Page size exploration for RISC-V systems: the case for HPC

Eduardo Tomasi Ribeiro, César Fuguet, Christian Fabre, Frédéric Pétrot

Page size exploration for RISC-V systems: the case for HPC

Eduardo Tomasi [*†], César Fuguet [*], Christian Fabre [*], Frédéric Pétrot [†]
[*] Univ. Grenoble Alpes, CEA, List, F-38000, Grenoble
{eduardo.tomasiribeiro, cesar.fuguettortolero, christian.fabre1}@cea.fr
[†] Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMA, 38000, Grenoble, France
frederic.petrot@univ-grenoble-alpes.fr

***Abstract*—The page size used for virtual to physical address translation has globally not changed since the late 1960's: the IBM 360, circa 1964, already had 4 KiB pages. This 4 KiB page size has proven to be incredibly robust given the changes in processor architectures, workloads behavior, memory size, and access patterns. However, with 64-bit registers, 57-bit virtual addresses, and increasingly bigger physical memories, we have to ask ourselves whether 4 KiB is still an adequate page size for modern workloads on modern machines. Inherently, the page size has an influence on (a) the miss rate of the translation lookaside buffer, the cache that contains the recently used virtual to physical translations, and (b) the memory allocated by the system versus the memory actually used by a process. The page size also constraints some microarchitectural choices, such as cache design, which impacts the overall performance and energy efficiency. We focus more particularly on High Performance Computing (HPC) applications because they are extremely demanding in terms of memory, and are indicative of future general-purpose needs.**

In this paper, we empirically study the evolution of the miss rate and memory occupancy with respect to the page size, and conclude that a page size of 32 KiB is better suited for current HPC systems. We also propose a page table scheme for RISC-V-based HPC systems based on our observations and discuss its benefits.

I. Introduction

On-chip computation has benefited from Dennard scaling and Moore's law for decades, seeing improvements that led to multicore architectures becoming the standard [1]. Matching workloads have gone parallel, and are now able to handle huge amounts of data. Meanwhile, the page size used for virtual to physical address translation has remained basically the same: 4 KiB.

In 1964, Amdahl *et al.* [2] suggested 4 KiB pages for easy program relocation in the IBM 360. Not long after, 1965, Comfort [3] formalized the concept of virtual memory and introduced a set of associative registers to cache the recent translations, that we now know as a Translation Lookaside Buffer (TLB). Since then, this 4 KiB size seems to have been unanimously adopted, as it is the one used in virtually all processors. Notable exceptions are the late Alpha 21x64 and Sparc64 V6 to V9, with 8 KiB page size, and more recently and more successfully the Apple ARM processors that have 16 KiB pages since the A9 and M1 [4], [5]. One would expect that the page size would be chosen similarly to cache-block length, which has been empirically determined using agreed-upon benchmarks. However, we are not aware of any formal publication reporting the reasons behind 8 KiB or 16 KiB pages.

Processors issue virtual addresses, which are translated into actual physical addresses. These translations occur at the granularity of the page size. A TLB is used to cache the recent translations, and, as any cache, implements a victim selection algorithm to evict an entry in case of conflict. The size of a page has a very important impact on performance: the bigger the page, the fewer the number of entries needed in the TLB, hence less conflicts. It also has an influence on the memory that is allocated compared to the memory that is actually needed. Indeed, memory is allocated at page granularity, so using 32 KiB pages to store a few bytes results in much more wasted memory than with 4 KiB pages.

Establishing the best page size in the general case makes little sense, as the trade-offs for embedded devices are different from those for servers, for instance. To narrow the scope, this paper focuses on the study of HPC systems, for three reasons:

1) HPC benchmarks are regular and work on large data sets, although across benchmarks their behaviour is heterogeneous [6],
2) They are, in part, executed on a local cluster on top of a coherent shared memory library, which heavily uses the memory system,
3) Processors in HPC clusters are already specialized, so relatively small modifications leading to improvements in performance can be adopted.

Most of the current top supercomputers are x86-based, and only a few years ago have we seen an ARM-based supercomputer take the top position in the Top500 for the first time [7], nearly a decade after ARM's adoption in commodity computers. Similarly, although RISC-V was conceived nearly a decade ago, there are currently no sufficiently high-performance micro-architectures to allow for its adoption in HPC. Still, many commercial RISC-V processors have already been announced for general-purpose applications, such as SiFive's P870 and Ventana's Veyron V2 processors. A few prototypes can also be found that show the feasibility and viability of RISC-V processors in the HPC context, e.g. [8], [9], or Monte Cimone, which is the first fully operational RISC-V-based cluster supporting a complete software stack for HPC [10]. As we expect this Instruction Set Architecture (ISA) to find its way to the Top500 in the foreseeable future, we think it is a great opening for revisiting some basic architectural choices, namely the page size.

To explore which is the more suitable page size for HPC systems, we propose to employ fast functional simulation and binary instrumentation to run multicore HPC benchmarks and feed a TLB simulator. Based on the simulation results and the trade-offs at hand, we define a proper page size for the type of workload we are interested in.

Even though this is not a novel subject, to the best of our knowledge no thorough study has been published on this topic for supercomputers. With Moore's law slowdown, the switch to more specialized multi-core processors [1], and the first draft of a 128-bit architecture emerging within the RISC-V space [11], we think it is time to reassess if 4 KiB is still an adequate page size for modern and future workloads.

The main contributions of this paper are: (1) An empirical approach to gather TLB statistics on simulated multicore machines, (2) The definition of a parametric method to determine, according to TLB- and memory-related criteria, the best page size for one or a set of

benchmarks, (3) Experimental evidence to recommend a page size for HPC systems.

II. Motivation and State-of-the-Art

Virtual addresses are a useful Operating System (OS) abstraction that allows each process to have its own isolated address space. However, the OS has to maintain a coherent mapping between the virtual and physical addresses (including when a process' memory is paged-out) and, more particularly, it adds a new level of indirection as, at each memory access, virtual addresses have to be translated into physical addresses. Given the throughput of data and instruction memory accesses, this translation is hardware assisted, and most processors include a page-walker that traverses the page table to find the translation, as well as a TLB to cache recent results.

A TLB has four important parameters that affect its performance: the number of entries, the level of associativity, the replacement or victim selection algorithm and the page size [12]. The impact of the number of entries is straightforward: the more entries, the more translations it can store. Depending on the TLB's level of associativity, a memory block can only occupy a single entry of the TLB (direct-mapped), any of the TLB entries (fully associative), or any entry in a set (set-associative). When a translation entry cannot be mapped to an empty entry, the replacement algorithm chooses an entry to be replaced by the new one. These parameters are important to reduce the miss rate, i.e. the frequency with which the entry is not found in the TLB and has to be searched for in main memory. Reducing the miss rate results in a reduction of the average translation latency. The page size, however, is directly linked to the number of entries, and is crucial to minimize the translation overhead: the bigger the page, the fewer the entries needed. Nonetheless, if pages are too big, it might result in wasteful memory bloat: once it is accessed, the whole page has to be loaded into memory, even though only part of it might be effectively used. In addition, pages might have to be swapped on disk, and in that situation copying void data is both useless and costly. There is, in consequence, a trade-off between miss rate and memory usage.

4 KiB has been the standard page size for decades, even though many solutions to reduce the total number of pages for a given process, and thus the TLB occupancy, have been proposed and implemented. For instance, modern OSs support superpages (e.g. Linux, FreeBSD, Windows, ...), which is a memory management technique that groups contiguous smaller pages into a single and bigger page (usually 2 MB), reducing the number of entries needed in the TLB. According to Zhu et al. [13], different implementations of superpage management exhibit very different performance characteristics in terms of runtime and memory consumption. This mechanism puts a lot of pressure on the OS, which has to keep track of available memory and move allocated pages to reduce fragmentation and allow for superpage allocation.

In ARM 64-bit processors, the Memory Management Unit (MMU) supports contiguous bits, which indicates that an entry belongs to a set of 16 contiguously mapped entries that can be cached in a single TLB entry instead of 16. The same mechanism is supported on RISC-V, with the Naturally-Aligned Power-Of-Two (NAPOT) Translation Continuity Extension (Svnapot) [14]. It can be extended to support up to 64 KiB contiguous page regions. To the best of our knowledge, there is no publication reporting the reasons behind this number.

The latest publication we found on this subject dates back almost 10 years. Weisberg and Wiseman [15] state that virtual memory systems should use pages larger than 4 KiB. By comparing the total number of TLB misses and overall memory usage of single-threaded benchmarks, they argue that for large data objects, a better page size would be 256 KiB, whereas for instructions and small data objects, 16 KiB would be a better fit. Although it is important to quantify the total number of misses and memory usage, we think it does not give a good view of the overall execution of an application. This is because, even though the number of misses reduces with larger pages, it might be sufficiently low when comparing to the total number of TLB accesses, hence having a low impact on the performance. For that reason, our method uses the rate between the number of misses and the total number of memory accesses (i.e. the miss rate) and the rate between the memory that is actually used by the benchmark and the total memory usage (i.e. the memory bloat rate).

TABLE I: Fujitsu A64FX TLB specifications

		Association method	Number of entries	Replacement algorithm
Data TLB	L1	Fully associative	16	FIFO
	L2	4-way	1 024	LRU
Instruction TLB	L1	Fully associative	16	FIFO
	L2	4-way	1 024	LRU

Furthermore, as we focus on HPC systems, we opt to use highly parallel benchmarks with large data sets. Manocha et al. [16] study the implications of superpage management on graph analytics, which presents very irregular memory access patterns. They conclude that superpages can indeed reduce address translation overheads in ideal scenarios. In less ideal scenarios, however, this reduction is highly impacted, specially because of memory fragmentation due to the huge size of the pages. They show that, with a better superpage management, this impact can be mitigated. However, for this same reason, we chose not to test pages bigger than 256 KiB. Superpages can be used regardless of the minimal size of the page, making it a useful approach anyhow.

There are also techniques that reduce translation latency through speculation, such as CoPTA [17]. It achieves an average address prediction accuracy of 82%, which improves overall performance. This solution could also be used with larger pages. They show that overheads caused by TLB misses can take up to 40% of total execution time, and that the Linux memory allocator tends to map contiguous virtual pages to contiguous physical pages. This gives credance to our claim that larger pages could improve TLB and overall performance.

III. Methods

A. Reference TLB Architecture

As a reference, we chose the specification of the Fujitsu's A64FX TLBs [18]. The A64FX is a 48-core 64-bit ARM processor designed for massively parallel computing used in the Fugaku supercomputer, which held the top position in the Top500 for two years [7]. Each core has two-level split instruction and data TLBs, each with their own specifications given in Table I.

In order to compare the TLB performance with different standard page sizes, we use two metrics:

- **Miss rate:** this is the ratio of the number of accesses that do not find a valid translation in the TLB over the total number of TLB accesses. Misses incur performance degradation as they require accessing lower-level memories to find the translation. As we work with multiple cores, hence multiple TLBs, we take the mean miss rate of all TLBs. Using a second level (L2) TLB

can help reduce this latency, so we will study its impact on both levels.

- **Memory bloat rate:** memory is reclaimed by the OS at page granularity, while actual use in programs is, at worse, at byte granularity. Therefore, increasing the page size might increase the total memory needed for a program execution while the memory that is effectively used by the program is constant, regardless of page size. The ratio of both gives information on wasted, unusable memory, that we call memory bloat.

By evaluating these two metrics over a set of HPC programs, we can determine a page size in line with the trade-off between reducing the TLB miss rate and increasing the memory bloat.

B. Functional ISA and TLB Simulators

We use the QEMU [19] cross-ISA simulator to make these measurements and validate our hypothesis. We trace all memory accesses, for both data and instructions, and feed a TLB simulator that can explore different configurations concurrently. We simulate RISC-V applications on a x86-64 host machine. Both the plugin used for tracing memory accesses in QEMU and the TLB simulator are open-source and available in [20], [21].

The simulated system may contain as many target Central Processing Units (CPUs) as needed, which are also known as Virtual CPUs (vCPUs). QEMU scales well with the number of cores on Symmetric Multiprocessing (SMP) host machines as long as the number of vCPUs is less than or equal to the number of CPUs on the host machine [22], [23]. It also supports binary instrumentation through the use of plugins [24]. This infrastructure allows for callback functions to be called on specific events, such as the execution of an instruction or a memory access. These functions are provided with information such as the virtual and physical addresses of the memory accesses, or the address and size of the instruction being executed. We exploit these callbacks to send this data to our simulator, a separate process, which will reproduce the TLB operation and output the metrics we want.

We use QEMU's user-mode simulation, which simulates an application on top of the host machine's OS. This means that the simulator provides only information concerning the running application.

The parameters of the simulations are the following:

1) Number of cores/threads: 32, 64 and 96 cores
2) Page size: from 4 KiB to 256 KiB
3) Multi-core benchmarks: NAS Parallel Benchmarks (NPB), PARSEC and SPLASH3

We run the benchmarks with very big datasets, requiring up to several giga-bytes, which is representative of real HPC applications. They are compiled with OpenMP or pthreads, according to the available implementation. The NPB benchmarks [25] are run using the class B or C dataset. PARSEC [26] and SPLASH [27] are run with the native or large inputs.

C. Modeling approach

Our experiments will produce two plots (miss rate and memory bloat) as a function of the page size $p \in P$ where $P = \{2^{12+l} \mid l \in \mathbb{N} \wedge l \leq 6\}$. We use them to define a cost function $J_{n,b}(p)$ computed as a weighted sum of the miss rate $mr_{n,b}$ and the memory bloat $mb_{n,b}$ for a given benchmark b and number of cores n, as a function of the page size p. This function is defined in Eq. 1, for which mr, mb and $w \in [0, 1]$. The weight w can be chosen according to the importance one wants to give to one or other metric.

$$J_{n,b}(p) = w \cdot mr_{n,b}(p) + (1 - w) \cdot mb_{n,b}(p) \tag{1}$$

We define $p^*_{n,b}$ as the page size that minimizes Eq. 1 for a given n and b (Eq. 2), and p^* as the size that minimizes the cost function for all values of n and b.

$$p^*_{n,b} = \arg\min_{p \in P} J_{n,b}(p) = \{p \mid J_{n,b}(p) = \min_{\pi \in P} J_{n,b}(\pi)\} \tag{2}$$

Obtaining a single p^* value for all benchmarks and number of cores can be done in many different ways. We opt for the one that minimizes the sum of all cost functions. First, the cost functions are normalized so that they all have the same weight on the summation. This leads to Eq. 3, in which $\max(J_{n,b})$ is the highest cost value for a given benchmark and number of cores. We then sum all normalized cost functions, and apply $\arg\min$ to this result, which gives Eq. 4.

$$\hat{J}_{n,b}(p) = \frac{J_{n,b}(p)}{\max(J_{n,b})} \tag{3}$$

$$p^* = \arg\min_{p \in P} \sum_{b \in \{\text{benchmarks}\}} \sum_{n \in \{\text{cores}\}} \hat{J}_{n,b}(p) \tag{4}$$

IV. Results

In this section, we evaluate how increasing the page size affects the miss rate and memory usage in both data and instruction TLBs.

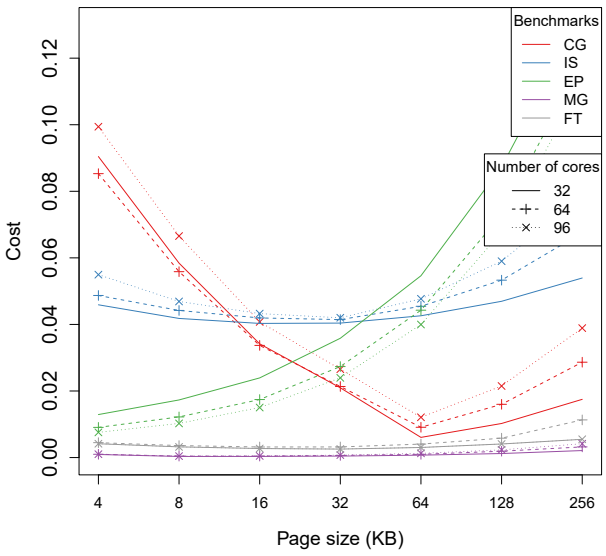
A. Data TLB

Fig. 1 plots the cost function (Eq. 1) of the L1 data TLB for all benchmarks, for 32, 64 and 96 cores, with $w = 0.5$, i.e. with both miss rate and memory bloat functions equally prioritized. Looking at each benchmark separately, we note that, in some cases, by increasing the page size, the miss rate reduces faster than the memory bloat increases. This explains the initial downward trend of some of the cost curves, such as for the Conjugate Gradient (CG) and Integer Sort (IS). These are the benchmarks that would greatly benefit from a bigger page size, but only up to a certain point, since the cost rises again after a certain page size. For some benchmarks, such as Fluidanimate, the cost function is almost constant. This does not mean larger pages do not have an impact on its performance, but rather that the miss rate and the memory bloat vary almost equally, which is acceptable if memory is not a crucial resource. In fact, going from 4 KiB to 32 KiB, for instance, reduces the miss rate on Fluidanimate by $5\times$. For other benchmarks, however, the cost is always increasing, so the impact of the page size on the memory is higher than on the miss rate, and would not benefit as much from a bigger page size, as their lowest cost is at 4 KiB.

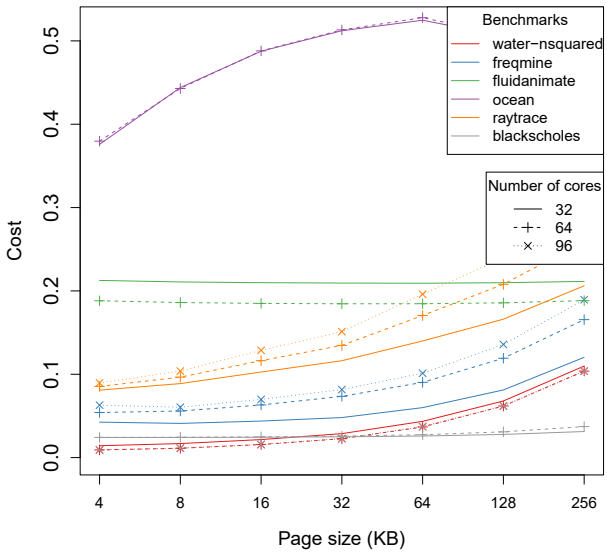
We can also note that, for most cases, increasing the number of cores increases the cost. By looking at it in more details, we observe that both the miss rate and the memory bloat increase in these cases, because more pages are required. This is probably due to OpenMP implementation and data initialization, since this augmentation is virtually only seen in the first core.

Fig. 1a and 1b alone make it difficult to take a decision on the page size that minimizes the cost. However, they give an outlook of the trade-off between the miss rate and the memory bloat when changing the page size for each benchmark. Table II compares the page size that minimizes the cost for each case. Although there are some discrepancies, NPB benchmarks clearly would benefit more from larger pages than the PARSEC and SPLASH benchmarks. This confirms that the ideal page size depends indeed on the application.

As defined in Eq. 4, p^* is the page size that reduces the summation of the costs for all sets of benchmarks and cores, which is shown in

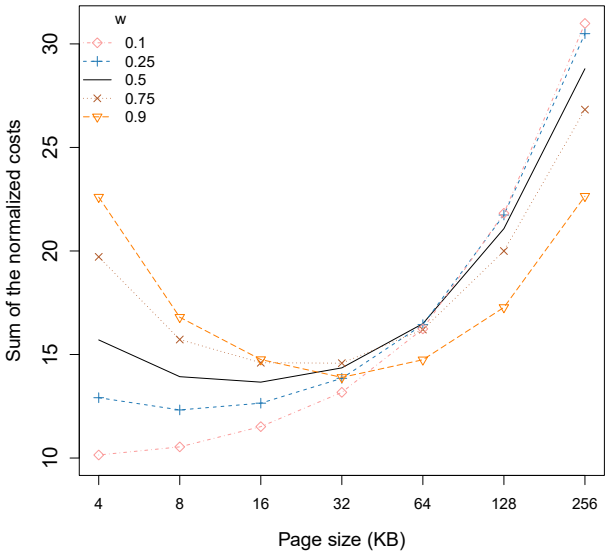


(a) NAS Parallel Benchmarks (NPB)

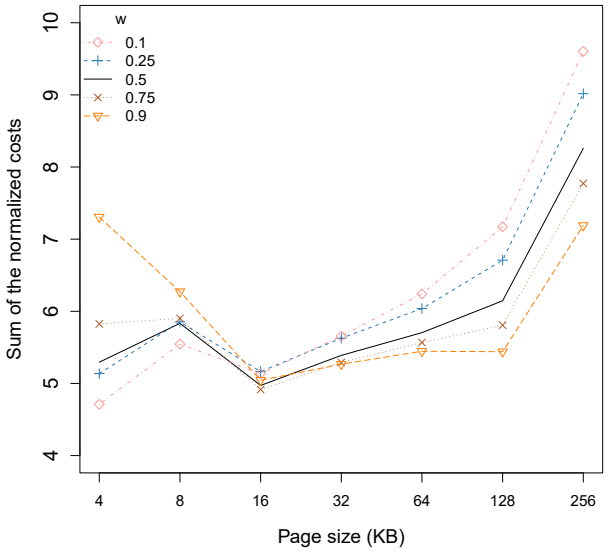


(b) PARSEC and SPLASH

Fig. 1: L1 Data TLB costs for all benchmarks and number of cores with $w = 0.5$



(a) L1 Data TLB



(b) L2 Data TLB

Fig. 2: Results for the sum of the normalized cost of all benchmarks for all number of CPU cores.

TABLE II: Best page size ($p^*_{n,b}$) for $w = 0.5$

Benchmark (b)	$p^*_{n,b}$ (KiB)		
(n)	32 cores	64 cores	96 cores
Conjugate Gradient (CG)	64	64	64
Integer Sort (IS)	32	32	32
Fourier Transform (FT)	32	16	32
Multi-Grid (MG)	8	8	8
Embarassingly Parallel (EP)	4	4	4
Blackscholes	4	4	4
Freqmine	8	4	8
Fluidanimate	64	32	-
Raytrace	4	4	4
Ocean	4	4	-
Water-nsquared	4	4	4

Fig. 2a for five different values of w. When prioritizing memory, i.e. $w < 0.5$, p^* would be either 4 KiB or 8 KiB. Likewise, if the miss rate reduction is prioritized, with $w > 0.5$, this page size becomes 32 KiB. Finally, if both metrics are equally prioritized, i.e. $w = 0.5$, the page size p^* is 16 KiB.

We can note that, after a certain page size, the value of the cost only increases, for all values of w. This shows that, even if miss rate reduction is overprioritized, choosing a too big page size is not a good idea, as its impact in memory usage becomes too significant.

Regarding the L2 data TLB, we observed that, in some cases, the miss rate increases with bigger page sizes, which seems counterintuitive. This is due to the fact that, after a certain size, there are so few misses on the L1 TLB, that the L2 TLB is barely accessed. For instance, in some benchmarks, when using 256 KiB pages, almost all of the most frequently used pages fit on the L1 TLB. However, the first access to a page causes a miss in both L1 and L2 TLBs,

so almost all accesses to the L2 TLB cause a miss, resulting in a miss rate close to 100%. To avoid these cases, we ignore the misses caused by the first access to a page. Applying our method on the L2 data TLB results in a p^* of 16 KiB for $w \geq 0.5$, and 4 KiB for $w < 0.5$, as can be seen in Fig. 2b.

B. Instruction TLB

As for the L1 instruction TLB, we realized that the miss rate is already incredibly low with 4 KiB pages, with a mean of 0.002%. These benchmarks have a small number of instructions that repeat quite often and that fit in a few pages. Even though there is a reduction in the miss rate when increasing the page size, it is insignificant when compared to the memory bloat it incurs.

The application of our method on the L1 instruction TLB concludes that the page size p^* for instructions is 4 KiB for the five w values. However, even though bigger page sizes increase memory bloat, memory used for the instructions of these benchmarks is quite low, in the order of a few hundred kilo-bytes. The L2 instruction TLB is nearly unused for the set of benchmarks, and its miss rate is insignificant.

C. Limitations

Our measurement methodology has some limitations that we discuss below.

The first one concerns the fact that we run QEMU in user-mode. This ignores the execution of the OS, which might slightly alter the TLB content as it accesses different memory zones, hence increasing the miss rate.

The second one is that we ignore OS decisions regarding remapping, page promotion, and a few other mechanisms. This is due to the way we gather our statistics: we just run addresses through our simulator. Although there is some loss in fidelity, Badaroux et al. [28] have shown that statistics gathered by QEMU in full-system mode (i.e., both OS and user applications are simulated) depend on the execution time of the binary instrumentation. In addition, HPC benchmarks very often make little use of the OS. So, overall, we believe that neglecting these OS mechanisms does not undermine our conclusions.

The third limitation is intercluster communication. We limit ourselves to a cluster of up to 96 CPUs sharing memory, and make the assumption that we can ignore the effects of the intercluster network accesses. As properly designed HPC programs tend to avoid intercluster communication, we think that network accesses can be ignored for the kind of metrics we are estimating.

Finally, the fact that we track memory accesses instead of memory allocation can also be seen as a limitation. Because of this, we ignore metadata and other allocated chunks of data that are never directly accessed by the application. Even though this possibly results in an overestimation of the memory bloat, this data should be constant for every page size, and should not impact the choice of the best size through the cost function.

V. Discussion

Trade-offs when choosing the page size: Page size has an impact on both TLB miss rate and memory bloat, and its extent depends on the application. Still, using the method proposed in Section III-C, we conclude that if we prioritise miss rate over memory bloat, which is reasonable for HPC systems, 32 KiB pages are the most suitable on average for the set of HPC benchmarks considered in this work.

Very common HPC kernels such as Conjugate Gradient (CG), which is extensively used in scientific applications, and Integer Sort (IS), are examples of applications where the increase of the page size significantly reduces the TLB miss rate (going from 4 KiB to 32 KiB pages leads to a reduction of $5.08\times$ and $1.36\times$ for CG and IS, respectively). These memory intensive benchmarks perform irregular memory accesses with address offsets bigger than 4 KiB. The increased page size helps ensure that more of these irregular accesses fall inside the same page. For more regular benchmarks, however, the benefit of using larger pages is not as significant, or might even be nonexistent. In these latter cases, the TLB miss rate is already low enough with 4 KiB pages and, even though larger pages reduce the miss rate, it is not as significant as the increase in memory bloat.

TABLE III: Minimum, maximum, average and normalized average miss rate and memory bloat for all page sizes

Page size (KiB)	Miss rate				Memory bloat			
	min	max	avg	avg norm	min	max	avg	avg norm
4	0.033%	19.5%	3.9%	1	0.02%	67%	9%	1
8	0.013%	12.7%	2.9%	0.74	0.03%	80%	11%	1.15
16	0.004%	8.7%	2.2%	0.56	0.05%	89%	13%	1.30
32	**0.002%**	**8.7%**	**1.7%**	**0.44**	**0.08%**	**94%**	**14%**	**1.44**
64	0.001%	8.6%	1.3%	0.34	0.14%	97%	16%	1.66
128	0.000%	7.3%	1.0%	0.26	0.24%	97%	19%	1.95
256	0.000%	7.3%	0.9%	0.23	0.42%	97%	24%	2.41

Table III gives the minimum, maximum and average miss rate and memory bloat for the considered benchmarks as a function of the page size. It also shows the average values normalized with respect to the 4 KiB case. For 32 KiB pages, there is an average TLB miss rate reduction of $2.27\times$ for an average memory bloat increase of $1.44\times$. Beyond this page size, the increase of the memory bloat is higher than the miss rate reduction.

We also compared the miss rate using different page sizes with different TLB configurations. If we increase the page size while reducing the number of TLB entries, so as to keep the amount of addressable memory constant, we expect to keep roughly the same miss rate. For a fully associative TLB, we tested the two cases of 64 entries/4 KiB pages and 16 entries/16 KiB pages. The results show that, in the second case, the miss rate is slightly lower. For 4 KiB pages, the average miss rate is 2.82%, whereas with 16 KiB pages the miss rate is 2.2%. In practice, this means that using larger pages can also translate to a reduction in the TLBs size, resulting in lower chip area while maintaining the same performance.

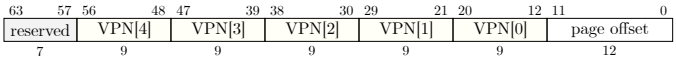
Microarchitecture level implications: Increasing the page size can also benefit the processor microarchitecture. To reduce memory access latency, processors often follow a Virtually Indexed Physically Tagged (VIPT) approach to access the caches and the TLB in parallel. This induces a constraint on the size of the cache, s, and its associativity, k, as it must be accessed using only bits of the page offset (the lower part of the address that does not undergo virtual to physical translation) that we assume is p-bit wide. The maximum size of a VIPT cache is then $s = k \cdot 2^p$, so the only hardware solution to have larger caches is to increase the associativity k, which in turn increases the implementation complexity of the cache controller. Parasar et al. [29] show that, for many real-world applications, increasing the cache associativity significantly worsens the access latency and energy consumption. By increasing the page size, we can have large caches with fewer ways (associativity level), and therefore improve both performance and energy efficiency.

Modern caches also implement hardware memory prefetching

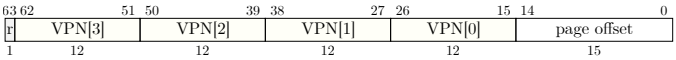
mechanisms that use speculation to increase performance. Roughly, these mechanisms take advantage of temporal and spatial locality to prefetch data that is contiguous in memory, but are commonly limited to data that resides on the same page [30]. By having larger pages, prefetchers can provide more data to the processor before hitting a page boundary, improving cache performance.

System level implications: Currently, the RISC-V standard defines 56-bit physical addresses for all memory systems, with 12 bits for the page offset, and 44 bits to represent the Physical Page Number (PPN). Using a different page size changes this organization: for 32 KiB pages, the page offset becomes 15-bit wide, and hence the PPN shrinks to 41 bits. Using all the available bits would allow to reach a physical address of 63 bits, with 48 bits to represent the PPN, quite enough for the next generations of supercomputers in the following 30 years. This leads to the alternative page-based 63-bit virtual-memory system for RISC-V, shown in Fig. 3b, that we call Sv63. A shorter term Sv51 scheme is also possible for those computer that do not require that large a memory space.

Since modern processors use multilevel page tables, the Virtual Page Number (VPN) part of the virtual address is split into multiple VPNs. A single TLB miss can result in a page walk, a sequence of dependent memory accesses to access the required virtual to physical translation. This has a huge impact on performance, as each memory access can take up to hundreds of processor cycles, degrading server applications by 5%-14% and HPC applications by up to 50% according to [31], [32]. Hence, reducing the number of levels of the page table is beneficial for performance. As shown in Fig. 3b, the Sv63 scheme eliminates one level of the page table with respect to the Sv57 scheme.



(a) Sv57: Standardized 5-level paging using 4 KiB pages



(b) Sv63: Proposed 4-level paging using 32 KiB pages

Fig. 3: When changing the page size, the VPN and page offset partition changes, allowing for fewer page walks

Another benefit of larger pages is that fewer page walks are needed to address the same, or even a larger, amount of memory. In RISC-V and x86-64, 5-level paging is required to address 2^{57} bytes of memory using 4 KiB pages, as shown Fig. 3a. In ARM64, 4-level paging is used to address up to 2^{52} bytes with the same base page size. With 32 KiB pages, 2^{51} bytes of memory could be addressed with 3-level paging, or up to 2^{63} bytes with 4-level paging, as showed in Fig. 3b.

Regarding persistent storage accesses, having larger contiguous pages is beneficial for storage transfers, as storage organization (be it magnetic or solid-state) favors long sequential reads and writes over sequences of shorter ones. Then, if the memory bloat stays under control, this may also lead to performance improvements. Of course this can have the negative effect of requiring more data to be written to the backing storage, even if only a small part of a page is actually modified. This effect could be exacerbated in the case of swapping, but this situation is not typical of HPC workloads.

General-purpose computing: Although we focused on HPC workloads, this methodology can also be applied to other environments. In general-purpose computing, for instance, the TLB miss rate and the memory bloat could be equally weighted, i.e. $w = 0.5$. We

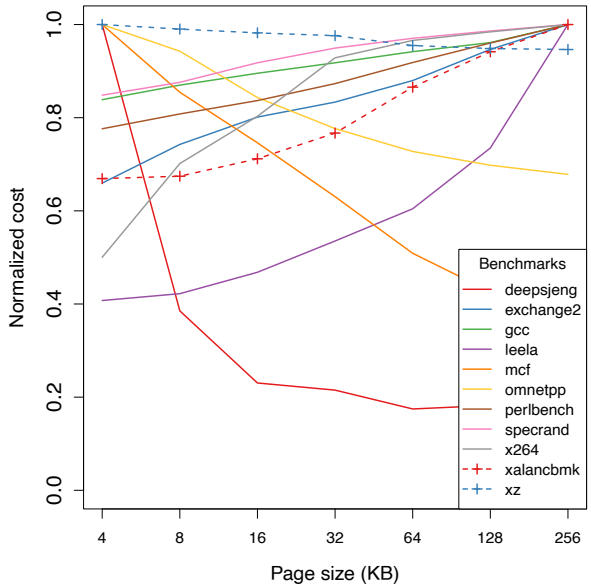


Fig. 4: Normalized cost of SPEC CPU2017 benchmarks for $w = 0.5$

applied our methodology on the SPEC CPU2017 Integer benchmarks, that measures compute intensive performance by stressing both the processor and the memory system. The results are shown in Fig. 4, and we conclude that the page size that reduces the cost for this set of applications is actually 16 KiB, with a miss rate reduction of $1.62\times$ and a memory bloat increase of $1.03\times$. These results appear to justify Apple's 16 KiB pages on their ARM line of processors, as they are designed for general-purpose use.

Yet, these results do not justify ARM's and RISC-V's choices of 64 KiB on their contiguous bit and Svnapot specifications. Fig. 2a shows that for the five cases studied, the cost is always increasing when using pages larger than 32 KiB. To reduce this cost, these solutions still support 4 KiB as the base page size, but at the expense of a bigger pressure in TLBs. This introduces more complexity in the OS as it is the responsible of finding how to best use this mechanism, in addition to superpages.

128-bit address space: Finally, future HPC systems might need to access very large memory pools, possibly requiring more than 64-bit addresses. The RISC-V includes a 128-bit specification, in which the general purpose registers of the processor and the base instructions work on 128 bits. However, the virtual memory organisation is yet to be defined [11]. We believe larger pages would be particularly beneficial to these systems, for which simply extrapolating from existing 64-bit architectures would be largely sub-optimal. With larger workloads and address spaces, it is expected that the pressure on the TLB will increase accordingly. Increasing the amount of addressable memory with larger pages, as well as allowing for a reduction in the number of page walks and other microarchitectural improvements, is a first step towards more scalable systems.

VI. Conclusion

Choosing the size of the pages can be difficult due to the compromise between translation performance and memory usage. Although 4 KiB pages have been the standard for more than half a century, with Apple recently moving up to 16 KiB pages and ways to merge TLB entries being added, this might change soon.

This study defines a method to determine the page size for a given system according to the trade-off between miss rate and memory

overhead. We analyzed HPC systems by exploring the impact of the page size on TLB miss rate and memory bloat for a set of well known parallel benchmarks. We show that, for systems in which memory usage is not as critical as performance, such as HPC, using 32 KiB pages can result in a mean reduction of $2.27\times$ in miss rate, with a $1.44\times$ increase in memory bloat compared to the 4 KiB baseline. Hence, we propose to use 32 KiB pages as an alternative page-based 63-bit virtual-memory system for RISC-V, specialized for HPC systems. Furthermore, we discussed some microarchitectural choices that become available with larger page sizes and that may improve overall performance, such as increasing the number of sets in the cache and reducing the number of page walk levels. This approach can also be applied to other cases, such as general-purpose processors, for which we show that 16 KiB pages is a good choice, which corroborates the choice in the newest Apple processors.

Although a 128-bit instruction set has already been specified in RISC-V, its memory organisation is yet to be defined. We believe these systems would specially benefit from bigger page sizes, so the results from this study can provide insight on this topic.

Acknowledgment

The authors would like to thank the French *Agence Nationale de la Recherche* (ANR) for funding the Maplurinum project (grant ANR-21-CE25-0016), and the project partners for the vivid discussions.

References

[1] John L. Hennessy and David A. Patterson. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[2] G. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr. Architecture of the IBM System/360. *IBM Journal*, pages 87–101, April 1964.

[3] Webb T. Comfort. A computing system design for user service. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, part I*, pages 619–626, 1965.

[4] Maynard Handley. M1 exploration - v 0.70. https://archive.org/details/m-1-explainer-070. Accessed: 2024-03-26.

[5] Apple A9. On web site "*7-Zip LZMA Benchmark*", at https://www.7-cpu.com/cpu/Apple_A9.html. Accessed: 2024-03-29.

[6] Jack J. Dongarra, Daisuke Takahashi, David Bailey, David Koester, Piotr Luszczek, Rolf Rabenseifner, Bob Lucas, and John McCalpin. Introduction to the HPC challenge benchmark suite, 2005.

[7] TOP500: The list. https://www.top500.org/. Accessed: 2024-02-20.

[8] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, et al. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 52–64. IEEE, 2020.

[9] Florian Zaruba, Fabian Schuiki, and Luca Benini. Manticore: A 4096-core RISC-V chiplet architecture for ultraefficient floating-point computing. *IEEE Micro*, 41(2):36–42, 2020.

[10] Federico Ficarelli, Andrea Bartolini, Emanuele Parisi, Francesco Beneventi, Francesco Barchi, Daniele Gregori, Fabrizio Magugliani, Marco Cicala, Cosimo Gianfreda, Daniele Cesarini, et al. Meet monte cimone: Exploring risc-v high performance compute clusters. In *19th ACM International Conference on Computing Frontiers*, pages 207–208, 2022.

[11] Andrew Waterman and Krste Asanović. Chapter 6, RV128I Base Integer Instruction Set, Version 1.7. In *The RISC-V Instruction Set Manual - Volume I: Unprivileged ISA*. The RISC-V Foundation, 20191213 edition, December 2019.

[12] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, USA, 4th edition, 2014.

[13] Weixi Zhu, Alan L. Cox, and Scott Rixner. A comprehensive analysis of superpage management mechanisms and policies. In *USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 2020.

[14] Andrew Waterman, Krste Asanović, and John Hauser. Chapter 5, "Svnapot" Standard Extension for NAPOT Translation Contiguity, Version 1.0. In *The RISC-V Instruction Set Manual - Volume II: Privileged Architecture*. The RISC-V Foundation, 20211203 edition, December 2021.

[15] Pinchas Weisberg and Yair Wiseman. Virtual Memory Systems Should Use Larger Pages rather than the Traditional 4KB Pages. *International Journal of Hybrid Information Technology*, 8(8), August 2015.

[16] Aninda Manocha, Zi Yan, Esin Tureci, Juan Luis Aragon, David Nellans, and Margaret Martonosi. The Implications of Page Size Management on Graph Analytics. In *IEEE International Symposium on Workload Characterization*, pages 199–214. IEEE, 2022.

[17] Yichen Yang, Haojie Ye, Yuhan Chen, Xueyang Liu, Nishil Talati, Xin He, Trevor Mudge, and Ronald Dreslinski. CoPTA: Contiguous Pattern Speculating TLB Architecture. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 67–83, 2020.

[18] Fujitsu Limited. A64FX Microarchitecture Manual, v1.3. https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.3.pdf, October 2020. Accessed: 2024-03-29.

[19] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, April 2005.

[20] Maplurinum anr - memory access plugin. https://gricad-gitlab.univ-grenoble-alpes.fr/maplurinum-anr/mem-access-plugin.

[21] Maplurinum anr - tlb model · GitLab. https://gricad-gitlab.univ-grenoble-alpes.fr/maplurinum-anr/tlb-model.

[22] Emilio G. Cota, Paolo Bonzini, Alex Bennee, and Luca P. Carloni. Cross-ISA machine emulation for multicores. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017.

[23] Marie Badaroux, Saverio Miroddi, and Frédéric Pétrot. To Pin or Not to Pin: Asserting the Scalability of QEMU Parallel Implementation. In *2021 24th Euromicro Conference on Digital System Design (DSD)*. IEEE, September 2021.

[24] Emilio G. Cota and Luca P. Carloni. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 2019.

[25] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications*, 1991.

[26] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[27] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 101–111. IEEE, 2016.

[28] Marie Badaroux, Julie Dumas, and Frédéric Pétrot. Fast instruction cache simulation is trickier than you think. In *Proceedings of the DroneSE and RAPIDO: System Engineering for Constrained Embedded Systems*, page 48–53, 2023.

[29] Mayank Parasar, Abhishek Bhattacharjee, and Tushar Krishna. SEESAW: Using Superpages to Improve VIPT Caches. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture*, pages 193–206, Los Angeles, CA, 2018.

[30] Georgios Vavouliotis, Gino Chacon, Lluc Alvarez, Paul V Gratz, Daniel A Jiménez, and Marc Casas. Page size aware cache prefetching. In *55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 956–974. IEEE, 2022.

[31] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, Seattle WA USA, 2008. ACM.

[32] Collin McCurdy, Alan L. Coxa, and Jeffrey Vetter. Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors. In *IEEE International Symposium on Performance Analysis of Systems and software*, Austin, TX, USA, 2008. IEEE.