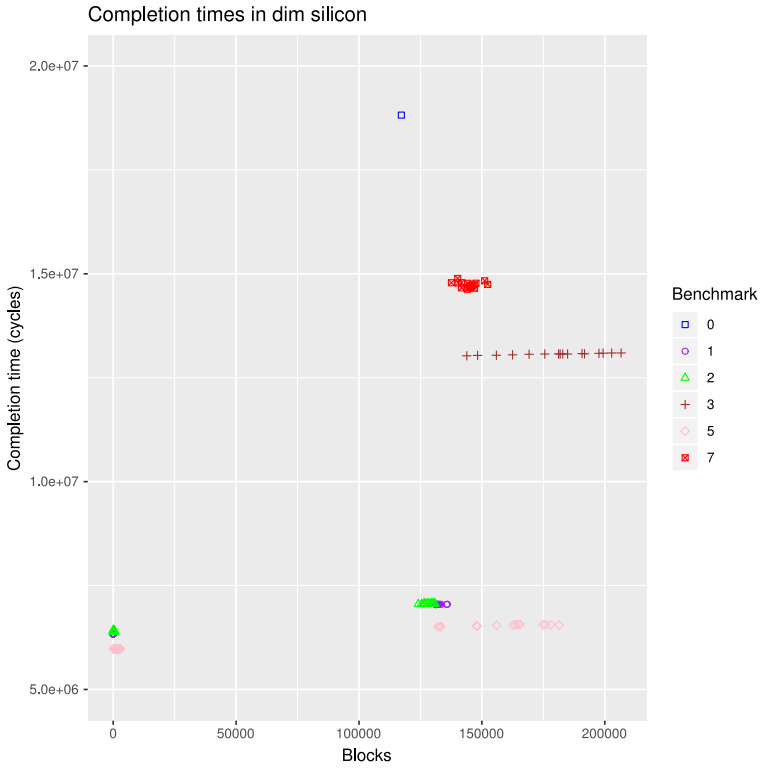

Figure 6.1: Completion times and blocks with dim silicon with 512-byte partial paging

in [141] which mentions interconnection networks consuming 10%, 17%, 28% and 36% of total design power respectively.

Cores operating at near-threshold voltages use less power but suffer a substantial time penalty (as the lower current demands a lower frequency) and it has been remarked that "multicores show poor performance in a low-voltage, dim silicon configuration, and prefer operating at a higher-voltage, dark silicon configuration." [152] Here we have cores running seven-times slower than nominal clock speed (with the network and routers using a fixed budget 30% of maximum design power we need the rest of the system to reduce power from 70% of design power to around 20%). The network itself continues running at nominal clock speed. We also assume that each bitmap lookup takes seven ticks.

Figure 6.1 shows that under this regime completion times are high and that blocking is relatively low. For the first iteration of the benchmarks plotted here[1] the coefficient of the relationship between blocking and completion time has fallen to 0.002612, so the ability of partial paging to speed up execution is very limited.

An experiment to judge the impact of dark silicon is not as straight forward. The most obvious issue is how to deal with the 30 - 50% (or more) time processors are blocked: as they

1 Excluding benchmark 0 for which only one data point is available.

are idle they can be relaxed and are unlikely to consume as much power/emit as much heat. But how much power reduction can we assign to this (it will be noted we assumed no power reduction for dark silicon)? That may be a vital performance question.

In fact a real consideration of the issues around dark silicon is likely to require real, partial-paging-supporting hardware if it is to have any practical value even as a design guide: the power management behaviour of real processors being so important in these circumstances.

ALTERNATIVE PAGE REPLACEMENT ALGORITHMS Every fault, hard or small, costs of the order of 1000 cycles due to blocking in the memory connect and faulting-in a 16 byte line that then needs to be replaced will double that cost. Here we only tested a very simple CLOCK and FIFO page replacement, but the high cost of a fault means there is scope for a computationally more expensive replacement policy to be deployed if it reduces the fault rate. One possibility is a version of the Linux 2Q policy [88] or a policy that seeks to avoid replacing pages that need to be written back.

MIXED SCHEDULING FOR LOAD CONTROL We have not considered scheduling questions in this thesis, focusing on questions of execution time instead. Scheduling is likely to be a rich area for experimental exploration as each processor's performance is dependent on that of all others in the system thanks to the shared memory connect. But an area of interest is whether a mixed (as suggested in 4.9.2.2) or adaptive approach to load control would improve overall performance.

6.4 FINAL REMARKS

In the past the standard advice to real-time systems developers has been to not use virtual memory. In the future the question is much more likely to be which form of virtualisation is most appropriate. GPUs and NoCs are similar in that they have a small pool of fast memory available locally and are being tasked to run programs that need to use much more memory than that pool provides. Placing more memory on chips has to be balanced against the loss of space for compute components and does not escape from the dark silicon trap either: so an efficient interaction with global memory storage is essential.

The traditional whole-page paradigm is, in many ways, a left over from the days when all such storage was likely to be some form of spinning magnetic media. It is likely in such circumstances to be more efficient to read off a large contiguous chunk of data in one go, even if much of it isn't needed, than to keep coming back for more. Solid state media eliminate that,

though of course there are other advantages in managing data in large chunks. Partial paging is an attempt to keep some of those advantages (such as a relatively simple mapping of global to local addresses) while also avoiding the dead-weight costs of unnecessarily moving data about a system that is already congested with such traffic.

We believe that we have demonstrated here that it is an avenue worth exploring, but its real test will come when hardware to support the paradigm is built.

A SYNCHRONISED PARALLEL COMPUTATION WITH PARTIAL PAGING

The benchmarks used in Chapters 4 and 5 are not synchronised parallel tasks and we wished to test how such tasks performed.

Using the same techniques and much of the code applied in the 128 core simulations we built a software simulation running a synchronised parallel task and compared performance using 256 cores, 64 cores and 16 cores. The results show that a many-core system, even with partial paging, may perform relatively poorly when conducting synchronised parallel tasks, with the blocking and entropy in the system negating any advantage from being to compute multiple tasks in parallel, especially with the per core slow-down likely to be seen with a many-core system.

As with the other simulations each of these cores can access 16 KB of local memory, but instead of processing an XML trace of a benchmark, we created a synthetic assembly language for the processor cores based loosely on the Ridiculously Simple Computer's instruction set [85]. Using our own coding system allowed us to build our own applications and, additionally, the ability to avoid handling large trace files made it practically possible to look at an even larger - in terms of core numbers - system.

Our instruction set[1] is limited to just 32 separate instructions many of which are pairs of the form `add_(RegA, RegB, RegC)` which adds `RegB` to `RegC` and stores the result in `RegA` and `addi_(RegA, RegB, imm)` which adds `RegB` to the immediate value `imm` and stores the result in `RegA`.

Each instruction is assumed to take a minimum of one or two cycles to operate, to account for the fetch, however each instruction fetch (particularly when operating in partial paging mode) could cause a (hard or small) fault. Instructions which only reference registers, e.g., of the form `add_(RegA, RegB, RegC)` are assumed to be 64 bits long, while instructions which reference an immediate, e.g., of the form `addi_(RegA, RegB, imm)` are taken to be 128 bits long. Each fetch of 64 bits takes at least one cycle (and more if a fault is raised). The two exceptions are the two supported branch instructions `br_` (unconditional branch) and `beq_` (branch if two registers are equal). These are taken to be 64 bits in length despite any reference to an immediate (as though all branching was local). The `div_` and `divi_` instructions, which model register and register/immediate division take 20 cycles to complete - to model the

1 The code can be seen at https://github.com/mcmenaminadrian/euclid - accessed 9 May 2019.

additional time taken by hardware division (a feature copied from the MicroBlaze). It will be seen that timing here is on a somewhat different, if similar, basis to the other simulations.

As in other simulations here a tree is used to connect cores to the external memory, though with 256 cores we have an extra level of depth compared to the examples above.

The system was tested with the Gaussian elimination of a system of 256 simultaneous equations (these were software generated and all begin with integer coefficients) [3]. Each number in the equation was stored in memory in a 136 byte block, with the first 8 bytes set aside to store sign and any other metadata, then the next 64 bytes to store the numerator and the final 64 bytes to store the denominator. The large spaces used are to simulate the space need to implement high precision integer arithmetic, as all numbers are represented in rational form and no floating point maths is used (cf. [94]) - though in fact we do not attempt to use anything beyond 64 bit arithmetic in this implementation.

Our code does not attempt to solve the whole system, merely to reduce the system to row echelon form. The principal computing task is to calculate the greatest common divisor (GCD) using Euclid's Algorithm [2].

When using partial paging 4 of the 16 available page frames were marked as unmovable - one to simulate local 'kernel' space, one to provide space for the page tables, one for the bitmaps and one for the stack. Code (other than interrupt handlers, which were treated as though they were pre-loaded in the kernel page frame) was treated as being loaded from global memory, and code pages were marked as read-only (so could be discarded without a write-back). Data pages were marked as read-write and could, under software control, be written back or simply discarded (e.g., if a page was simply being read to understand global status, the relevant parts were loaded and then discarded.) On a write-back only those parts of a page that were marked valid in the bitmap needed to be written.

A simple CLOCK-like page replacement algorithm was used. Every 40,000 cycles a flag bit in the page table for eight (moveable) pages was reset and the page removed from the TLB but was still present in the page table. A subsequent access would see the page returned to the TLB and the bit reset. If a page frame was required to accommodate a page currently not in local memory and there were no free page frames, pages marked by the CLOCK were liable to be replaced. If no such pages were available then a page would be chosen for replacement through a simple sequence. We did not use the mesh interconnect to co-ordinate the cores, using only shared global memory.

In one system 256 processors are deployed against the 256 rows, in the second 64 processors are used (each processor tackling 4 rows) and in the third 16 processors (each processor tackling 16 rows).

Gaussian elimination can be thought of as one of the classical problems of parallel computing, but it is important to note that our algorithm does contain serial and semi-serial elements.

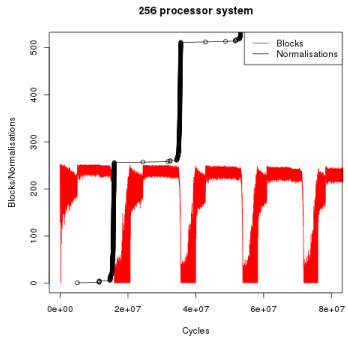

Figure A.1: Gaussian elimination: 256 core system

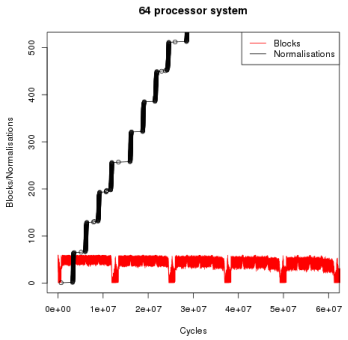

Figure A.2: Gaussian elimination: 64 core system

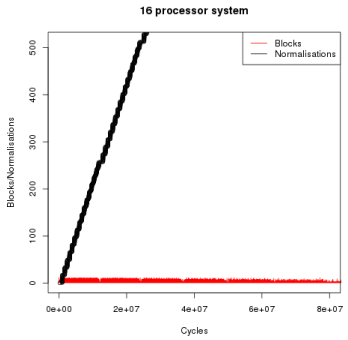

Figure A.3: Gaussian elimination: 16 core system

Ignoring questions of initial synchronisation for now (which may involve processors waiting in a queue until the system is ready), the first computing task is to normalise row 0 of the augmented matrix formed by the system of equations we wish to solve. We do this by having a single processor attack each coefficient of row 0 at a time, so requiring 256 computations in serial order. This represents 100% serial code. (We could instead have multiple processors tackle the row coefficients, though synchronisation and memory management issues may have left this as less efficient but the question is moot.)

In the 256 core system the other 255 processors modify the coefficients on their assigned line while the processor that tackled row 0 loops waiting for the other lines to be finished. In the 64 (or 16) core system all 64 (or 16) cores run through the system in groups of 64 (16) equations.

We can consider this to be semi-serial, but the fraction of code that is serial rises as calculations for different lines will finish at different times if only because of factors inherent in the Euclidean Algorithm itself[2].

As later rows in the system are normalised then processors in the 256 core system allocated to previous rows stand idle, meaning the system becomes progressively more serial-like in nature.

Figures A.1 - A.3 show the performance of the different systems. Blocking is so low in the 16 core system it is difficult to follow progress but in the 256 and 64 core systems we can see a plateau of high blocking where all (or nearly all) the cores are active (and blocked).

The evidence suggests that system performance is dominated by blocking in the memory tree: each processor has to read in continuous ranges of memory as each line is processed. Synchronisation is managed through shared memory with the processors using polling

2 These are outside our scope but are discussed in pp. 356 - 373 of [94]

Iteration and system	Normalise pivot line	Total normalisation time	Fully parallel time
1st pass: 256 core system	4932449	15819591	10887142
2nd pass: 256 core system	3745597	14906061	11160464
3rd pass: 256 core system	2876858	13888268	11011410
4th pass: 256 core system	2659862	13571418	10911556
1st pass: 64 core system	656811	11907815	11251004
2nd pass: 64 core system	652412	13216775	12564363
3rd pass: 64 core system	654657	11647881	10993224
4th pass: 64 core system	700807	11594288	10893481
1st pass: 16 core system	304904	11784360	11479456
2nd pass: 16 core system	333996	11968158	11634162
3rd pass: 16 core system	347868	12363788	12015920
4th pass: 16 core system	373966	12934073	12560107

Table A.1: Small (16 core), medium (64 core) and large (256 core) system performances at Gaussian normalisation compared (in simulated cycles)

to check for signals, so adding to demand on the memory system. This (admittedly crude) mechanism can generate high levels of memory traffic, even though a simple backoff procedure is used to limit traffic.

Specifically considering the 256 core system - for the first 5 million cycles traffic is generated largely by processors polling to see if the first row has been normalised: this is completed at 5,084,476 cycles and then the system becomes saturated with memory requests, the number of blocks only falling slightly as a small group of processors finish processing after around 11.5 million cycles. Blocking then falls rapidly as the bulk of processors start to complete after around 14.7 million cycles - but it is not until about 15.9 million cycles that the typical number of blocked processors falls to 128 or fewer - so before that point most processors are blocked most of the time.

Table A.1 suggests that the 64 core arrangement may be the most efficient of the three tested if all cores ran at the same speed - at least after the initial runs and over the range chosen: the 256 core system loses a processor on each run, while the 64 core system will not have that as an issue until 192 lines have been normalised. This, though ignores the factors outlined in

Table 2.1: which suggest that a 64-core system could see each core run around 3 times faster than a 256 core system and a 16 core system run cores at 3.7 times the speed of the 64 core system. Assuming the whole system ran at the same clock speed this would indicate that the smaller system ran at the fastest speed.

(The normalisation of the pivot line - a piece of code executed in serial - runs faster on the simulations with fewer cores, almost certainly because fewer cores generate less polling traffic in the memory connect. This traffic, combined with the larger memory connection trees in the bigger system, increases average waiting time and - by increasing entropy - the maximum waiting time also, so degrading performance. A better algorithm might reduce this traffic and lead to better performance of the larger systems here, but that would not matter for the fully parallel timings where most of the traffic is related to memory fetching.)

B LACKEYML OUTPUT

Listing B.1 shows the initialisation code for benchmark 0 (in fact this code is common to all the benchmarks). The first two lines load an address to jump to and then jump there. Lines 3 - 90 are initialisation code that references either immediate values or other registers, then, at line 91 we have the first three external memory references with three `SD` (store double) operands, which store data on the stack.

Listing B.2 shows the lackeyml that is generated by this code. After the DTD (lines 1 - 20) and the namespace definition (line 21), there is a one-to-one correspondence between the assembly code and the XML until we reach line 113 of the XML (which maps to line 91 of the assembly): here the `SD ra, 56(SP)` maps to two lines of Lackeyml XML - an `instruction` at address `0x80001934` and a `store` which writes to address `0x80024CF8`.

In terms of our simulator the assembly that only references immediate and other registers will require a minimum of one cycle to execute, while code that writes (`store`) or reads (`load`) external memory will require a minimum of two cycles to execute. There are no `modify` type instructions with RISC-V.

```
1  core 0: 0x0000000000001000 (0x7ffff297) auipc t0, 0x7ffff
2  core 0: 0x0000000000001004 (0x00028067) jr t0
3  core 0: 0x0000000080000000 (0x00000297) auipc t0, 0x0
4  core 0: 0x0000000080000004 (0x16428293) addi t0, t0, 356
5  core 0: 0x0000000080000008 (0x30529073) csrw mtvec, t0
6  core 0: 0x000000008000000c (0x00000093) li ra, 0
7  core 0: 0x0000000080000010 (0x00000113) li sp, 0
8  core 0: 0x0000000080000014 (0x00000193) li gp, 0
9  core 0: 0x0000000080000018 (0x00000213) li tp, 0
10 core 0: 0x000000008000001c (0x00000293) li t0, 0
11 core 0: 0x0000000080000020 (0x00000313) li t1, 0
12 core 0: 0x0000000080000024 (0x00000393) li t2, 0
13 core 0: 0x0000000080000028 (0x00000413) li s0, 0
14 core 0: 0x000000008000002c (0x00000493) li s1, 0
15 core 0: 0x0000000080000030 (0x00000513) li a0, 0
16 core 0: 0x0000000080000034 (0x00000593) li a1, 0
17 core 0: 0x0000000080000038 (0x00000613) li a2, 0
18 core 0: 0x000000008000003c (0x00000693) li a3, 0
19 core 0: 0x0000000080000040 (0x00000713) li a4, 0
20 core 0: 0x0000000080000044 (0x00000793) li a5, 0
```

```
core 0: 0x0000000080000048 (0x00000813) li a6, 0
core 0: 0x000000008000004c (0x00000893) li a7, 0
core 0: 0x0000000080000050 (0x00000913) li s2, 0
core 0: 0x0000000080000054 (0x00000993) li s3, 0
core 0: 0x0000000080000058 (0x00000a13) li s4, 0
core 0: 0x000000008000005c (0x00000a93) li s5, 0
core 0: 0x0000000080000060 (0x00000b13) li s6, 0
core 0: 0x0000000080000064 (0x00000b93) li s7, 0
core 0: 0x0000000080000068 (0x00000c13) li s8, 0
core 0: 0x000000008000006c (0x00000c93) li s9, 0
core 0: 0x0000000080000070 (0x00000d13) li s10, 0
core 0: 0x0000000080000074 (0x00000d93) li s11, 0
core 0: 0x0000000080000078 (0x00000e13) li t3, 0
core 0: 0x000000008000007c (0x00000e93) li t4, 0
core 0: 0x0000000080000080 (0x00000f13) li t5, 0
core 0: 0x0000000080000084 (0x00000f93) li t6, 0
core 0: 0x0000000080000088 (0x0001e2b7) lui t0, 0x1e
core 0: 0x000000008000008c (0x3002a073) csrs mstatus, t0
core 0: 0x0000000080000090 (0xf10022f3) csrr t0, misa
core 0: 0x0000000080000094 (0x0002c663) bltz t0, pc + 12
core 0: 0x00000000800000a0 (0x0202f293) andi t0, t0, 32
core 0: 0x00000000800000a4 (0x08028463) beqz t0, pc + 136
core 0: 0x00000000800000a8 (0x00301073) csrw fcsr, zero
core 0: 0x00000000800000ac (0xf0000053) fmv.s.x ft0, zero
core 0: 0x00000000800000b0 (0xf00000d3) fmv.s.x ft1, zero
core 0: 0x00000000800000b4 (0xf0000153) fmv.s.x ft2, zero
core 0: 0x00000000800000b8 (0xf00001d3) fmv.s.x ft3, zero
core 0: 0x00000000800000bc (0xf0000253) fmv.s.x ft4, zero
core 0: 0x00000000800000c0 (0xf00002d3) fmv.s.x ft5, zero
core 0: 0x00000000800000c4 (0xf0000353) fmv.s.x ft6, zero
core 0: 0x00000000800000c8 (0xf00003d3) fmv.s.x ft7, zero
core 0: 0x00000000800000cc (0xf0000453) fmv.s.x fs0, zero
core 0: 0x00000000800000d0 (0xf00004d3) fmv.s.x fs1, zero
core 0: 0x00000000800000d4 (0xf0000553) fmv.s.x fa0, zero
core 0: 0x00000000800000d8 (0xf00005d3) fmv.s.x fa1, zero
core 0: 0x00000000800000dc (0xf0000653) fmv.s.x fa2, zero
core 0: 0x00000000800000e0 (0xf00006d3) fmv.s.x fa3, zero
core 0: 0x00000000800000e4 (0xf0000753) fmv.s.x fa4, zero
core 0: 0x00000000800000e8 (0xf00007d3) fmv.s.x fa5, zero
core 0: 0x00000000800000ec (0xf0000853) fmv.s.x fa6, zero
core 0: 0x00000000800000f0 (0xf00008d3) fmv.s.x fa7, zero
core 0: 0x00000000800000f4 (0xf0000953) fmv.s.x fs2, zero
core 0: 0x00000000800000f8 (0xf00009d3) fmv.s.x fs3, zero
core 0: 0x00000000800000fc (0xf0000a53) fmv.s.x fs4, zero
core 0: 0x0000000080000100 (0xf0000ad3) fmv.s.x fs5, zero
```

```
core 0: 0x0000000080000104 (0xf0000b53) fmv.s.x fs6, zero
core 0: 0x0000000080000108 (0xf0000bd3) fmv.s.x fs7, zero
core 0: 0x000000008000010c (0xf0000c53) fmv.s.x fs8, zero
core 0: 0x0000000080000110 (0xf0000cd3) fmv.s.x fs9, zero
core 0: 0x0000000080000114 (0xf0000d53) fmv.s.x fs10, zero
core 0: 0x0000000080000118 (0xf0000dd3) fmv.s.x fs11, zero
core 0: 0x000000008000011c (0xf0000e53) fmv.s.x ft8, zero
core 0: 0x0000000080000120 (0xf0000ed3) fmv.s.x ft9, zero
core 0: 0x0000000080000124 (0xf0000f53) fmv.s.x ft10, zero
core 0: 0x0000000080000128 (0xf0000fd3) fmv.s.x ft11, zero
core 0: 0x000000008000012c (0x00005197) auipc gp, 0x5
core 0: 0x0000000080000130 (0xf4418193) addi gp, gp, -188
core 0: 0x0000000080000134 (0x00005217) auipc tp, 0x5
core 0: 0x0000000080000138 (0xc0b20213) addi tp, tp, -1013
core 0: 0x000000008000013c (0xfc027213) andi tp, tp, -64
core 0: 0x0000000080000140 (0xf1402573) csrr a0, mhartid
core 0: 0x0000000080000144 (0x00100593) li a1, 1
core 0: 0x0000000080000148 (0x00b57063) bgeu a0, a1, pc + 0
core 0: 0x000000008000014c (0x01151613) slli a2, a0, 17
core 0: 0x0000000080000150 (0x00c20233) add tp, tp, a2
core 0: 0x0000000080000154 (0x00150113) addi sp, a0, 1
core 0: 0x0000000080000158 (0x01111113) slli sp, sp, 17
core 0: 0x000000008000015c (0x00410133) add sp, sp, tp
core 0: 0x0000000080000160 (0x7d00106f) j pc + 0x17d0
core 0: 0x0000000080001930 (0xfc010113) addi sp, sp, -64
core 0: 0x0000000080001934 (0x02113c23) sd ra, 56(sp)
core 0: 0x0000000080001938 (0x02813823) sd s0, 48(sp)
core 0: 0x000000008000193c (0x02913423) sd s1, 40(sp)
core 0: 0x0000000080001940 (0x04010413) addi s0, sp, 64
```

Listing B.1: RISCV Assembly for initialisation of benchmark 0

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE lackeyml [
<!ELEMENT lackeyml (application,(instruction|store|load|modify)*)>
<!ATTLIST lackeyml version CDATA #FIXED "0.1">
<!ATTLIST lackeyml xmlns CDATA #FIXED "http://cartesianproduct.wordpress.com">
<!ELEMENT application EMPTY>
<!ATTLIST application command CDATA #REQUIRED>
<!ELEMENT instruction EMPTY>
<!ATTLIST instruction address CDATA #REQUIRED>
<!ATTLIST instruction size CDATA #REQUIRED>
<!ELEMENT modify EMPTY>
<!ATTLIST modify address CDATA #REQUIRED>
<!ATTLIST modify size CDATA #REQUIRED>
```

```
<!ELEMENT store EMPTY>
<!ATTLIST store address CDATA #REQUIRED>
<!ATTLIST store size CDATA #REQUIRED>
<!ELEMENT load EMPTY>
<!ATTLIST load address CDATA #REQUIRED>
<!ATTLIST load size CDATA #REQUIRED>
]>
<lackeyml xmlns="http://cartesianproduct.wordpress.com">
<instruction address='0x0000000000001000' size='4' />
<instruction address='0x0000000000001004' size='4' />
<instruction address='0x0000000080000000' size='4' />
<instruction address='0x0000000080000004' size='4' />
<instruction address='0x0000000080000008' size='4' />
<instruction address='0x000000008000000c' size='4' />
<instruction address='0x0000000080000010' size='4' />
<instruction address='0x0000000080000014' size='4' />
<instruction address='0x0000000080000018' size='4' />
<instruction address='0x000000008000001c' size='4' />
<instruction address='0x0000000080000020' size='4' />
<instruction address='0x0000000080000024' size='4' />
<instruction address='0x0000000080000028' size='4' />
<instruction address='0x000000008000002c' size='4' />
<instruction address='0x0000000080000030' size='4' />
<instruction address='0x0000000080000034' size='4' />
<instruction address='0x0000000080000038' size='4' />
<instruction address='0x000000008000003c' size='4' />
<instruction address='0x0000000080000040' size='4' />
<instruction address='0x0000000080000044' size='4' />
<instruction address='0x0000000080000048' size='4' />
<instruction address='0x000000008000004c' size='4' />
<instruction address='0x0000000080000050' size='4' />
<instruction address='0x0000000080000054' size='4' />
<instruction address='0x0000000080000058' size='4' />
<instruction address='0x000000008000005c' size='4' />
<instruction address='0x0000000080000060' size='4' />
<instruction address='0x0000000080000064' size='4' />
<instruction address='0x0000000080000068' size='4' />
<instruction address='0x000000008000006c' size='4' />
<instruction address='0x000000008000006c' size='4' />
<instruction address='0x0000000080000070' size='4' />
<instruction address='0x0000000080000074' size='4' />
<instruction address='0x0000000080000078' size='4' />
<instruction address='0x000000008000007c' size='4' />
<instruction address='0x0000000080000080' size='4' />
<instruction address='0x0000000080000084' size='4' />
```

```
<instruction address='0x0000000080000088' size='4' />
<instruction address='0x000000008000008c' size='4' />
<instruction address='0x0000000080000090' size='4' />
<instruction address='0x0000000080000094' size='4' />
<instruction address='0x00000000800000a0' size='4' />
<instruction address='0x00000000800000a4' size='4' />
<instruction address='0x00000000800000a8' size='4' />
<instruction address='0x00000000800000ac' size='4' />
<instruction address='0x00000000800000b0' size='4' />
<instruction address='0x00000000800000b4' size='4' />
<instruction address='0x00000000800000b8' size='4' />
<instruction address='0x00000000800000bc' size='4' />
<instruction address='0x00000000800000c0' size='4' />
<instruction address='0x00000000800000c4' size='4' />
<instruction address='0x00000000800000c8' size='4' />
<instruction address='0x00000000800000cc' size='4' />
<instruction address='0x00000000800000d0' size='4' />
<instruction address='0x00000000800000d4' size='4' />
<instruction address='0x00000000800000d8' size='4' />
<instruction address='0x00000000800000dc' size='4' />
<instruction address='0x00000000800000e0' size='4' />
<instruction address='0x00000000800000e4' size='4' />
<instruction address='0x00000000800000e8' size='4' />
<instruction address='0x00000000800000ec' size='4' />
<instruction address='0x00000000800000f0' size='4' />
<instruction address='0x00000000800000f4' size='4' />
<instruction address='0x00000000800000f8' size='4' />
<instruction address='0x00000000800000fc' size='4' />
<instruction address='0x0000000080000100' size='4' />
<instruction address='0x0000000080000104' size='4' />
<instruction address='0x0000000080000108' size='4' />
<instruction address='0x000000008000010c' size='4' />
<instruction address='0x0000000080000110' size='4' />
<instruction address='0x0000000080000114' size='4' />
<instruction address='0x0000000080000118' size='4' />
<instruction address='0x000000008000011c' size='4' />
<instruction address='0x0000000080000120' size='4' />
<instruction address='0x0000000080000124' size='4' />
<instruction address='0x0000000080000128' size='4' />
<instruction address='0x000000008000012c' size='4' />
<instruction address='0x0000000080000130' size='4' />
<instruction address='0x0000000080000134' size='4' />
<instruction address='0x0000000080000138' size='4' />
<instruction address='0x000000008000013c' size='4' />
<instruction address='0x0000000080000140' size='4' />
```

```
104 <instruction address='0x0000000080000144' size='4' />
105 <instruction address='0x0000000080000148' size='4' />
106 <instruction address='0x000000008000014c' size='4' />
107 <instruction address='0x0000000080000150' size='4' />
108 <instruction address='0x0000000080000154' size='4' />
109 <instruction address='0x0000000080000158' size='4' />
110 <instruction address='0x000000008000015c' size='4' />
111 <instruction address='0x0000000080000160' size='4' />
112 <instruction address='0x0000000080001930' size='4' />
113 <instruction address='0x0000000080001934' size='4' />
114 <store address='0x80024cf8' size='8' />
115 <instruction address='0x0000000080001938' size='4' />
116 <store address='0x80024cf0' size='8' />
117 <instruction address='0x000000008000193c' size='4' />
118 <store address='0x80024ce8' size='8' />
119 <instruction address='0x0000000080001940' size='4' />
```

Listing B.2: Lackeyml listing for opening code of Benchmark o

C LINEAR MODELLING FOR TRADITIONAL 1KB PAGING

Table C.1 shows the results of a linear model of benchmark performance with traditional paging and it can be seen that all the benchmarks (with the exception of 6, for which no results were available as no benchmarks completed in the test time) point to a very strong linear relationship between blocking and completion times, though also indicate a slightly smaller coefficient linking blocking to completion times compared to partial paging (cf. Table 4.4). In fact for traditional paging the proportion between the theoretical intercepts and the minimal theoretical completion times are smaller and the blocking coefficients are also smaller - suggesting that the traditional method might be more fundamentally efficient. However to look at these measures alone is to ignore the very large decrease in efficiency caused by excess blocking and also does not consider the way in which blocking adds to uncertainty in the system by increasing entropy.

Benchmark	Observations	Intercept	Std. Error	Significance[1]	Proportion[2]	Coefficient	Std. Error	Significance
0	580	1.756×10^6	2.401×10^4	$< 2 \times 10^{-16}$	1.39	0.9772	5.420×10^{-3}	$< 2 \times 10^{-16}$
1	1128	4.471×10^5	1.028×10^3	$< 2 \times 10^{-16}$	1.19	1.045	3.879×10^{-4}	$< 2 \times 10^{-16}$
2	1174	4.519×10^5	1.002×10^3	$< 2 \times 10^{-16}$	1.19	1.043	3.981×10^{-4}	$< 2 \times 10^{-16}$
3	372	8.337×10^5	8.266×10^3	$< 2 \times 10^{-16}$	1.27	1.041	1.010×10^{-3}	$< 2 \times 10^{-16}$
4	273	9.710×10^6	2.158×10^3	$< 2 \times 10^{-16}$	1.17	1.048	7.633×10^{-4}	$< 2 \times 10^{-16}$
5	1159	4.079×10^5	1.333×10^3	$< 2 \times 10^{-16}$	1.23	1.046	5.223×10^{-4}	$< 2 \times 10^{-16}$
6	0[3]							
7	165	9.639×10^5	2.089×10^4	$< 2 \times 10^{-16}$	1.31	1.051	1.126×10^{-3}	$< 2 \times 10^{-16}$

Table C.1: Modelling execution times as a dependent variable on blocks in the 1KB traditional system

[1] Smaller numbers indicate more significant results
[2] The ratio between Intercept and minimal theoretical execution time
[3] No instance of this benchmark completed in the 2.22×10^8cycles across which we collected data.

D LINEAR MODELLING FOR 1KB PARTIAL PAGING WITH DELAY

Table D.1, built with R's linear model (cf. 4.4), shows that the coefficient for blocking remains in the range $1.0 \sim 1.2$ for all benchmarks. The intercept, as expected, doubles in almost all cases: as the intercept represents the time the model predicts the benchmark would take if there was no blocking, the 100% increase in basic execution time caused by bitmap reading is reflected here.

In balance this would suggest that where blocking is high, the reduction in overall system blocking rates (by around one-quarter) should deliver an improved performance but elsewhere performance would be expected to fall - and that seems to be the pattern, with benchmarks 0, 3, 6 and 7 improving performance and others seeing performance deteriorate.

Very approximately, if 25% of the time lost to blocking for the process without the bitmap reading delay is greater than the theoretical minimal execution time then adding the delay is likely to speed the benchmark up. Of course the opposite also applies and other benchmarks are slowed down. However this does suggest that an adaptive approach might work: if a process generates a high number of faults it could be subject to a form of load control and yet speed overall system performance.

Benchmark	Observations	Intercept	Std. Error	Significance[1]	Proportion[2]	Coefficient	Std. Error	Significance
0	64	3.513×10^6	2.433×10^5	$< 2 \times 10^{-16}$	2.78	1.185	1.099×10^{-2}	$< 2 \times 10^{-16}$
1	1393	9.163×10^5	1.105×10^3	$< 2 \times 10^{-16}$	2.43	1.093	2.158×10^{-3}	$< 2 \times 10^{-16}$
2	1442	9.116×10^5	9.557×10^2	$< 2 \times 10^{-16}$	2.41	1.105	2.065×10^{-3}	$< 2 \times 10^{-16}$
3	400	1.920×10^6	7.737×10^3	$< 2 \times 10^{-16}$	2.92	1.029	2.627×10^{-3}	$< 2 \times 10^{-16}$
4	96	1.950×10^7	5.943×10^3	$< 2 \times 10^{-16}$	2.35	1.107	7.848×10^{-3}	$< 2 \times 10^{-16}$
5	1100	8.509×10^5	2.085×10^3	$< 2 \times 10^{-16}$	2.56	1.086	2.255×10^{-3}	$< 2 \times 10^{-16}$
6	16	4.261×10^7	1.464×10^6	6.33×10^{-14}	3.17	1.057	1.870×10^{-2}	$< 2 \times 10^{-16}$
7	449	2.007×10^6	9.372×10^3	$< 2 \times 10^{-16}$	2.73	1.060	4.047×10^{-3}	$< 2 \times 10^{-16}$

Table D.1: Modelling execution times as a dependent variable on blocks in the 1KB partial paging system with 1 cycle delay for bitmap reading

[1] Smaller numbers indicate more significant results
[2] The ratio between Intercept and minimal theoretical execution time

E DISTRIBUTION OF OBSERVED COMPLETION TIMES FOR INDIVIDUAL BENCHMARKS FOR 1KB PAGING

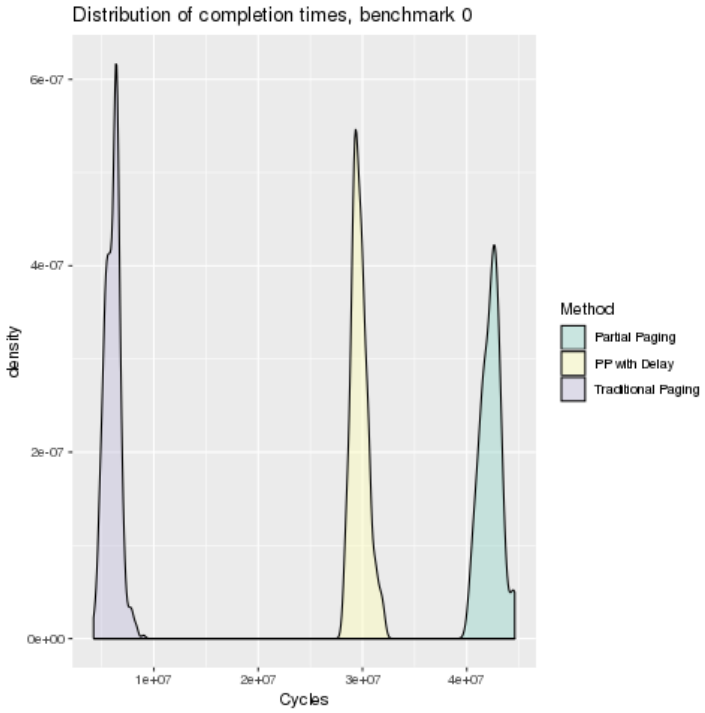

Figure E.1: Completion times for benchmark 0 with 1KB pages for partial paging, traditional paging and partial paging with 1 cycle cost of bitmap access

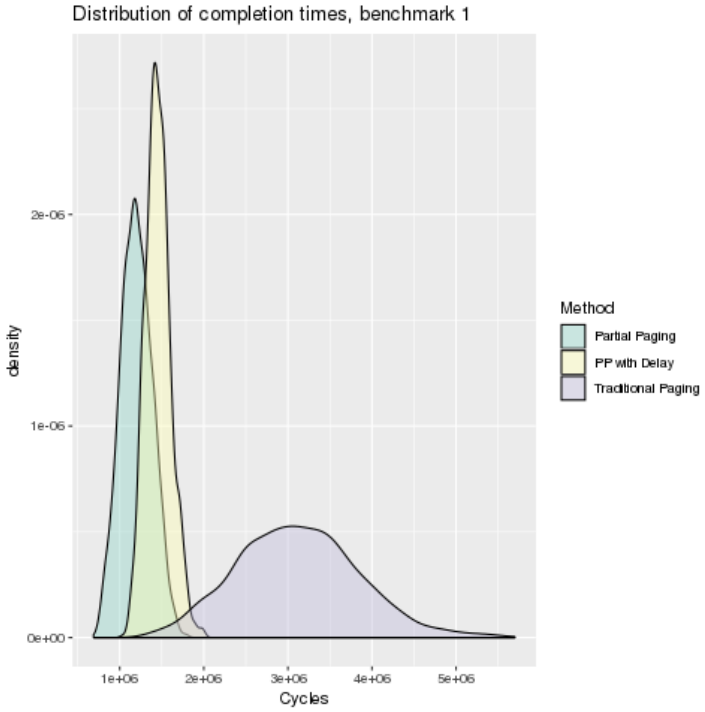

Figure E.2: Completion times for benchmark 1 with 1KB pages for partial paging, traditional paging and partial paging with 1 cycle cost of bitmap access

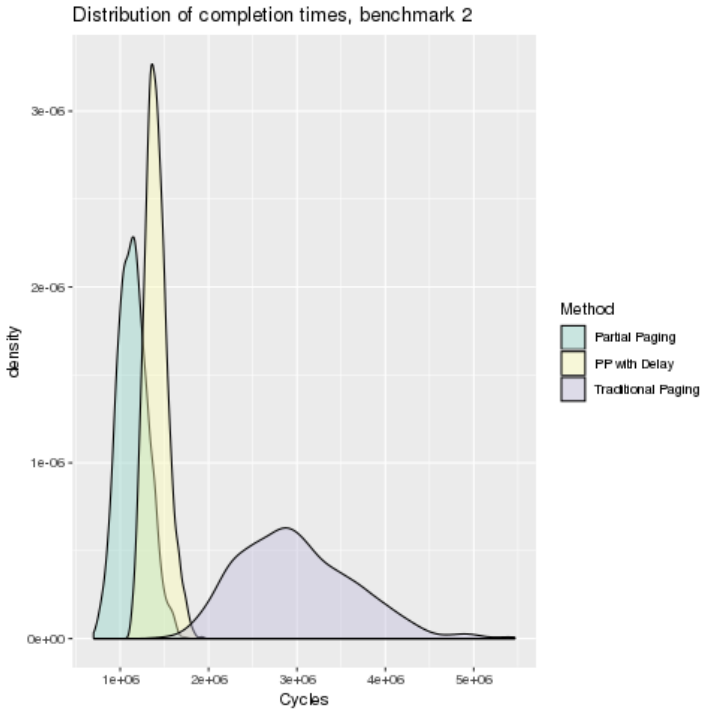

Figure E.3: Completion times for benchmark 2 with 1KB pages for partial paging, traditional paging and partial paging with 1 cycle cost of bitmap access

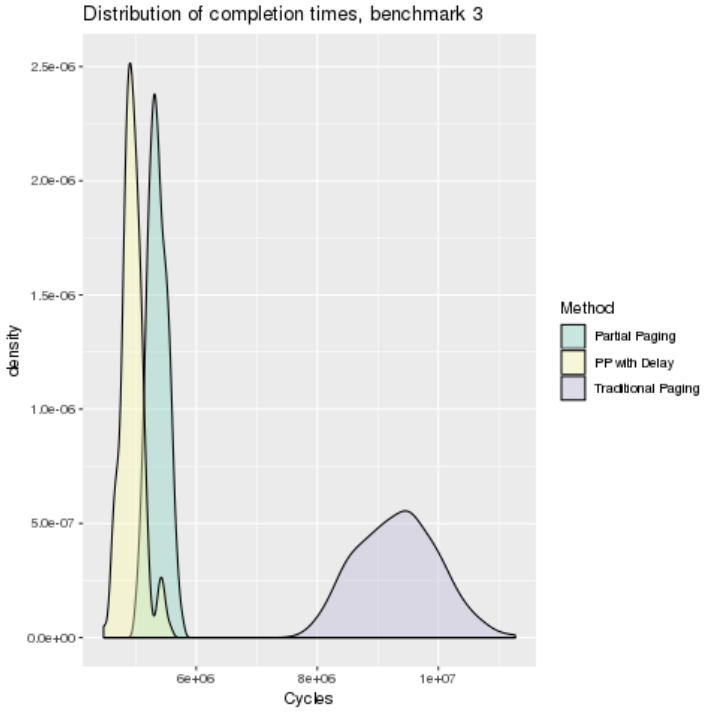

Figure E.4: Completion times for benchmark 3 with 1KB pages for partial paging, traditional paging and partial paging with 1 cycle cost of bitmap access

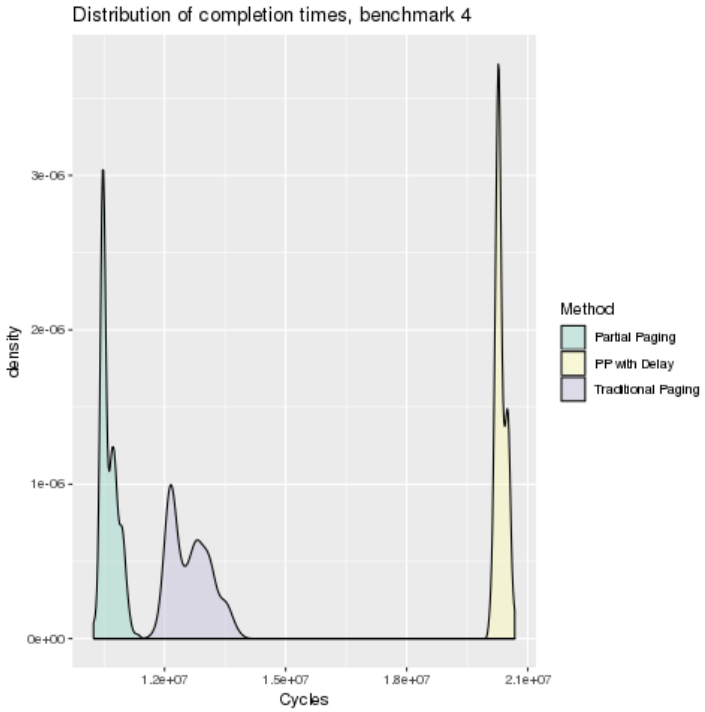

Figure E.5: Completion times for benchmark 4 with 1KB pages for partial paging, traditional paging and partial paging with 1 cycle cost of bitmap access

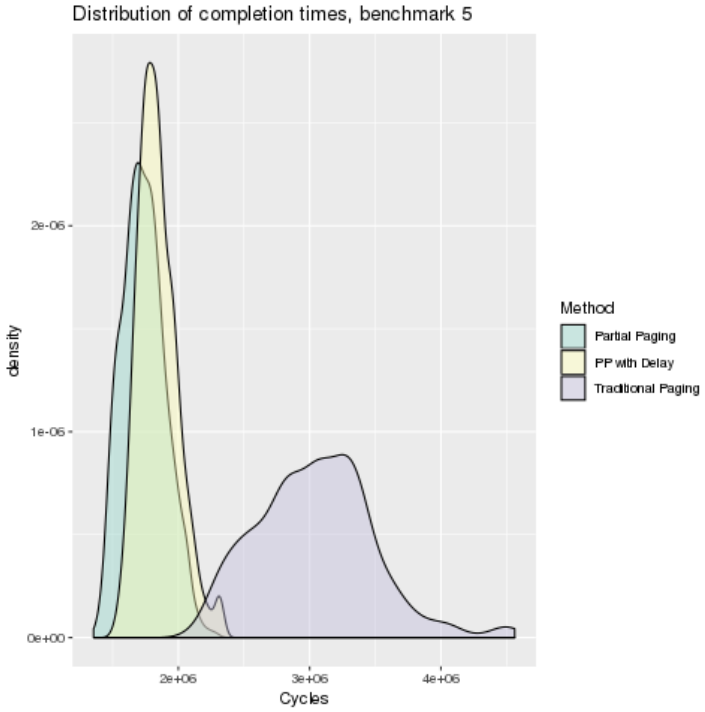

Figure E.6: Completion times for benchmark 5 with 1KB pages for partial paging, traditional paging and partial paging with 1 cycle cost of bitmap access

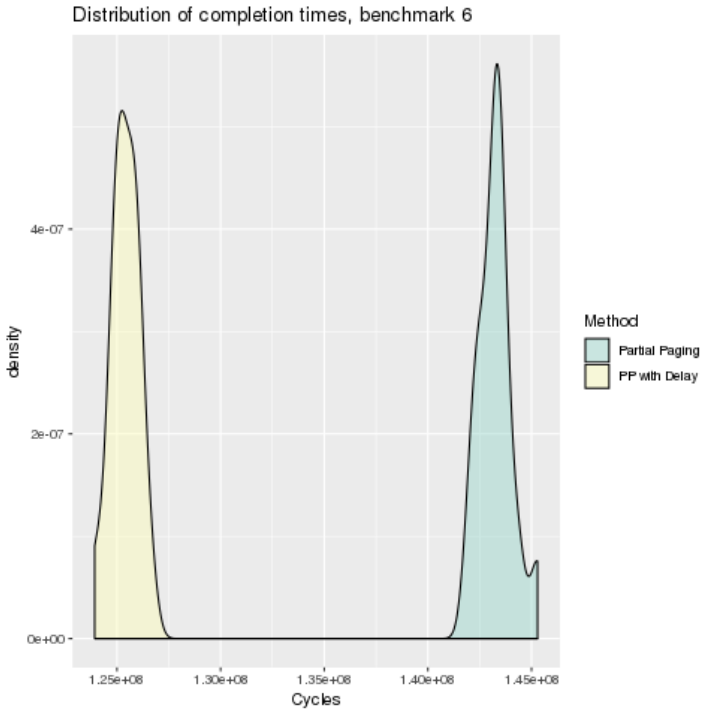

Figure E.7: Completion times for benchmark 6 with 1KB pages for partial paging and partial paging with 1 cycle cost of bitmap access

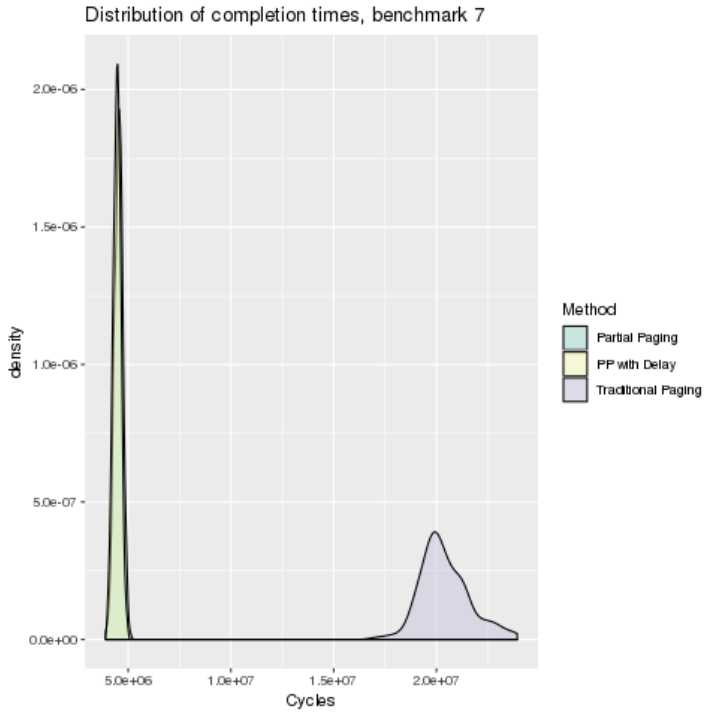

Figure E.8: Completion times for benchmark 7 with 1KB pages for partial paging, traditional paging and partial paging with 1 cycle cost of bitmap access

F GUMBEL DISTRIBUTION PLOTS FOR 1KB PARTIAL PAGING (CLOCK)

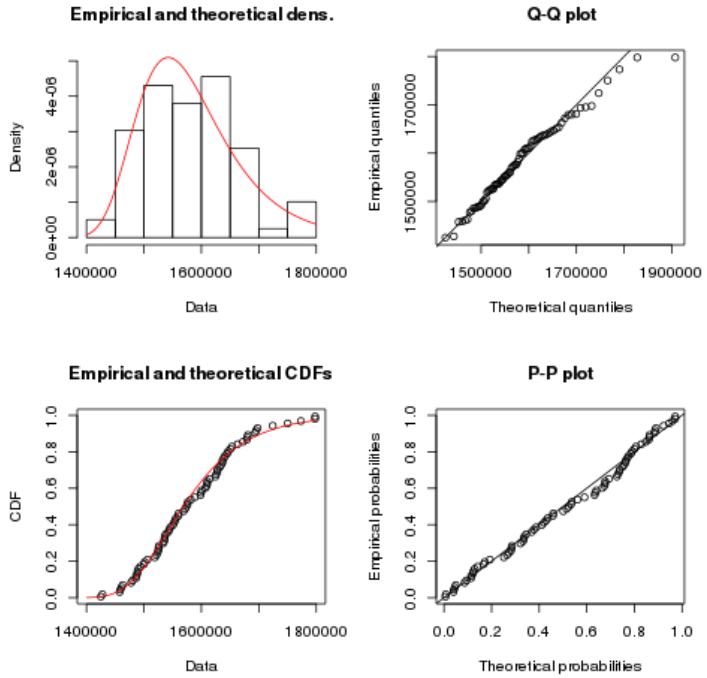

Figure F.1: Computed Gumbel distribution for benchmark 1 maxima with 1KB partial paging.

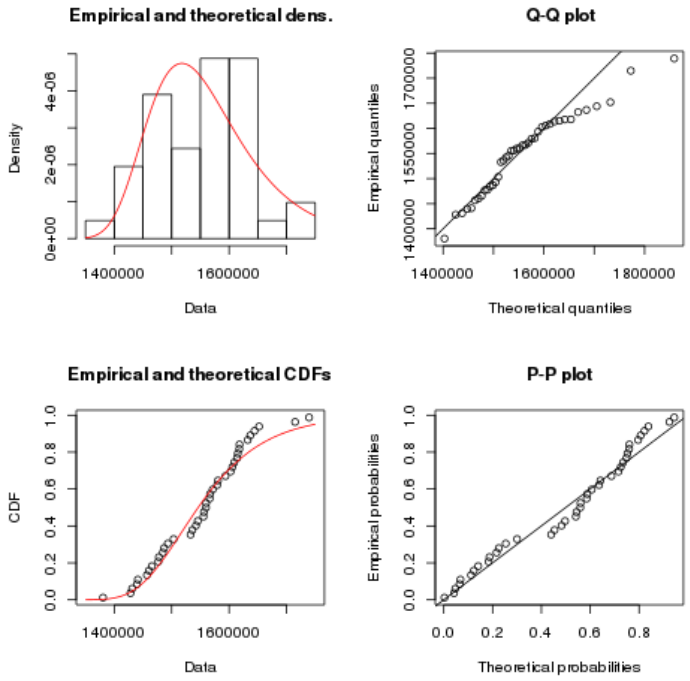

Figure F.2: Computed Gumbel distribution for benchmark 2 maxima with 1KB partial paging.

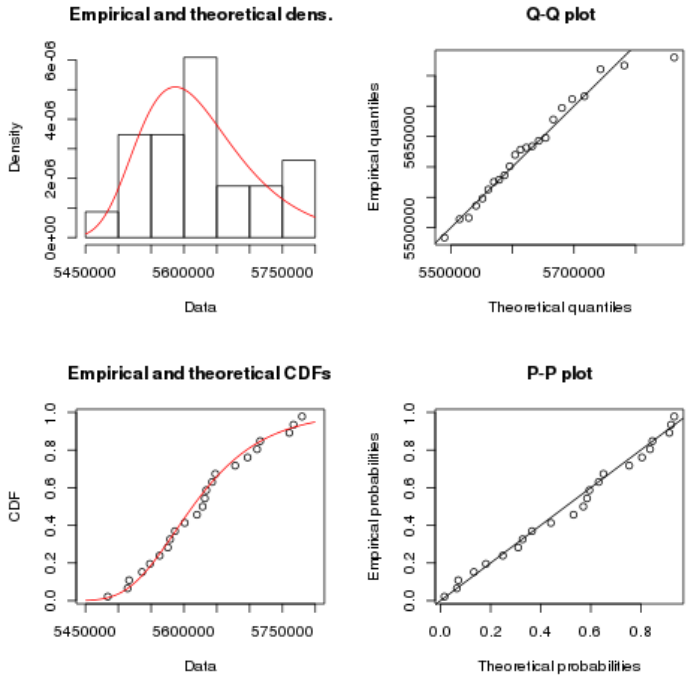

Figure F.3: Computed Gumbel distribution for benchmark 3 maxima with 1KB partial paging

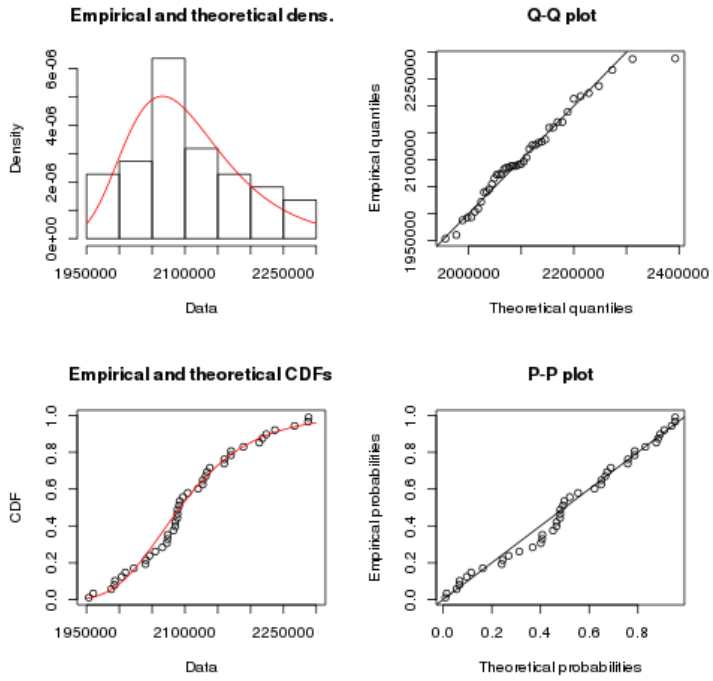

Figure F.4: Computed Gumbel distribution for benchmark 5 maxima with 1KB partial paging.

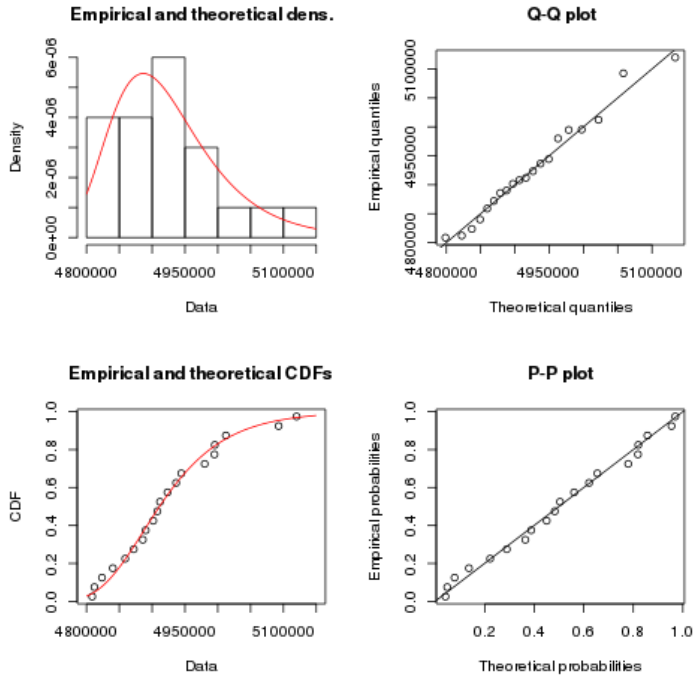

Figure F.5: Computed Gumbel distribution for benchmark 7 maxima with 1KB partial paging.

G LINEAR MODELLING FOR 512 BYTE PARTIAL PAGING

NB: All significance scores were $< 2 \times 10^{-16}$ - the minimum value supported by R - and so while all results were judged significant the significance values are not listed separately.

Benchmark	Observations	Intercept	Std. Error	Proportion[1]	Coefficient	Std. Error
0	1635	1.549×10^{6}	7.542×10^{2}	1.23	1.186	1.591×10^{-3}
1	7712	4.416×10^{5}	2.935	1.17	1.144	8.354×10^{-5}
2	6683	4.451×10^{5}	5.132×10^{1}	1.18	1.168	6.347×10^{-4}
3	1005	1.133×10^{6}	2.977×10^{4}	1.72	1.038	1.367×10^{-3}
4	327	9.773×10^{6}	8.622×10^{2}	1.18	1.155	2.220×10^{-3}
5	3302	4.704×10^{5}	6.725×10^{2}	1.42	1.045	1.231×10^{-3}
6	32	2.528×10^{7}	1.495×10^{6}	1.88	1.099	2.040×10^{-2}
7	1041	1.197×10^{6}	5.518×10^{3}	1.63	1.050	2.805×10^{-3}

Table G.1: Calculated linear regression factors for benchmarks with 512 byte pages and partial paging

[1] The ratio between Intercept and minimal theoretical execution time

H GUMBEL DISTRIBUTION PLOTS FOR 512 BYTE PARTIAL PAGING (CLOCK)

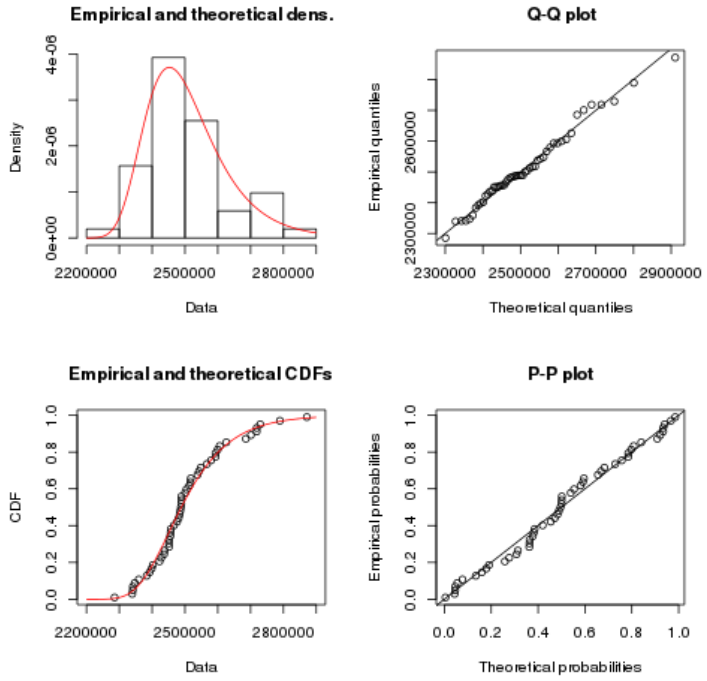

Figure H.1: Fitted Gumbel distribution for extreme completion data for benchmark 0 with 512 byte pages under partial paging

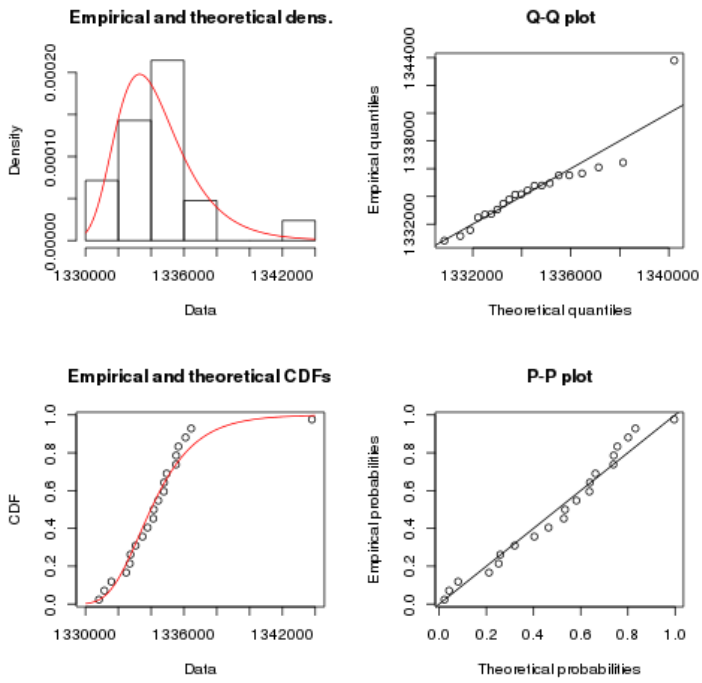

Figure H.2: Fitted Gumbel distribution for extreme completion data for benchmark 1 with 512 byte pages under partial paging

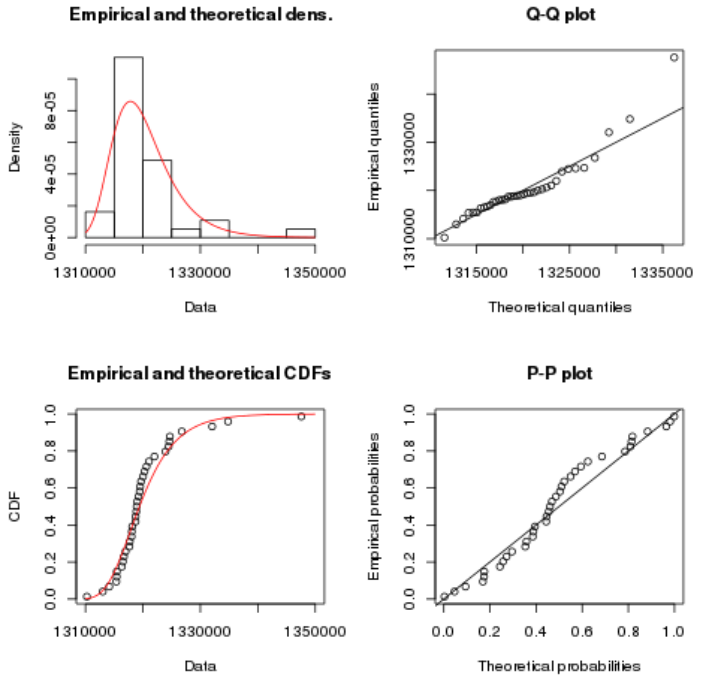

Figure H.3: Fitted Gumbel distribution for extreme completion data for benchmark 2 with 512 byte pages under partial paging

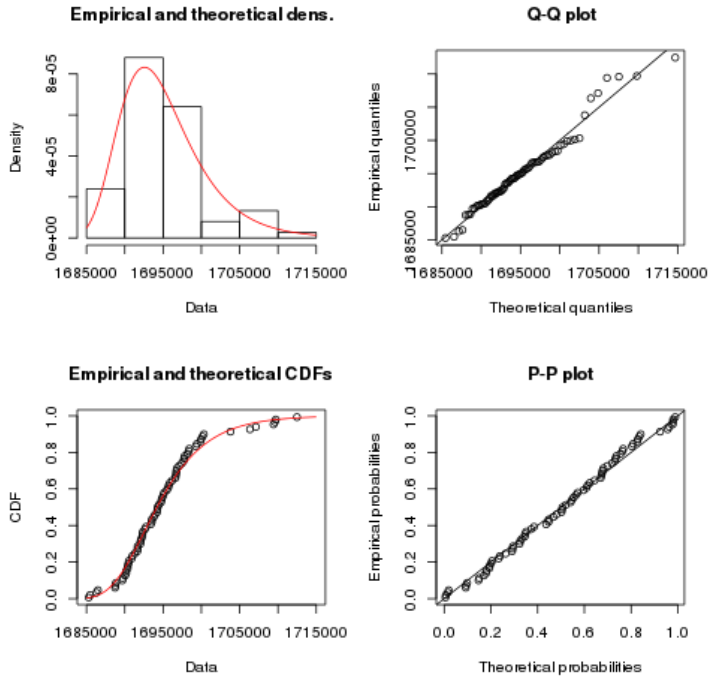

Figure H.4: Fitted Gumbel distribution for extreme completion data for benchmark 5 with 512 byte pages under partial paging

I GUMBEL DISTRIBUTION PLOTS FOR 256 BYTE PARTIAL PAGING (CLOCK)

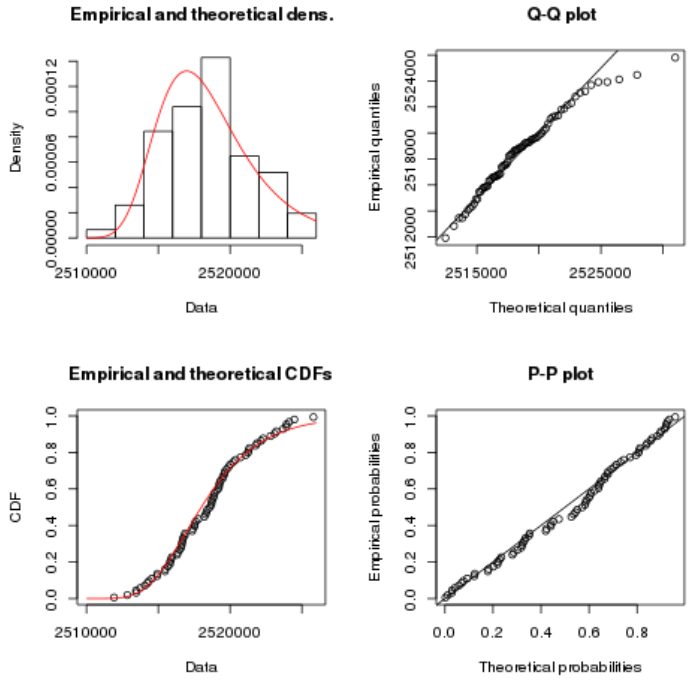

Figure I.1: Gumbel distribution for benchmark 0 with 256-byte pages

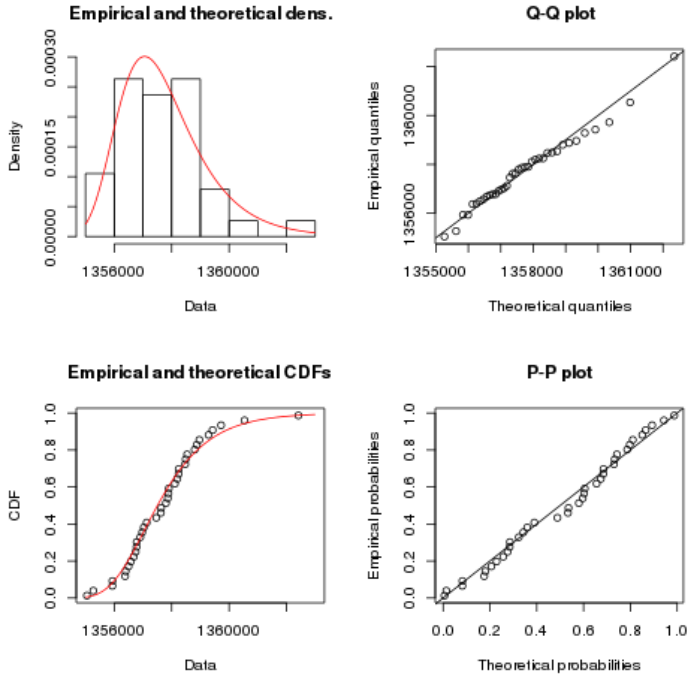

Figure I.2: Gumbel distribution for benchmark 1 with 256-byte pages

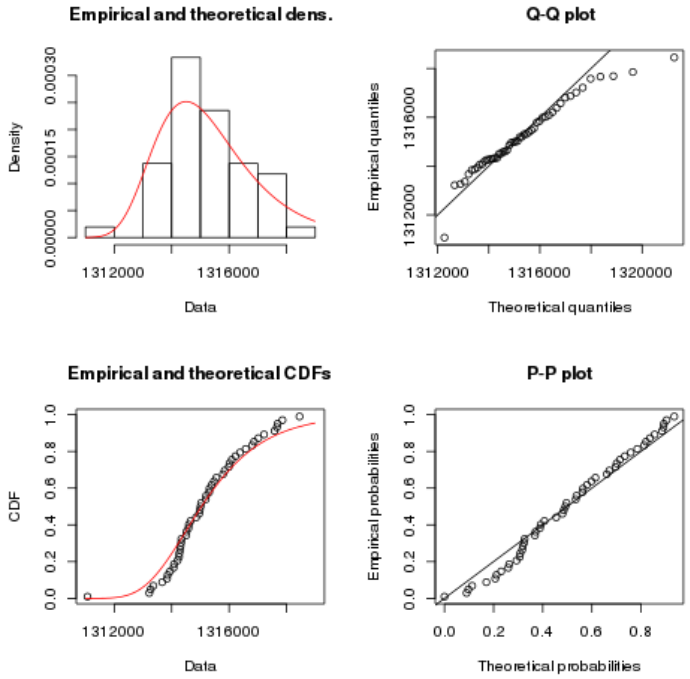

Figure I.3: Gumbel distribution for benchmark 2 with 256-byte pages

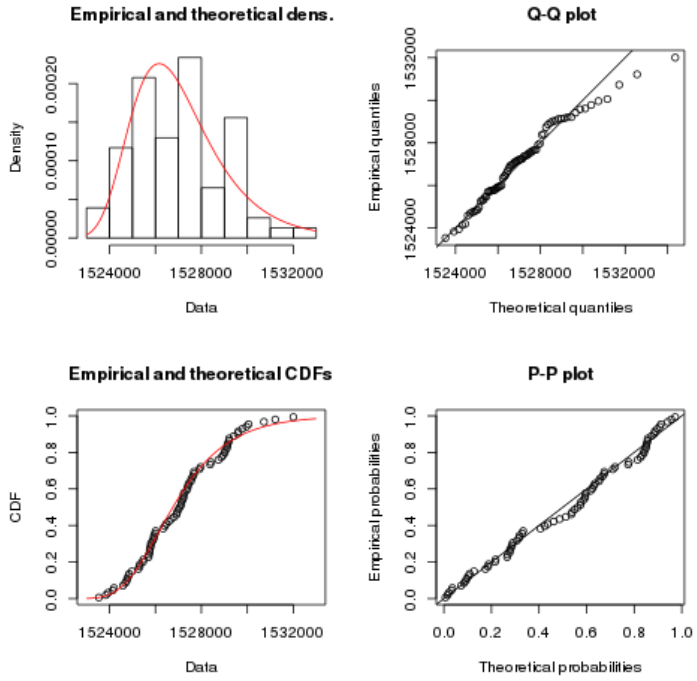

Figure I.4: Gumbel distribution for benchmark 5 with 256-byte pages

J GUMBEL DISTRIBUTION PLOTS FOR 512 BYTE PARTIAL PAGING WITH FIFO

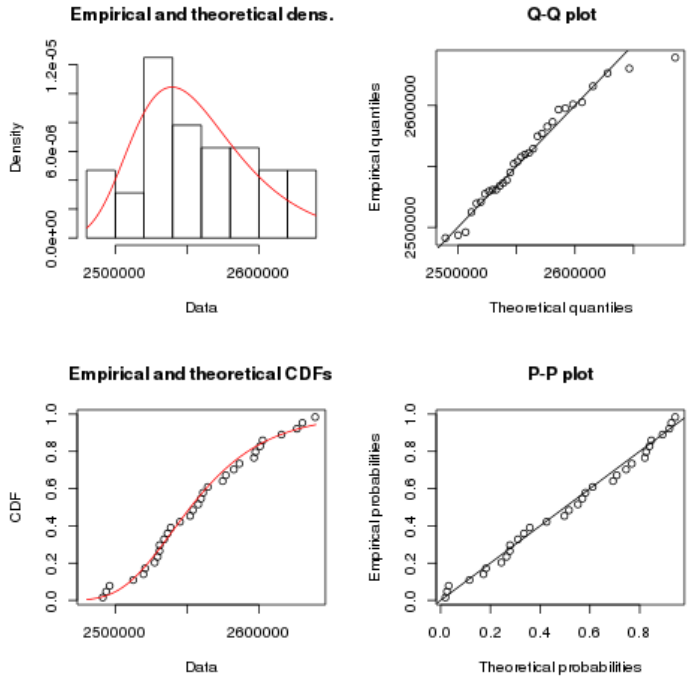

Figure J.1: Fitted Gumbel distribution for FIFO with benchmark 0

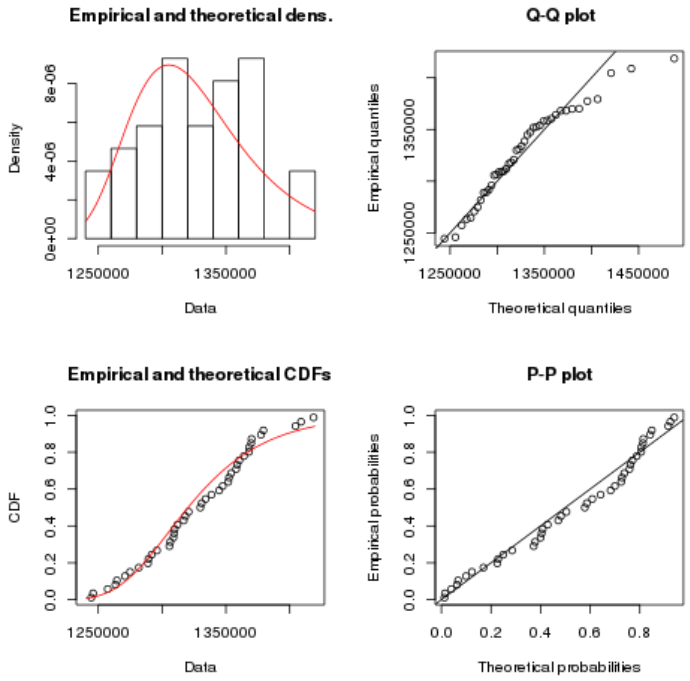

Figure J.2: Fitted Gumbel distribution for FIFO with benchmark 2

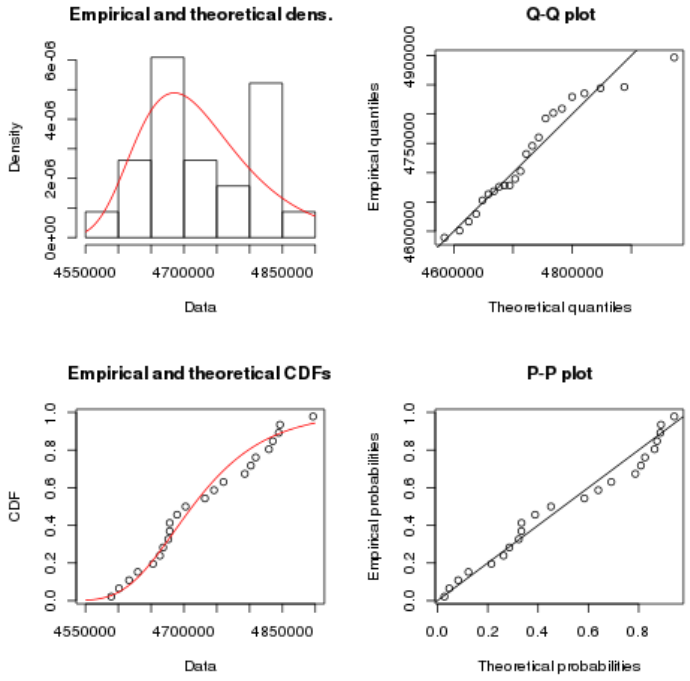

Figure J.3: Fitted Gumbel distribution for FIFO with benchmark 3

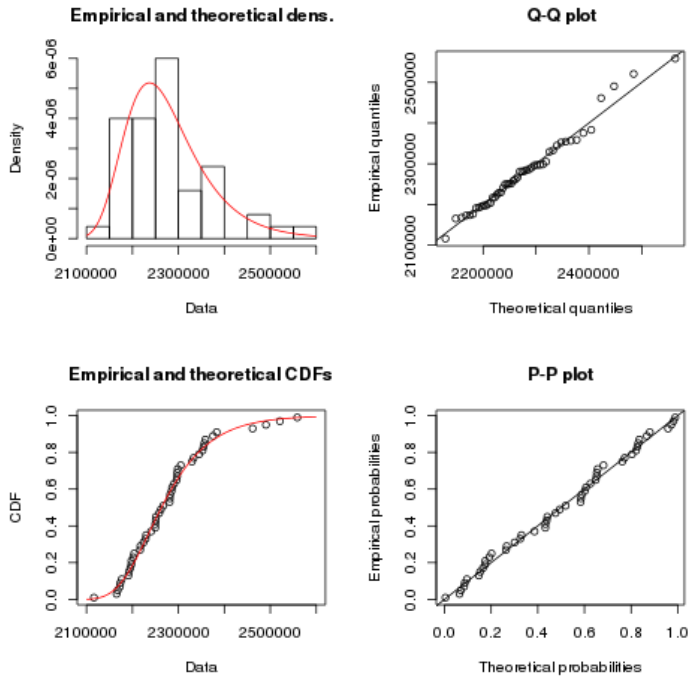

Figure J.4: Fitted Gumbel distribution for FIFO with benchmark 5

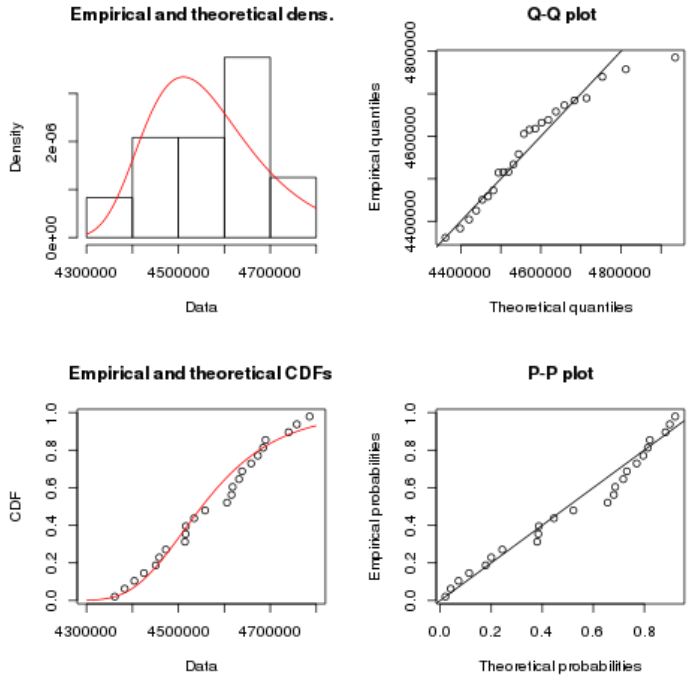

Figure J.5: Fitted Gumbel distribution for FIFO with benchmark 7

BIBLIOGRAPHY

[1] Augmented Dickey-Fuller Test - R documentation. https://www.rdocumentation.org/packages/aTSA/versions/3.1.2/topics/adf.test. (Accessed on 15/12/2019).

[2] Euclidean Algorithm – from Wolfram MathWorld. http://mathworld.wolfram.com/EuclideanAlgorithm.html. (Accessed on 12/01/2019).

[3] Gaussian Elimination Algorithm - Wolfram | Alpha. https://www.wolframalpha.com/input/?i=Gaussian+elimination+algorithm. (Accessed on 11/01/2019).

[4] Roel Aaij, Marianna Fontana, Renaud Le Gac, Emilia Anna Zacharjasz, Rainer Schwemmer, Conor Fitzpatrick, Johannes Albrecht, Lucia Grillo, Tomasz Szumlak, Hang Yin, et al. Upgrade trigger: Biannual performance update. Technical report, CERN, 2017.

[5] Jaume Abella, Carles Hernandez, Eduardo Quiñones, Francisco J Cazorla, Philippa Ryan Conmy, Mikel Azkarate-Askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10. IEEE, 2015.

[6] Adrijean Adriahantenaina, Herve Charlery, Alain Greiner, Laurent Mortiez, and Cesar Albenes Zeferino. SPIN: a scalable, packet switched, on-chip micro-network. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 70–73. IEEE, 2003.

[7] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Computer Architecture, 1988. Conference Proceedings. 15th Annual International Symposium on*, pages 280–289, 30 May-2 June 1988 1988.

[8] T. Agerwala. Systems Trends and their Impact on Future Microprocessor Design. In *Keynote of 35th Annual International Symposium on Microarchitecture*, 2002.

[9] S. T. Allworth and R. N. Zobel. *Introduction to Real-time Software Design (2nd ed.)*. Macmillan Education Ltd, Basingstoke, Hampshire, 1987.

[10] Adrijean Andriahantenaina and Alain Greiner. Micro-network for SoC: Implementation of a 32-port SPIN network. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, page 11128. IEEE Computer Society, 2003.

[11] Krste Asanović and David A Patterson. Instruction sets should be free: The case for RISC-V. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[12] Neil Audsley. Memory architectures for NoC-based real-time mixed criticality systems. *Proc. WMC, RTSS*, pages 37–42, 2013.

[13] Todd Austin, David Blaauw, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Wayne Wolf. Mobile supercomputers. *Computer*, 37(5):81–83, 2004.

[14] Oleg Ivanovich Aven, Edward Grady Coffman, and Yakov Afroimovich Kogan. *Stochastic analysis of computer storage*, volume 38. Springer Science & Business Media, 1987.

[15] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78. ACM, 2002.

[16] Arnab Banerjee, Robert Mullins, and Simon Moore. A power and energy exploration of network-on-chip architectures. In *Proceedings of the First international Symposium on Networks-on-Chip*, pages 163–172. IEEE Computer Society, 2007.

[17] Janibul Bashir, Eldhose Peter, and Smruti R. Sarangi. A survey of on-chip optical interconnects. *ACM Comput. Surv.*, 51(6):115:1–115:34, January 2019.

[18] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 237–248. ACM, 2013.

[19] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[20] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353, June 1969.

[21] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, feb. 2008.

[22] L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *Computer*, 35(1):70–78, jan 2002.

[23] Luca Benini, Alberto Macii, Enrico Macii, and Massimo Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *IEEE Design & Test of Computers*, 17(2):74–85, 2000.

[24] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton University, January 2008.

[25] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of Network-on-chip. *ACM Comput. Surv.*, 38(1), June 2006.

[26] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *Solid-State Circuits Newsletter, IEEE*, 12(1):11–13, 2007.

[27] Terry Bossomaier, Lionel Barnett, Michael Harré, and Joseph T Lizier. *An Introduction to Transfer Entropy*. Springer, 2016.

[28] Daniel Pierre Bovet and Marco Cassetti. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[29] A. Burns and A.J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Addison Wesley, 2001.

[30] Sebastian Canovas-Carrasco, Antonio-Javier Garcia-Sanchez, Felipe Garcia-Sanchez, and Joan Garcia-Haro. Conceptual design of a nano-networking device. *Sensors*, 16(12):2104, 2016.

[31] Francisco J Cazorla, Tullio Vardanega, Eduardo Quiñones, and Jaume Abella. Upper-bounding program execution time with extreme value theory. In *OASIcs-OpenAccess Series in Informatics*, volume 30. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[32] Robert N Charette. This car runs on code. *IEEE Spectrum*, 46(3):3, 2009.

[33] Jin-Hyuck Choi, Jung-Hoon Lee, Seh-Woong Jeong, Shin-Dug Kim, and Charles Weems. A low power TLB structure for embedded systems. *IEEE Computer Architecture Letters*, 1(1):3–3, 2002.

[34] Carsten Clauss, Stefan Lankes, Pablo Reble, and Thomas Bemmerl. New system software for parallel programming models on the Intel SCC many-core processor. *Concurrency and Computation: Practice and Experience*, 2013.

[35] Edward Grady Coffman and Peter J Denning. *Operating systems theory*, volume 973. prentice-Hall Englewood Cliffs, NJ, 1973.

[36] The Qt Company. C++ classes documentation. Technical report, https://doc.qt.io/qt-5/reference-overview.html. (Accessed on 01/12/2019).

[37] Jim E Cooling. Languages for the programming of real-time embedded systems a survey and comparison. *Microprocessors and Microsystems*, 20(2):67–77, 1996.

[38] Fernando J Corbato. A paging experiment with the Multics system. Technical report, Defense Technical Information Center, 1968.

[39] Thomas M Cover and Joy A Thomas. *Elements of Information Theory*, chapter 3, pages 57–70. John Wiley & Sons, second edition, 2006.

[40] David Roxbee Cox and Hilton David Miller. *The theory of stochastic processes*. Chapman and Hall, 1965.

[41] DR Cox and Valerie Isham. The virtual waiting-time and related processes. *Advances in Applied Probability*, 18(2):558–573, 1986.

[42] Silviu S Craciunas, Christoph M Kirsch, Hannes Payer, Ana Sokolova, Horst Stadler, and Robert Staudinger. A compacting real-time memory management system. In *USENIX Annual Technical Conference*, pages 349–362, 2008.

[43] William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 684–689, New York, NY, USA, 2001. ACM.

[44] Asit Dan and Don Towsley. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. *SIGMETRICS Perform. Eval. Rev.*, 18(1):143–152, April 1990.

[45] Dakshina Dasari, Benny Akesson, Vincent Nelis, Muhammad Ali Awan, and Stefan M Petters. Identifying the sources of unpredictability in COTS-based multicore systems. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 39–48. IEEE, 2013.

[46] Giovanni De Micheli and Luca Benini. *Networks on chips: technology and tools*. Morgan Kaufmann, 2006.

[47] P. J. Denning. Working Sets Past and Present. *IEEE Trans. Softw. Eng.*, 6(1):64–84, January 1980.

[48] Peter J. Denning. Thrashing: Its causes and prevention. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 915–922, New York, NY, USA, 1968. ACM.

[49] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[50] Peter J Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[51] Peter J Denning and Ted G Lewis. Exponential Laws of Computing Growth. *Communicdations of the ACM*, 60(1):54–65, 2017.

[52] Simon Andrew Dobson. *An Approach to Scalable Parallel Programming*. PhD thesis, University of York, 1993.

[53] R.G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits. *Proceedings of the IEEE*, 98(2):253–266, Feb 2010.

[54] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, April 2009.

[55] Stewart Edgar and Alan Burns. Statistical analysis of WCET for scheduling. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, pages 215–224. IEEE, 2001.

[56] Stewart Frederick Edgar. *Estimation of worst-case execution time using statistical analysis*. PhD thesis, University of York, 2002.

[57] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *SIGARCH Comput. Archit. News*, 39(3):365–376, June 2011.

[58] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In *OASIcs-OpenAccess Series in Informatics*, volume 55. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[59] Farshad Firouzi, Amir M. Rahmani, K. Mankodiya, M. Badaroglu, G.V. Merrett, P. Wong, and Bahar Farahani. Internet-of-Things and big data for smarter healthcare: From device to architecture, applications and analytics. *Future Generation Computer Systems*, 78:583 – 586, 2018.

[60] Free Software Foundation. GCC, the GNU Compiler Collection. Technical report, https://gcc.gnu.org.

[61] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei Ge, Jian Zhang, Yangang Wang, Chunbo Zhou, and Guangwen Yang. The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences*, 59(7):072001, Jun 2016.

[62] Samuel H. Fuller and Lynette I. Millett, editors. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011.

[63] Simon Gansel. *Konzepte und Mechanismen für die Darstellung von sicherheitskritischen Informationen im Fahrzeug*. PhD thesis, University of Stuttgart, 2017. (Extended English abstract).

[64] Jamie Garside. *Real-Time Prefetching on Shared-Memory Multi-Core Systems*. PhD thesis, University of York, 2015.

[65] Jamie Garside and Neil C Audsley. Investigating Shared Memory Tree Prefetching within Multimedia NoC Architectures. In *Memory Architecture and Organisation Workshop 2013*, Montreal, 2013.

[66] Jamie Garside and Neil C Audsley. Prefetching across a shared memory tree within a Network-on-Chip architecture. In *ISSoC*, pages 1–4, 2013.

[67] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Fuston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on NUMA systems. In *USENIX Annual Technical Conference*, pages 231–242, 2014.

[68] Mark Gebhart, Stephen W Keckler, Brucek Khailany, Ronny Krashinsky, and William J Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 96–106. IEEE Computer Society, 2012.

[69] Francisco Gilabert, Simone Medardoni, Davide Bertozzi, Luca Benini, María Engracia Gomez, Pedro Lopez, and José Duato. Exploring high-dimensional topologies for NoC design through an integrated analysis and synthesis framework. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 107–116. IEEE Computer Society, 2008.

[70] Ruth E. Goldenberg. *Open VMS Alpha Internals and Data Structures: Memory Management*. Digital Press, Newton, MA, USA, 2002.

[71] E González-Estrada and JA Villaseñor. An R package for testing goodness of fit: goft. *Journal of Statistical Computation and Simulation*, 88(4):726–751, 2018.

[72] James R. Goodman. Using cache memory to reduce processor-memory traffic. *SIGARCH Comput. Archit. News*, 11(3):124–131, June 1983.

[73] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414–421, 2005.

[74] David Griffin and Alan Burns. Realism in statistical analysis of worst case execution times. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[75] Adam Grzywaczewski. Training AI for Self-Driving Vehicles: the Challenge of Scale. *Nvidia Parallel for All Blog - https://devblogs.nvidia.com/parallelforall/training-self-driving-vehicles-challenge-scale/*, 2017. (Accessed on 28/10/17).

[76] Jeffery Hansen, Scott Hissam, and Gabriel A Moreno. Statistical-based WCET estimation and validation. In *9th international workshop on worst-case execution time analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[77] Nor Zaidi Haron and Said Hamdioui. Why is CMOS scaling coming to an END? In *2008 3rd International Design and Test Workshop*, pages 98–103. IEEE, 2008.

[78] Donald J. Hatfield. Experiments on page size, program access patterns, and virtual memory performance. *IBM Journal of research and development*, 16(1):58–66, 1972.

[79] Haifeng He, Saumya K Debray, and Gregory R Andrews. The revenge of the overlay: automatic compaction of OS kernel code via on-demand code loading. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 75–83. ACM, 2007.

[80] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Elsevier, revised 1st edition, 2012.

[81] M.D. Hill and M.R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, 2008.

[82] Martin Hirzel. Data layouts for object-oriented programs. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 265–276. ACM, 2007.

[83] Wei Huang, Karthick Rajamani, Mircea R Stan, and Kevin Skadron. Scaling with design constraints: Predicting the future of big chips. *IEEE Micro*, 31(4):16–29, 2011.

[84] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3B: System programming Guide, Part 2*, 2011.

[85] Bruce Jacob. The RiSC-16 Instruction-Set Architecture. *ENEE 446: Digital Computer Design*, 2000.

[86] Akshay Jain, Mahmoud Khairy, and Timothy G. Rogers. A Quantitative Evaluation of Contemporary GPU Simulation Methodology. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2):35:1–35:28, June 2018.

[87] Song Jiang and Xiaodong Zhang. Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems. *Performance Evaluation*, 60(1-4):5–29, 2005.

[88] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[89] Alin Jula and Lawrence Rauchwerger. Two memory allocators that use hints to improve locality. In *Proceedings of the 2009 international symposium on Memory management*, pages 109–118. ACM, 2009.

[90] R. Bodik B. C. Catanzaro J. J. Gebis P. Husbands K. Keutzer D. A. Patterson W. L. Plishker J. Shalf S. W. Williams K. Asanovic and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS- 2006-183, EECS Department, University of California, Berkley, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html, December 2006.

[91] James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM journal of Research and Development*, 49(4.5):589–604, 2005.

[92] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, dec. 2003.

[93] Raimund Kirner and Peter Puschner. Discussion of misconceptions about WCET analysis. In *Proceedings of the 3rd Euromicro International Workshop on WCET Analysis*,

pages 61–64, http://www.vmars.tuwien.ac.at/php/pserver/extern/docdetail.php?DID=1235&viewmode=published&year=2003, 2003.

[94] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Longman, Inc, third edition edition, 1998.

[95] David Lammers. Intel cancels Tejas, moves to dual-core designs. *EETimes*, May 2004.

[96] Stefan Lankes, Pablo Reble, Carsten Clauss, and Oliver Sinnen. The path to MetalSVM: Shared virtual memory for the SCC. In *4th Many-core Applications Research Community (MARC) Symposium*, page 7, 2011.

[97] Jim Larus. Spending Moore's Dividend. Technical report, Microsoft, May 2008.

[98] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[99] Sang-Won Lee, Bongki Moon, and Chanik Park. Advances in flash memory ssd technology for enterprise database applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 863–870, New York, NY, USA, 2009. ACM.

[100] Sang-Won Lee, Bongki Moon, and Chanik Park. Advances in flash memory SSD technology for enterprise database applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 863–870. ACM, 2009.

[101] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH computer architecture news*, 38(3):451–460, 2010.

[102] C.E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *Computers, IEEE Transactions on*, C-34(10):892–901, 1985.

[103] Don S Lemons. *A student's guide to entropy*. Cambridge University Press, 2013.

[104] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, 17(1):91–113, Feb 2018.

[105] C. C. Liu, I. Ganusov, M. Burtscher, and Sandip Tiwari. Bridging the processor-memory performance gap with 3D IC technology. *IEEE Design Test of Computers*, 22(6):556–564, Nov 2005.

[106] Tao Liu and Ralf Huuck. Case study: Static security analysis of the Android Goldfish kernel. In *International Symposium on Formal Methods*, pages 589–592. Springer, 2015.

[107] G. Macher, H. Sporer, R. Berlach, E. Armengaud, and C. Kreiner. Sahara: A security-aware hazard and risk analysis method. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 621–624, March 2015.

[108] Timothy G Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, et al. The 48-core SCC processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[109] Adrian McMenamin. Applying Working Set Heuristics to the Linux Kernel. Master's thesis, Birkbeck College, University of London, 2011. Available at https://cartesianproduct.files.wordpress.com/2011/12/main.pdf (Accessed 12/11/17).

[110] Adrian McMenamin and Neil C Audsley. Partial Paging for Real-Time NoC Systems. In *11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 13–18, 2015.

[111] Paris Mesidis and Leandro Soares Indrusiak. Genetic mapping of hard real-time applications onto noc-based mpsocs—a first approach. In *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–6. IEEE, 2011.

[112] Robert M Metcalfe and David R Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[113] Suzana Milutinovic, Enrico Mezzetti, Jaume Abella, Tullio Vardanega, and Francisco J Cazorla. On uses of extreme value theory fit for industrial-quality WCET analysis. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–6. IEEE, 2017.

[114] Sparsh Mittal and Jeffrey S Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015.

[115] Ethan Mollick. Establishing Moore's Law. *IEEE Ann. Hist. Comput.*, 28(3):62–75, 2006. 1158837.

[116] Trevor Mudge. Power: A First-Class Architectural Design Constraint. *Computer*, 34(4):52–58, April 2001.

[117] S Muthukrishnan, R Karthikeyan, and T Janani. Algorithm and architecture for a low-power content addressable memory based on sparse compression technique. *architecture*, 2(11), 2015.

[118] Jorge L. Navarro. Demystifying response time: Beyond constancy. Technical report, IBM, 2012.

[119] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[120] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 5–14. ACM, 2017.

[121] Nvidia. NVIDIA's next generation CUDA compute architecture: FERMI. *Comput. Syst.*, 26:63–72, 01 2009.

[122] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[123] OVPWorld.org. Ovp guide to using processor models. `http://www.ovpworld.org/documents/OVP_Guide_To_Using_Processor_Models.pdf`. (Accessed on 1/12/19).

[124] Gopal Pandurangan and Eli Upfal. Can entropy characterize performance of online algorithms? In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 727–734. Society for Industrial and Applied Mathematics, 2001.

[125] Hae-woo Park, Kyoungjoo Oh, Soyoung Park, Myoung-min Sim, and Soonhoi Ha. Dynamic code overlay of sdf-modeled programs on low-end embedded systems. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 945–946. European Design and Automation Association, 2006.

[126] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., Burlington, MA, USA, 4th edition, 2009.

[127] Paul S. Peercy. The drive to miniturization. *Nature*, 406(6799):1023–1026, Aug 2000.

[128] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage management in the NVRAM era. *Proceedings of the VLDB Endowment*, 7(2):121–132, 2013.

[129] Roger Penrose. *Cycles of time: an extraordinary new view of the universe*. Random House, 2010.

[130] Steven M Pincus. Approximate entropy as a measure of system complexity. *Proceedings of the National Academy of Sciences*, 88(6):2297–2301, 1991.

[131] Gary Plumbridge, Jack Whitham, and Neil Audsley. Blueshell: A platform for rapid prototyping of multiprocessor nocs and accelerators. *SIGARCH Comput. Archit. News*, 41(5):107–117, June 2014.

[132] Ruxandra Pop and Shashi Kumar. A survey of techniques for mapping and scheduling applications to network on chip systems. *School of Engineering, Jonkoping University, Research Report*, 4(4), 2004.

[133] José C. Principe. *Information Theory, Machine Learning, and Reproducing Kernel Hilbert Spaces*, pages 1–45. Springer New York, New York, NY, 2010.

[134] Isabelle Puaut and Damien Hardy. Predictable paging in real-time systems: A compiler approach. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 169–178. IEEE, 2007.

[135] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, pages 1–6. IEEE, 2007.

[136] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: the next computing revolution. In *Design Automation Conference*, pages 731–736. IEEE, 2010.

[137] Anand Ranganathan and Roy H Campbell. What is the complexity of a distributed computing system? *Complexity*, 12(6):37–45, 2007.

[138] Jan Reineke and Daniel Grund. Sensitivity of cache replacement policies. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):42, 2013.

[139] Rolf-Dieter Reiss and Michael Thomas. *Statistical Analysis of Extreme Values: With Applications to Insurance, Finance, Hydrology and Other Fields*. Springer Science & Business Media, 2007.

[140] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K John. Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB. *ACM SIGARCH Computer Architecture News*, 45(2):469–480, 2017.

[141] Ahmad Samih, Ren Wang, Anil Krishna, Christian Maciocco, Charlie Tai, and Yan Solihin. Energy-efficient interconnect via router parking. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 508–519. IEEE, 2013.

[142] Luca Santinelli, Jérôme Morio, Guillaume Dufour, and Damien Jacquemart. On the sustainability of the extreme value theory for WCET estimation. In *OASIcs-OpenAccess Series in Informatics*, volume 39. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[143] Curt Schimmel. *UNIX systems for modern architectures: symmetric multiprocessing and caching for kernel programmers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[144] Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparsø. A time-predictable memory network-on-chip. In *OASIcs-OpenAccess Series in Informatics*, volume 39. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[145] Carlo H Séquin and David A Patterson. Design and Implementation of RISC I. Technical report, Computer Science Division, University of California, 1982.

[146] Azam Seyedi, Vasileios Karakostas, Stefan Cosemans, Adrian Cristal, Mario Nemirovsky, and Osman Unsal. NEMsCAM: A novel CAM cell based on nano-electro-mechanical switch and CMOS for energy efficient TLBs. In *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on*, pages 51–56. IEEE, 2015.

[147] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[148] Karila Palma Silva, Luis Fernando Arcaro, and Romulo Silva de Oliveira. On Using GEV or Gumbel Models When Applying EVT for Probabilistic WCET Estimation. In *Real-Time Systems Symposium (RTSS), 2017 IEEE*, pages 220–230. IEEE, 2017.

[149] Jeffrey R Spirn. *Program behavior: models and measurements*. Elsevier Science Inc., 1977.

[150] William Stallings. *Operating Systems: Internals and Design Principles, 6/E*. Pearson Prentice Hall, 2009.

[151] Vivy Suhendra, Chandrashekar Raghavan, and Tulika Mitra. Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 401–410, New York, NY, USA, 2006. ACM.

[152] K. Swaminathan, E. Kultursay, V. Saripalli, V. Narayanan, M. T. Kandemir, and S. Datta. Steep-slope devices: From dark to dim silicon. *IEEE Micro*, 33(5):50–59, Sept 2013.

[153] Michael B Taylor. A landscape of the new dark silicon design regime. *IEEE Micro*, 33(5):8–19, 2013.

[154] Rollins Turner and Henry Levy. Segmented FIFO Page Replacement. *SIGMETRICS Perform. Eval. Rev.*, 10(3):48–51, September 1981.

[155] Yash Ukidave, Amir Kavyan Ziabari, Perhaad Mistry, Gunar Schirner, and David Kaeli. Quantifying the energy efficiency of FFT on heterogeneous platforms. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 235–244. IEEE, 2013.

[156] András Vajda. *Programming many-core chips*. Springer Science & Business Media, 2011.

[157] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H Loh, and Abhishek Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 161–171. IEEE, 2016.

[158] James Vincent. 99.6 percent of new smartphones run Android or iOS https://www.theverge.com/2017/2/16/14634656/android-ios-market-share-blackberry-2016, 2017.

[159] J. Wang, Q. Wang, L. Jiang, C. Li, X. Liang, and N. Jing. IBOM: An Integrated and Balanced On-Chip Memory for High Performance GPGPUs. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2017.

[160] A Waterman and Y Lee. Spike, a RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim. (Accessed 1 December 2019).

[161] Pinchas Weisberg and Yair Wiseman. Using 4kb page size for virtual memory is obsolete. In *Information Reuse & Integration, 2009. IRI'09. IEEE International Conference on*, pages 262–265. IEEE, 2009.

[162] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5):15–31, 2007.

[163] Jack Whitham and Neil Audsley. The scratchpad memory management unit for microblaze: Implementation, testing, and case study. *University of York, Tech. Rep. YCS-2009-439*, 2009.

[164] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[165] Maurice V Wilkes. The memory gap and the future of high performance memories. *ACM SIGARCH Computer Architecture News*, 29(1):2–7, 2001.

[166] Carol Woody. Using Malware Analysis to Reduce Design Weaknesses. Technical report, Carnegie Mellon University, https://samate.nist.gov/docs/SwMM-RSV2016/SwMM-RSV_13_UsingMalwareAnalysis_CWoody.pdf, 2016. (Accessed 18 November 2019).

[167] Terry Tao Ye, Luca Benini, and Giovanni De Micheli. Packetized on-chip interconnect communication analysis for MPSoC. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 344–349. IEEE, 2003.

[168] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on parallel and distributed systems*, 15(6):505–519, 2004.