# Key-Study to Execute Code Using Demand Paging and NAND Flash at Smart Card Scale

Geoffroy Cogniaux and Gilles Grimaud

LIFL, CNRS UMR 8022, University of Lille I. INRIA, Nord-Europe, France

**Abstract.** Nowadays, the desire to embed more applications in systems as small as Smart Cards or sensors is growing. However, physical limitations of these systems, like very small main memory, and their cost of production make it very difficult to achieve. One solution is to execute code from a secondary memory, cheaper, denser, but slower, as NAND Flash. Solutions based on Demand Paging and using a cache in main memory, began to be proposed and implemented in the domain of mobile phones, but consume too much RAM yet, compared to what a Smart Card can provide. In this paper, we show that we can dramatically increase performance by reducing the size of pages in the cache. This solution then allows a more intelligent access to the NAND. We also show that our solution allows to use Demand Paging within the limits of Smart Cards memories, where a conventional approach, offering too low bandwidth, makes code execution impossible from this kind of secondary memory. Finally, we present important future keys to optimize our proposal even more, and specially off-line code specialization aware of NAND characteristics and advanced cache properties.

## 1 Introduction

Nowadays, the desire to embed more applications in systems as small as Smart Cards or sensors is growing. However, challenges are not the same between a mobile phone and a Smart Card. Beyond the extreme physical constraints, as only a few kilobytes of RAM and not much more storage space for code, a Smart Card also has an extremely restrictive specification in regard to costs of production or energy consumption. It is not conceivable in these targets to run programs more expensive in space or energy only by inflating the hardware. The possible solution is to replace or at least extend the code storage capacity with another hardware with lower cost per bit of storage.

Generally, NOR flash is used as code storage space on systems such as Smart Cards. The NAND flash is another type of flash memory with different properties. This memory is also nonvolatile, has greater density, and therefore allows to embed more data than the NOR on a same silicon size, near 60%. The temptation is then huge to replace NOR by NAND. Unfortunately, the NAND, although coming with faster writings, is much slower to perform readings, and is not accessible byte per byte as NOR interface but per pages. This access mode causes incompressible latency and *a priori* excludes random accesses. NAND is then

D. Gollmann, J.-L. Lanet, J. Iguchi-Cartigny (Eds.): CARDIS 2010, LNCS 6035, pp. 102–117, 2010.
© IFIP International Federation for Information Processing 2010

most often used to support a file system and so far, its development has mainly been driven for this purpose.

Executing code from memories such as NAND is made possible by the introduction of a cache in main memory or the use of SRAM buffers added directly into the NAND. This new category is called hybrid Flash. However, the presence of a dedicated RAM space is not sufficient to close the gaps between NAND and NOR. A cache manager will then have to be smart enough to hide the latency of the NAND and ensure fluidity when providing instructions to a processor or a virtual machine. In this context of paged memory, the Demand Paging is the most popular caching strategy because it offers the best balance between overall performance and reduced cache space. The idea of this method is to access slow hardware only in cases of extreme necessity and try to keep in cache as much data as possible, generally sorted by relevance of future uses. Of course, knowing if a cached data will be finally useful or not, is a non-trivial problem.

If hardware researches are working to provide more efficient device to reduce the latency of the NAND, at software level, greater efforts are usually done on the optimization of the page replacement policy in the cache. But in systems with tiny main memory, it remains to be seen how many bytes to sacrifice for a cache, in addition to the vital needs of the system. But beyond that, the main problem with these systems will be to know if this sacrifice will be enough to perform code execution from this slow memory.

The remaining of this paper is organized as follow: Section 2 describes research motivations and issues. Section 3 lists related works. Section 4 discusses a new way to use Demand Paging and NAND flash memory to use it in Smart Card too. Section 5 presents experimental results on reading performance. Finally, section 6 will discuss about best cache page size and its direct interactions with page content for future optimizations.

## 2 Motivations Around Smart Card and Flash Memories Characteristics

In this paper, we focus on very small embedded systems such as Smart Cards or sensors. This kind of devices has very restrictive specifications and hardware constraints aiming to reduce production cost and energy consumption dramatically. Despite their size, industrial trends would like to embed more applications and bigger programs in those targets.

Smart Card and sensors are mainly built around an 8 or 16 bits processor and generaly between 1 and 8KB of RAM. Code is then executed from flash memory such as NOR, reserving RAM for volatile data such as execution stack and variables. Thus, executing bigger applications will require more NOR, which would also be too expensive in energy, size and production cost. As an alternative, read-only code could be executed from a cheaper and denser flash memory such as NAND.

NOR is byte-wise and has a good and constant performance in random readings. With these properties, NOR can support XIP (execute-In-Place), to run

programs directly, without any intermediate copy in main memory. The NAND flash has better writing performance and is cheaper and denser. It can store more informations on same die size and thus has a lower cost per bit. But NAND is much more slower in readings. NAND is block-wise [1]. The smallest read and write unit is the page that can store 512 or 2048 bytes. To read a byte randomly, the corresponding page must be first completly loaded in the hardware NAND data register. Only after this long time consuming operation, bytes can be fetched sequentially from the register until the wanted one is reached. In those conditions (Random worst case column of Table 1), NAND is not designed to support intensive random readings and is widely used to support file systems, designed with a per-block logic and almost always accessed sequentially.

**Table 1.** NAND characteristics aligned on a bus at 33MHz and 8bits wide

| Device | Page Size | Latency | Byte Fetch | Page Fetch | Min[1] | Max[2] |
|---|---|---|---|---|---|---|
| K9F1208Q0B [3] | 512 bytes | 15$\mu$s | 50ns | 25.6$\mu$s | 0.02 | 12.03 |
| MT29F2G08A-ACWP [4] | 2048 bytes | 25$\mu$s | 40ns | 81.94$\mu$s | 0.01 | 18.27 |
| NOR Flash | - | - | 40ns | - | - | 23.84 |

**Table 2.** NAND characteristics aligned on a bus at 54MHz and 16bits wide

| Device | Page Size | Latency | Byte Fetch | Page Fetch | Min[1] | Max[2] |
|---|---|---|---|---|---|---|
| K9F1208Q0B [3] | 512 bytes | 15$\mu$s | 50ns | 25.6$\mu$s | 0.02 | 12.03 |
| MT29F2G08A-ACWP [4] | 2048 bytes | 25$\mu$s | 20ns | 40.9$\mu$s | 0.01 | 29.61 |
| NOR Flash | - | - | 9ns | - | - | 108 |

[1] Worst Case: random reading bandwidth in MB/s
[2] Sequential reading Bandwidth in MB/s
[3] http://www.samsung.com
[4] http://www.micron.com

## 3 Related Works

The studies of code execution from NAND flash can be classified in four categories. In the first place, we can distinguish pure hardware solutions since NAND is not a well-adapted device to do random readings, and so far XIP [2, 3, 4, 5]. [6, 7, 8, 9, 10, 11, 12, 13] proposed software solutions using already available NAND, even if software is often limited to firmware here.

Besides, we can split both categories into two others: one aiming to propose an universal solution working with any program, and another one aiming to find best performance dedicating solutions to one program or one type of programs such as multimedia streaming.

In systems with limited SRAM size, copying the entire program in main memory is not possible. Instead, hardware and software researchers have worked on

solutions using a cache managed by the Demand Paging strategy. It can be either in existing main memory or in SRAM integrated directly in the NAND device [2], resulting in a new one, now called hybrid NAND.

Generally, hardware based solutions are coming with a local SRAM cache managed by new circuits but many parts of the implementation are performed by the firmware. The main purpose to introduce a SRAM and a cache in the NAND device is to provide an interface similar to the NOR one, which is byte-wise and randomly accessible. But as explained in [10], in already available hybrid NAND such as OneNAND™ by Samsung[14], the integrated SRAM is not a cache yet, and doesn't offer enough performance to actually execute code[1]. If OneNAND offers a double buffers mechanism to access stored data more randomly by overlapping some internal NAND accesses and their latency, it doesn't hide all of them. [11, 15] propose a software-based solution to use this buffer as a true but secondary cache in relation with a higher level cache in main memory. When a NAND page is accessed more often than a static threshold, this page is promoted from in-NAND cache to main memory cache which allows faster accesses.

Most of the time, solutions aim at improving or adapting the replacement policy needed by the Demand Paging strategy because researches [16] showed that a gap always exists up to the theoretical best one, known as MIN [17]. When a cache is full and when the CPU tries to read instructions not present in cache yet, the new page loaded from the NAND must overwrite an existing one in the cache. Then, the replacement policy must decide which one to erase, hoping that this choice will be relevant for the future execution of the program. In the studied works, this choice may be based on priorities [2], pinning [9, 12], derived from well-known policies like LRU (Least Recently Used) [8, 9, 11],... , or using a completely new approach [4] by embedding a trace-based prediction tree within the NAND device, with good performance (being close to MIN behaviour) but with a RAM overhead, and universality lost.

All of these proposals have mobile phones as smallest devices. In this paper, we take an interest in extreme constraints of Smart Cards-sized devices with very small SRAM space to manage a cache (less than 4KB). We also want to evaluate performance behind very slow bus (33MHz, 8bits wide). Because in such devices, reserving SRAM bytes for a dedicated process represents a strong sacrifice, and buses are well-known bottlenecks. To successfully execute code from NAND-like memory, we will have to tune implementation to reduce this sacrifice, but as a first issue, it remains to be seen whether this sacrifice will be enough to reach abilities, before performance.

## 4 Demand Paging at Smart Card Scale

### 4.1 Conventional Demand Paging Issue in Smart Card

In conventional Demand Paging used in literature, a page in the cache has the same size that a physical NAND page, since reading a NAND page has a very

[1] Though it can execute a boot loader.

high cost, due to long access latency. An efficient Demand Paging and its replacement policy are aware of this problem and access the managed slow memory as less as possible.

Smart Cards, mobile sensors, and other small embedded systems have constraints that make conventional Demand Paging unusable in their context because of low performance. For instance, if we consider a system with only 2KB of RAM available for a cache, and a NAND with a page size of 2KB, there will be only one page in the cache. With this assumption, a replacement policy is useless, and the read bandwidth will be close to worst case of Table 1, unless executed program is only 2KB.

It can be said that the main problem with NAND is its long access latency. But, in our context, hiding latencies is not enough. We have explained there are two steps for bytes reading from NAND : a load phase and a sequential fetch phase. If the load phase has a good attention in research and industry, the second topic has not been well investigated yet, since, in mobile phone, authors considered good clock and bandwidth for buses. But, even if we have a NAND claiming to have a 108MB/s bandwidth and hiding all latencies, we will not have a real bandwidth faster than the bus one and copying a complete NAND page into SRAM cache may take a long time too (5th column of table 1).

### 4.2 Performance Is under Influence

A parameter may influence another and an accurate assessment must take these interactions into account. Into a graph of influence (Fig. 1), we listed and put physical and logical parameters face to intermediate results that are necessary for a proper assessment of our study. They can be classified into 4 categories:

- physical parameters related to hardware (rectangle boxes)
- logical parameters configuring the cache (inversed house boxes)
- development toolkit, languages, compilers, ... (ellipse boxes)
- intermediate statistics influencing the overall bandwidth (rounded boxes)

As shown in Fig. 1, page fault or MISS causes a series of transactions that affect the overall throughput. To make an instruction not present in cache available, the cache manager must load the new page into the hardware NAND data register, read the content of this register through the bus, then copy this page into the cache and finally actually read the requested instruction. (For short, in the remaining of this paper, we will refer to these four steps as *load*, *fetch*, *write*, and *read* steps). A MISS is itself influenced by the cache size (in pages count), the size of a cache page, the content of this page and of course the replacement policy. The Demand Paging is not only dependent on the way it is managed. If a replacement policy follows locality principle [18], only page size and page content create conditions for a better locality.

A compiled program is not a random series of instructions but follows some patterns of instructions and controls flows. The analysis of these patterns shows that a program has two interesting features grouped under the concept of the

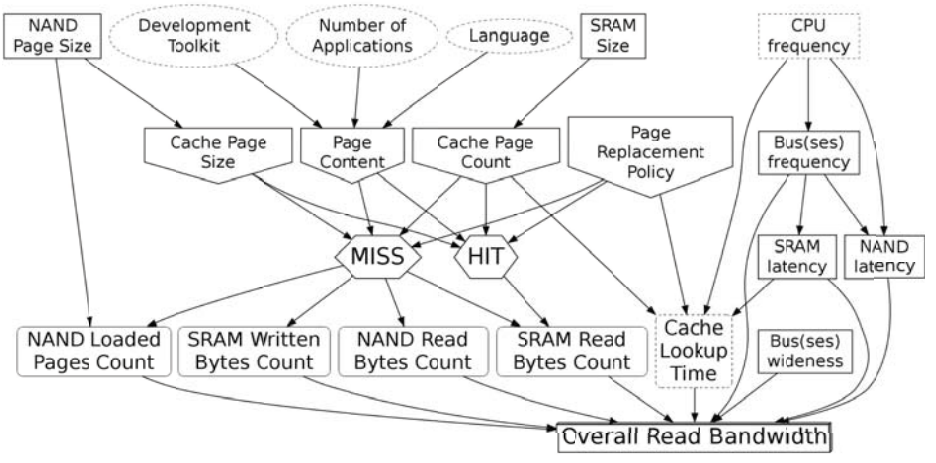



**Fig. 1.** Parameters influence graph

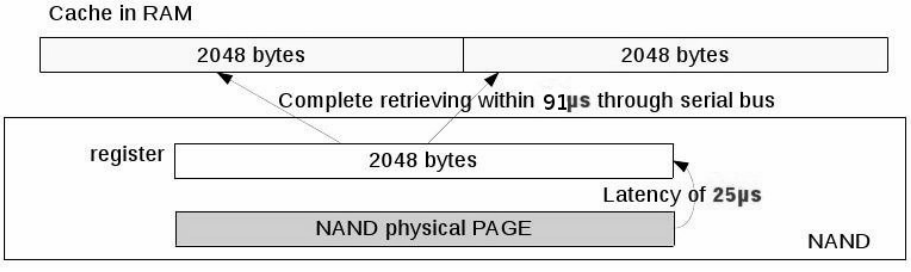



**Fig. 2.** Conventional approach of Demand Paging

locality principle [18]. A program tends to use instructions located in the same memory area. This is the principle of spatial locality and this area is therefore strongly related to the size of a page. A program also tends to reuse instructions that have already served in the recent past, typically in a loop. This is the temporal locality. To help a replacement policy to use more efficiently the temporal principle, blocks of code should be grouped by affinity, and therefore in the same page of NAND to avoid or reduce page faults.

### 4.3 Performance Improvement without Replacement Policy Change or Redesign

Regardless of the manner in which it is implemented and which policy is used, if we analyse the behaviour of the Demand Paging using NAND with small pages compared to NAND with big pages, we can see that the smallest achieves best performance. First because it has shorter access latency, and second because at the same cache size we can store more pages in it, for instance within 4KB of

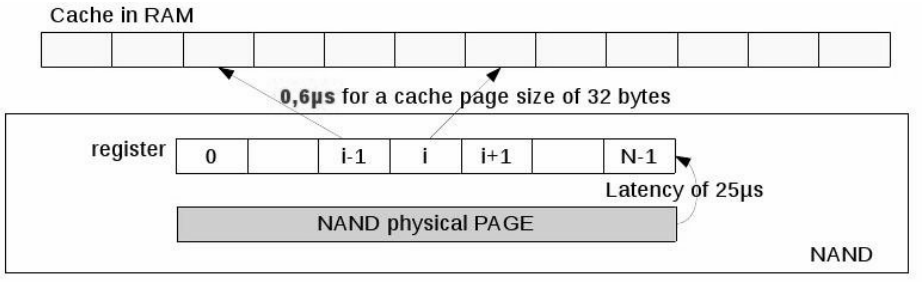



**Fig. 3.** A new approach using reduced cache page size

SRAM, we will store 8 pages of 512 bytes versus 2 pages of 2048. Using small pages, the changes applied to the content of the cache by the replacement policy are more relevant and efficient. It then speaks in favour of smaller cache pages.

If we reduce the size of a cache page, the number of bytes fetched from NAND, put on the bus, and then written into the SRAM cache decreases because there is no need to retrieve complete pages of NAND, but only smaller inner-block. The number of pages loaded into the data register of the NAND also decreases for two reasons:

1. The replacement policy tends to keep in cache only needed code blocks instead of full pages of NAND with potentially unnecessary methods. (The smaller the methods are, the more this probability increases).
2. Reducing the size of a cache page can increase its number into same consumed SRAM space, and thus improve the relevance of the cache content.

Reducing the cache page size introduces another advantage, completely useless until now if a page of cache and a page of NAND have the same size. Without hardware changes, we can now consider the data register as a real buffer. We have seen that for a conventional NAND (non-hybrid), it was sequentially accessible. Let us consider now a page of NAND with 512 bytes, a data register with the same capacity and a page cache size of 64 bytes, which divides the data register into 8 consecutive logical blocks. Three cases may arise after the fetch of the block number $i$ of page number P and following a new page fault:

1. the page fault requests another page than P: we have to load the new page in the NAND register
2. the page fault requests block $i+j$ of P, then in that case there is no need to reload data register, we can fetch this block
3. the page fault requests the block $i$-$j$ of P, which causes a reload of the register, because we cannot turn back

$j$ is here the number of block to skip to reach the wanted one. (We notice that all blocks between $i$ and $j$ will have to be fetched too)

Storing in memory the page number that is currently in the NAND register and the current block number is enough to distinguish the case (1) of the cases

(2) and (3), and up to avoid unnecessary register reloads which slow down overall performance.

## 5 Experiments

We investigated cache configurations independently of replacement policies and cache manager implementations, and different contexts such as native programs or Java programs. Our goal is to find solutions in case that replacement policies are useless, as described in Section 4.1, instead of redesigning a new one. Thus, we used LRU and MIN as references (which are the most common boundaries in cache studies), to instead investigate more specially NAND characteristics, cache configurations and crtitical Smart Card's hardware constraints such as the external bus frequency.

### 5.1 Experimental Setup

The results are from a new dedicated simulator based on the parameters described in Fig. 1. The simulation process happens in two stages. After providing the physical parameters to build a simulated architecture, a trace is replayed against it by applying the desired replacement policy, and then counting the MISS and HIT as well as collecting the necessary informations to compute the overall bandwidth in MB/s. We wanted to analyse and characterize the impact of pages sizes. Results are thus presented with the assumption that the simulator knew with a null cost if an instruction was already in the cache or not. It assumes a negligible overhead executing cache lookups since we will use finally a limited number of cache pages here. But it will be investigated in future works.

Traces of C programs (like Typeset from the MiBench benchmark suite) were obtained using Callgrind on Linux. The traces of Java programs have been generated by JITS (Java In The Small [19]), an instrumental JVM developed and maintained by the INRIA-POPS team[2]. Regarding the storage in NAND, we consider only what would be in the NOR: the *.text* section of ELF binaries, and raw bytecode arrays of Java class files. They are in consecutive pages of NAND (if necessary, the same function can be distributed among several pages) as if they were copied by a programmer, in the order of the binary produced by the compiler.

### 5.2 Event-Driven Java Applications

We also needed a true Smart Card scale use-case, because most of benchmark programs come with big loops and/or deep dependencies on external libraries. At the same time, existing embedded systems benchmarks do not provide similar environment to those used in today's Smart Card. Systems such as JavaCard are based on event-driven multi-applicative environment (like Cardlets processing APDU).

[2] http://www.inria.fr/recherche/equipes/pops.en.html

The JITS platform is also one of them, when configured for sensors or Smart Cards. This suite of tools allows off-board code specialization to reduce the total memory footprint to strictly necessary Java classes and methods. Different from other embedded JVM, JITS allows the use of standard Java API instead of being limited to a specific subset, like in JavaCard. We then use the on-board JITS bytecodes interpreter, working in events fashion, which enables us to simulate parallel execution of several Java applications without the need to have a per-thread execution stack.

In our simulation, the JITS platform is configured for applications running on a Crossbow MicaZ target: 4KB of RAM, 128KB of NOR Flash. The applications deployed on the target then contain 56 classes and 119 methods for 6832 bytes of Java bytecodes, and requires 2.1 KB of RAM for volatile data such as stack, some basic Java objects and critical sections of code like the event manager. It leaves only 1.9 KB of RAM to physically store a cache on-board. In the simulator, we will consider that the JVM is in NOR, and Java bytecodes stored in a simulated NAND. The results reproduced here are from a trace of 1,534,880 Java bytecodes through 172,411 calls/returns of methods during the execution of 3 parallel event-driven applications, sampling data at different frequencies and then forwarding samples over the air network.

### 5.3 Experimental Results

**Impact of NAND Pagesize and Buses Constraints.** As a starting point, we explored conventional Demand Paging with cache page size aligned on NAND page size. Fig. 4(up) reports reading performance of the MiBench-Typeset applications after simulations of caches between 2 and 16KB, and using NAND flash with 512 or 2048 bytes per page. It shows that small pages outperforms large pages, even if it is no more an industrial trend to provide NAND with small pages. If we look at Fig. 4(bottom), with hardware reduced to Smart Card abilities (see Table 1), we are further from NOR performance. Code caching seems to require much RAM to hide weaknesses of NAND and buses, which is impossible in our targets.

**Impact of Cache Page Size.** Aiming at giving power back to replacement policy with another way, we investigated reduction of cache page size, discovering a new way to manage the NAND register. Fig. 5a shows, using a cache of 2KB, that performance is becoming closer to NOR. Fig. 5a reproduces simulations of our JITS sample with cache page size reduced by 2 each time. The first consequence is that cache page count increases. As described in Section 4.3, Fig. 5b shows with the same configurations as Fig. 5a, the impact on each page fault step : a decrease of *write* (due to reduction), and then a decrease of *fetch* and *load* (due to reduction and new register management).

With this new configuration, the gain is double:

- Either, at same cache size, we are improving reading performance.
- Or, at same reading performance, we are reducing dramatically the size of the cache.

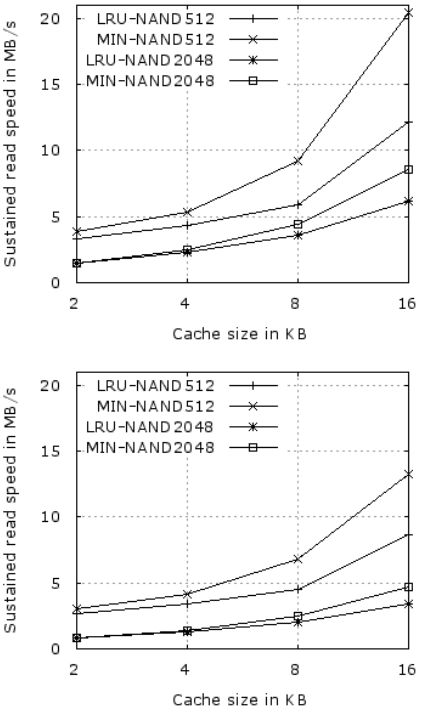
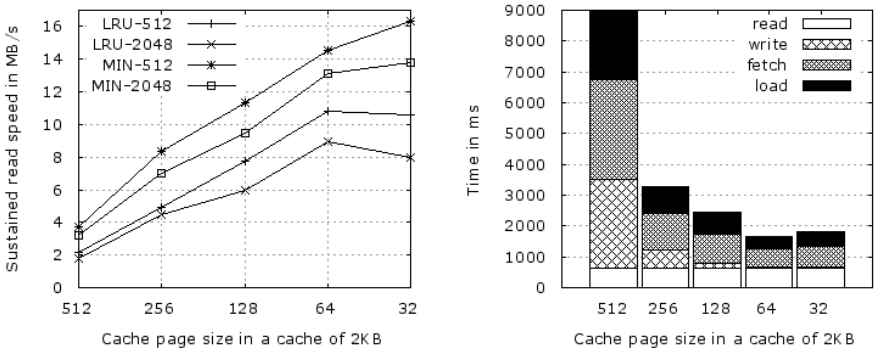


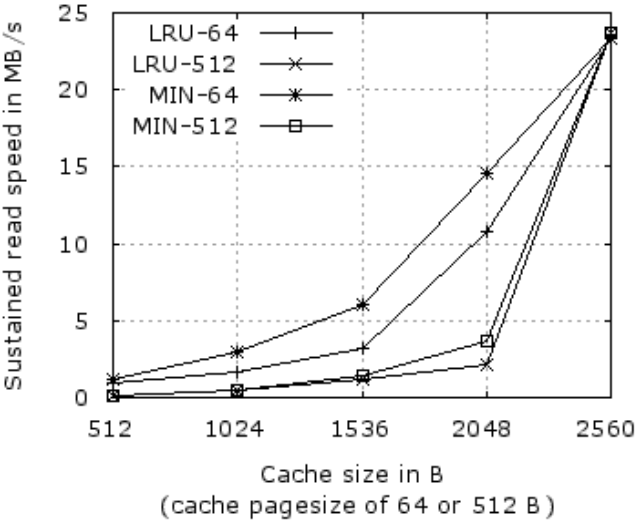

**Fig. 4.** Impact of NAND page size and bus speed on read Bandwidth, on a bus @54MHz-16bits (up) and on a bus @33MHz-8bits (bottom)



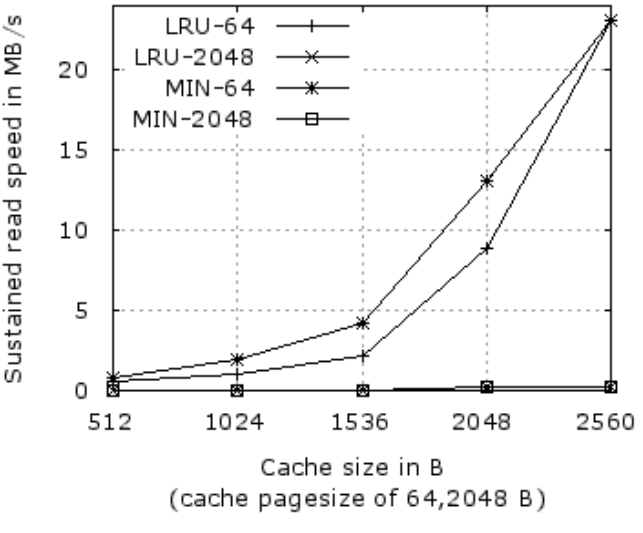

**(a)** JITS - Cache page reduction **(b)** JITS - Reduction's effect

**Fig. 5.** Reduction of cache page size

If we take a look at our JITS application side (Fig. 6a for a 512 bytes NAND page size, Fig. 6b for a 2048 bytes NAND), we can show another very interesting advantage. These figures report reading performance using a cache page size of




**(a)** JITS - NAND 512




**(b)** JITS - NAND 2048

**Fig. 6.** Conventional Demand Paging compared to a cache with reduced page size

64 bytes. With a cache of 2KB (32 pages of 64 bytes), reading performance outperforms conventional Demand Paging, and even more, it makes code execution from a NAND with Demand Paging strategy possible (see Section 4.3). We are closer to NOR performance.

## 6 Discussion about the Best Cache Page Size

As shown in Fig. 5a, tuning the cache page size can improve overall performance. Unfortunately, the best cache page size is not unique, and may differ from programs to programs, and what is more, it may differ in a same program. For instance, the best cache page size for our JITS Java sample is 64 bytes when using LRU, though it is 32 when using MIN. To understand this difference, we have to distinguish two concurrent phenomenons, both getting better each other, but with some limits we will discuss here.

### 6.1 Buffer Effect

The first phenomenon, the buffer effect, has already been described above and refers to the new management of the NAND register. Based on it, a good page replacement policy is able to specialize the code on the fly, while searching and keeping only what it needs, without loading unnecessary instructions. Nevertheless, the limit to this specialization cannot be the instruction itself (ie, a cache page size of one byte), mainly because performance bottleneck would be transferred to lookups and indexing phases of the replacement policy algorithm.

### 6.2 Matching Code Granularity

The second phenomenon is based on code granularity concept and can set the limit of the first one, while explaining for example why a small cache page size can be better than another smaller one. In fact, a simple code specialization is already done by the page replacement policy. It means that finally, the policy splits code into two distinct categories: used and unused code. Following this, to keep efficiently data in cache, it will access NAND only to find useful code. If we help it with doing this distinction, overall performance will be improved.

At sources level, a program consists of classes and methods, or functions, depending on the programming languages. Once compiled, it then consists in series of basic blocks. A basic block is a group of several instructions, which will always be executed sequentialy because being between two jumping instructions (call/return, conditional jumps). With these bounds and to be efficient, a replacement policy should load an entire basic block when accessing its first instruction for the first time. Reducing cache page size to match the average size of a basic block as close as possible then gives better chance to the replacement policy to find more quickly useful sub-parts in a program, that may be spread over many pages into the NAND. For instance, the average basic block size (among only used basic blocks during simulation) of our JITS Java sample is about 60 bytes. That is the reason why the best cache page size for this sample is between 32 and 64 bytes. At this point, we can understand the impact of the noticed concurrent phenomenons. The best cache page size for LRU is 64 bytes, with the average code size aligned up to the cache page size. But MIN, having a better prediction mecanism than LRU, can take more advantage of the previously described buffer effect, and continue to have better performance with

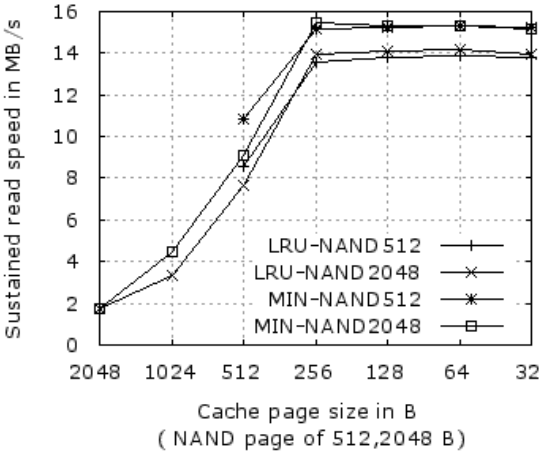



**Fig. 7.** MiBench-MAD - LRU and cache page reduction in a cache of 2KB

smaller and smaller cache page size. Its limit will be now the number of pages to maintain in cache and the time consumed by the replacement algorithm for looking up or indexing.

The MiBench-MAD benchmark (Fig. 7) shows that in fact the average basic block size is dominating the search for the best cache page size. MAD is a MP3 decoder calling, 90% of the time, two big functions. It puts the average size to 220 bytes. As shown in Fig. 7, best cache page size is still around this value. Nevertheless, with a higher value we see that we don't have the positive buffer as expected any more. Code granularity is thus as important as other cache parameters to tune an efficient cache system.

### 6.3 Toward a Better In-Binary Organization

We noticed in section 4.3 that a page fault can request a new block in the page already present in the NAND register. It results in a block jump (from block $i$ to $j$). Getting $j$ smaller than $i$ is not an optimal situation because it causes the page to be reloaded. On the other hand, getting $j$ greater than $i+1$ is neither a good situation because unnecessary bytes between block $i$ and block $i+j$ will have to be fetched. Fig. 8 shows a spectrum of distance between $i$ and $j$ during our JITS sample (cases 2 and 3 described in section 4.3). A negative distance means that the same page had to be reloaded int the NAND register to read a block towards. A distance of 0 means that whereas the replacement has choosen to replace a block, it has finally needed the same block again resolving the next page fault. A distance of 31 means that the full NAND page had to be loaded and fetched, only to access its last logical block. Only 11.5% are in the optimal case where $j=i+1$, the only one without useless *fetches*.

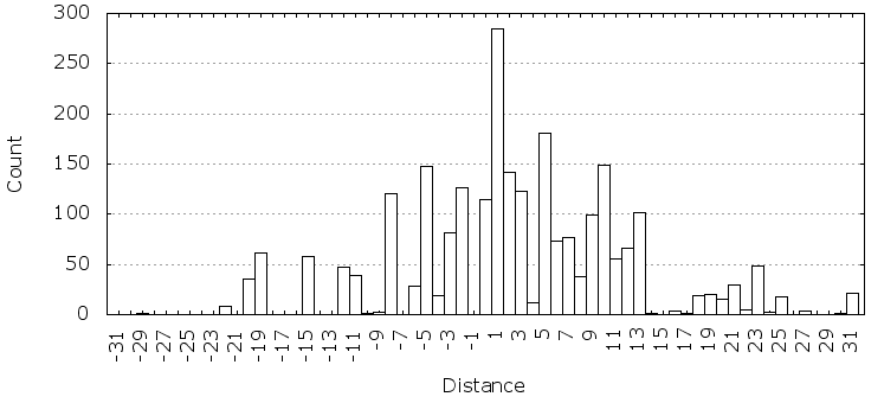



**Fig. 8.** JITS sample - Distance between two consecutive block requests using a cache of 2KB (32 pages of 64B)

In-binary organization is conditioned either by language, by compiler, or both. Fig. 8 speaks in favour of a control-flow analysis to group blocks by affinity of calls or sequences, directly in the binary.

## 7 Conclusion

We presented a new approach to parameterize the Demand Paging strategy to cache a slow secondary memory such as NAND flash in the context of Smart Card or embedded systems with the same size and the same constraints, such as slow buses and tiny main memory. Combined with an intelligent access to the NAND data register, reducing cache page size dramatically improves performance where conventional approach was close to worst case in term of bandwidth. We discussed about the optimality of our proposal, showing that the best cache page size were close to average size of used basic blocks. We noticed that this value could differ from programs to programs, and then may not always be shared between them to optimize all in once. We also pointed that the page replacement policy acted almost naturally as a on-the-fly code specializer. It means that running already Demand Paging-specialized code on this kind of targets may represent another interesting opportunity of gains.

## References

[1] Micron Technical Note 29-19: NAND Flash 101 (2006)
[2] Park, C., Seo, J., Bae, S., Kim, H., Kim, S., Kim, B.: A low-cost memory architecture with nand xip for mobile embedded systems. In: CODES+ISSS 2003: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pp. 138–143. ACM, New York (2003)

[3] Lee, K., Orailoglu, A.: Application specific non-volatile primary memory for embedded systems. In: CODES/ISSS 2008: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis, pp. 31–36. ACM, New York (2008)
[4] Lin, J.H., Chang, Y.H., Hsieh, J.W., Kuo, T.W., Yang, C.C.: A nor emulation strategy over nand flash memory. In: 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2007, pp. 95–102 (2007)
[5] Lee, J.H., Park, G.H., Kim, S.D.: A new nand-type flash memory package with smart buffer system for spatial and temporal localities. Journal of Systems Architecture: the EUROMICRO Journal (2005)
[6] Lin, C.C., Chen, C.L., Tseng, C.H.: Source code arrangement of embedded java virtual machine for nand flash memory, pp. 152–157 (2007)
[7] In, J., Shin, I., Kim, H.: Swl: a search-while-load demand paging scheme with nand flash memory. In: LCTES 2007: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, pp. 217–226. ACM, New York (2007)
[8] Lachenmann, A., Marrón, P.J., Gauger, M., Minder, D., Saukh, O., Rothermel, K.: Removing the memory limitations of sensor networks with flash-based virtual memory. SIGOPS Oper. Syst. Rev. 41, 131–144 (2007)
[9] Kim, J.C., Lee, D., Lee, C.G., Kim, K., Ha, E.Y.: Real-time program execution on nand flash memory for portable media players. In: RTSS 2008: Proceedings of the 2008 Real-Time Systems Symposium, Washington, DC, USA, pp. 244–255. IEEE Computer Society, Los Alamitos (2008)
[10] Hyun, S., Lee, S., Ahn, S., Koh, K.: Improving the demand paging performance with nand-type flash memory. In: ICCSA 2008: Proceedings of the 2008 International Conference on Computational Sciences and Its Applications, Washington, DC, USA, pp. 157–163. IEEE Computer Society Press, Los Alamitos (2008)
[11] Joo, Y., Choi, Y., Park, C., Chung, S.W., Chung, E., Chang, N.: Demand paging for onenand$^{TM}$ flash execute-in-place. In: CODES+ISSS 2006: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis, pp. 229–234. ACM, New York (2006)
[12] Park, C., Lim, J., Kwon, K., Lee, J., Min, S.L.: Compiler-assisted demand paging for embedded systems with flash memory. In: EMSOFT 2004: Proceedings of the 4th ACM international conference on Embedded software, pp. 114–124. ACM, New York (2004)
[13] Kim, S., Park, C., Ha, S.: Architecture exploration of nand flash-based multimedia card. In: DATE 2008: Proceedings of the conference on Design, automation and test in Europe, pp. 218–223. ACM, New York (2008)
[14] `http://www.samsung.com` (Onenand$^{TM}$ clock application note)
[15] Joo, Y., Choi, Y., Park, J., Park, C., Chung, S.W.: Energy and performance optimization ofdemand paging with onenand flash. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(11), 1969–1982 (2008)
[16] Al-Zoubi, H., Milenkovic, A., Milenkovic, M.: Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In: ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference, pp. 267–272. ACM, New York (2004)

[17] Belady, L.A.: A study of replacement algorithms for virtual storage computers. IBM Systems Journal 5(2), 78–101 (1966)
[18] Denning, P.J.: The locality principle (2005)
[19] Courbot, A.: Efficient off-board deployment and customization of virtual machine based embedded systems. ACM Transactions on Embedded Computing Systems (2010)