

Review of paper

SWL: A Search-While-Load Demand Paging Scheme with NAND Flash Memory

by J. In, I. Shin, H. Kim

Student Number:

1. Introduction

This paper seeks to address the problem of supporting the increasing number and size of applications stored and executed on mobile phone platforms. It asserts that the current methods of application execution - NOR-XIP (which utilises true random-access NOR flash memory to execute applications in place) and Shadowing (which stores applications in cheaper block-access NAND flash and loads the entire application into main memory SRAM on execution) - suffer from increasing costs and prohibitive boot times as the number and size of applications increases. Demand paging - a well established method in non-embedded computing of supporting the execution of applications that exceed main memory size - is noted as a candidate solution suffering from unpredictably long page fault latency which renders it unsuitable for embedded platforms.

The authors aim to increase the attractiveness of demand paging as a solution by reducing the latency overhead. It is proposed to exploit a particular aspect of NAND flash memory when combined with SRAM buffers - the fact that a read operation is a two stage process, with the first stage being independent of destination address - to parallelise the operations of reading the page in from secondary storage and finding a free frame in which to place the page in primary memory. The method, Search-While-Load Demand Paging, is implemented, two common page replacement algorithms are modified to take advantage of the implementation, and testing is performed to quantify the effectiveness of the approach.

2. Discussion of Approach

The paper begins with a reasonable introduction to the functional details of NAND memory and the architecture of OneNAND, a NAND flash device developed by Samsung. The fact that NAND flash is a block-accessed device whereas NOR flash is true random access could have been made more explicit, though this may have been knowledge assumed of the reader. It is described that NAND flash memory generally contains a "stopover buffer" into which a read operation first transfers the data block before transferring to the host; this first step does not require the address on the host in order to complete. The intuition is that this two-step process presents the opportunity for parallelising at least part of the page fault handling; it is noted that existing demand paging algorithms are geared toward single-step operation of HDDs, motivating the need to develop a specific method for NAND flash that can exploit this opportunity for parallelisation. The intuition and motivation are both sound.

The authors do a good job of describing their approach to implementing parallelisation. The steps of page fault handling are enumerated, named, and described, and diagrams are used to make it clear those steps that can occur in parallel. It is made explicit that the *LoadStopover* step, responsible for reading the required page from the NAND cell array into the NAND buffer, is completely independent of the *GetFreeFrame* step, which utilizes a page replacement algorithm to find a physical frame, potentially evicting an existing frame if the primary memory is full; hence these two steps be parallelized. Equations are developed which quantify the latency and effect of the parallelisation in terms of the time taken for each step; though these equations are obvious they are nonetheless appreciated. It is assumed that *LoadStopover* is more costly than *GetFreeFrame* - indeed, approximate time for *LoadStopover* in OneNAND is stated as 30us - and it is noted that *GetFreeFrame* can thus be "hidden" by *LoadStopover*. The authors refer to the difference between these two steps as "spare time".

A further intuition is that the page replacement algorithms themselves were written for sequential page fault handling and hence can be further optimised by exploiting the "spare time" created by parallelisation. Two page replacement algorithms are surveyed: Clock and FIFO with Second Chance. The operation of Clock is briefly described and the major drawback of varying and potentially long latency in finding a free frame is noted. A Parallelisation-Aware

Clock (PA-Clock) scheme intended to overcome this latency is detailed, in which a free frame is searched for by using the standard algorithm up until the "spare time" is used, at which point the current page being considered is evicted, effectively degrading the algorithm to random. A natural concern is how this degradation to a random algorithm might affect the page cache miss ratio, however this concern is dealt with later in the paper. Additionally, it is not made clear how the parallelisation is synchronised, i.e. how does the algorithm become aware that the *LoadStopover* step has completed and hence the spare time is used; further details on this point would have been appreciated.

The FIFO with Second Chance (FIFO-SC) algorithm is also detailed, and the definitions of "true" faults, in which the page does not exist in main memory, and "false" faults, in which the fault is intended to determine that a candidate page for eviction has been referenced and hence should be given a second chance, are both provided. It is described how a true page fault is always constant and short, but a false fault which results in several list modification operations is relatively longer and cannot be parallelised because, given that the required page already exists in main memory, it does not result in a read from the NAND flash. A scheme, Parallelisation-Aware FIFO with Second Chance (PA-FIFO-SC), is proposed in which some of the list modification operations of a false fault are delayed and performed in the "spare time" of a true fault, hence reducing the latency of a false fault whilst not increasing the latency of a true fault. The intuition here is sound, though the reader did find the details a little confusing and pseudocode would have been welcomed to describe the scheme in further detail.

3. Discussion of Evaluation

The authors conducted evaluation of their method in two stages: first, the performance of the parallelised fault handling was measured and discussed using the standard page replacement algorithms; second, their parallelisation-aware algorithms were utilised and evaluated. This two-phase approach is sound in that it allows isolation of the efficiencies gained in both aspects of the proposed method.

The performance of the parallelised page fault handler with the standard page replacement

algorithms was measured by implementation and experimentation on top of the Nucleus operating system running on an ARM9 processor with a Samsung OneNAND flash device as secondary storage. It is unclear, however, how the experiments were performed: neglecting to describe the number and type of experiments run limits the extent to which we can interpret the results; this is a defect in the evaluation which must be addressed.

A graph of the page fault latency for (non-PA) Clock and (non-PA) FIFO-SC algorithms using both sequential and parallelised page fault handling is presented; this shows that *GetFreeFrame* is completely hidden by *LoadStopover*, and hence is effectively eliminated as a source of latency. The results are quantified as an 11.5% gain for Clock and an 8.23% gain for FIFO-SC; one can assume that the lower gain for FIFO-SC was a result of the non-parallelised false-fault handling, which further highlights the necessity of separating the non- and PA-aware evaluations. Additionally, as it is noted that the Clock latency depends on a variable input to the algorithm (page list walk count), a graph comparing the latencies for different values of the variable for both sequential and parallelised page faulting is provided. This graph describes the latency increasing linearly with page list walk count for the sequential method, but remaining constant for the parallel method until the latency exceeds the "spare time", at which point it also increases linearly. The observations from this experiment are correctly interpreted by the authors, and it provides further justification for the provision of a parallelisation-aware algorithm. In summary, these results were as expected and, along with showing the potential gains of the technique, illustrate the author's success in predicting the outcomes.

The performance of the parallelisation-aware page replacement algorithms was evaluated by simulation; whilst this is a reasonable method for evaluation, and justification is provided, the reader is left wondering why both evaluations weren't performed in the same manner. At least in these experiments the full details of the traces used and the conditions under which they were performed were provided.

First, the PA-Clock algorithm was tested against the standard Clock algorithm with parallel page fault handling for variable page list walk count; a graph is provided illustrating that the PA-Clock does not suffer from a linear increase in latency after the "spare time" is consumed; this is expected given that the algorithm explicitly bounds the latency to this spare time. The concern mentioned above regarding the possible increase in page misses due to the degradation

of the algorithm to random is then effectively considered: the page miss ratio of both schemes is measured and compared. Very minor increases are listed for the PA-Clock but none are significant; the method is thus successful in decreasing the worst case latency without affecting the average case. This is an important contribution in and of itself; it should be noted that the technique utilised is not specific to a parallelisation-aware Clock algorithm and indeed could be implemented in the standard Clock algorithm as a method of reducing the worst case latency. It would have been interesting to see the results of implementing the method for the standard algorithm without parallelisation.

Second, the PA-FIFO-SC algorithm was tested against the standard FIFO-SC algorithm with parallel page fault handling. The method of delaying some of the list modification operations until a true page fault in order to reduce the latency of a false page fault was successful: the figures and graph provided show an approximately 21.7% reduction in false page fault latency. The natural concern with this method is to what extent the reduction in false page fault latency contributes to a reduction in real average latency; this concern is effectively managed by comparing the average instruction fetch time for varying page cache sizes in each simulation, though is unclear to the reader as to why only the instruction fetch time was compared and justification for this would have been appreciated. The results show effective gains of up to 34.1%, though very low gains are realised when the simulation involves a high percentage of instructions that can be serviced by the instruction cache (and hence a high instruction cache hit ratio); again, it would have been prudent to also measure the effect on average latency for data references.

4. Major Shortcomings

Aside from the drawbacks in methodology described in the discussions above, there were a few major shortcomings in this paper which may severely impact the utility of the method proposed and which should have been dealt with by the authors.

Firstly, a write operation to NAND is significantly slower than a read; the proposed method deals with this by paging out only code and clean data. This represents a major variation to the standard algorithms implemented, yet it is glossed over (indeed, it is only mentioned in a

foot-note), and its effect is not explicitly tested. No mention is made of the write-load of the tests used, and several tests measure only the instruction fetch latency, completely disregarding potential writes altogether. A greater mention of this issue should have been made, and tests with a heavy write load should have been used (such as the ACROBAT test used in their first referenced paper, in which 60% of the memory references are writes).

Secondly, it is established the the cost is dominated by reading from the NAND flash (i.e. the *LoadStopover* step), and this cost is stated as 30us for a 2Kb page on the OneNAND device. However, it is not mentioned that the OneNAND has an upper limit of 2Kb page size. The ARM9 is listed as supporting 1Kb, 4Kb, and 64Kb page sizes; 1Kb is selected, and no justification or discussion of this selection is provided. I would posit that the 1Kb page size on the ARM was selected because it was less than the maximum 2Kb page size supported by the OneNAND: were a 4Kb or 64Kb page size to be selected, multiple *LoadStopover* steps would have to be performed, each at a cost of 30us, which would severely limit the effectiveness of the technique. At the very least a discussion of this issue should have been provided; testing should have been performed over varying page sizes to measure its effect.

Thirdly, no testing of simultaneous application execution was performed, and hence the possible effect of context switching was simply ignored. Demand paging is particularly useful for real-world mobile device workloads in which several applications are executing at the same time (say, an MP3 player, a game, and an internet browser); however, all tests measure the effectiveness of the method for single application execution only.

These major shortcomings combined limit the extent to which we can declare the method a success for real-world workloads. Other minor shortcomings include:

- Little to no survey is made of existing work in the same space.
- No mention was made of the effect on power consumption savings resulting from the technique, which is an important consideration for mobile devices.
- It would have been interesting to test the technique on other embedded platforms with different cache characteristics.

5. Conclusion

This paper makes some important contributions to the problem of supporting an increasing number and size of executing applications on mobile device platforms. It presents a novel method of exploiting the two-stage reading characteristic of NAND-flash devices in order to reduce the unpredictably long page fault latency typically associated with demand paging. While the implementation and testing methodology invoked are sound, and the authors effectively deal with some concerns that arise from the methodology, there are several possible scenarios that are likely to occur under real-world workloads that are not discussed or tested. These drawbacks limit the extent to which we can declare the method a success.

Review of paper

Construction of a Highly Dependable Operating System

by J.N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum

Student Number:

1. Introduction

This paper seeks to address the problem of constructing a highly dependable operating system: it begins with a statement surmising that dependability is primarily affected by buggy drivers, and hence seeks to focus on limiting the impact of driver errors. The authors posit that moving device drivers outside the kernel is key to achieving this goal, and describe their design and implementation of this technique on a microkernel-based system, Minix.

2. Discussion of Approach

The authors approach to moving device drivers into userland begins with an analysis of the dependencies between drivers and the kernel on an existing microkernel-based system (Minix2). This analysis was performed by copying all files related to a particular driver to a separate directory, attempting to compile it in isolation from the kernel, and investigating the errors that result. 5 different types of dependencies between different parts of the system were discovered, enumerated, and described, and a general method of resolving each type was proposed. Furthermore, this general method of resolution was enhanced by analysing the *reasons* for the dependancies, resulting in a *functional classification* of drivers that describes precisely what it is they do when they interact with their dependencies. These methods of classification are sound and represent a good initial approach to the general problem of extracting subsystems from a larger system.

Next, the authors adopted an iterative approach to resolving these dependencies. System calls were added to the kernel for the various classes of dependencies in order to enable the drivers to perform the same functions via a syscall instead of a function call. Driver "servers" were created for each driver, and the interrupt handlers were stripped down such that they only send a message to the driver server alerting them to the occurrence of the interrupt. This process was undertaken for all drivers excepting the clock driver; the justification is that the clock driver is small and simple (and hence unlikely to contain bugs), and is needed inside the kernel for scheduling purposes. While this reasoning is sound, the authors may have considered also implementing the clock driver in user space so that drivers that depend on the clock driver have one less dependency on the kernel.

A "reincarnation server" was created that is responsible for the run-time configuration of devices: it can start, stop, and reload drivers, and apparently can detect defects and restart buggy drivers on error. While this is a novel mechanism, very little attention is paid to the method of detection and the reader is pointed elsewhere to determine how it was achieved - it appears this contribution was made in a different paper altogether.

In addition to this approach of extracting device drivers from the kernel, several improvements were made to the underlying microkernel itself. The IPC mechanism was greatly improved, with asynchronous messaging and the restriction that drivers perform a paired send and receive (to prevent the kernel from hanging) added. Servers are now given a bitmap specifying which classes of recipients (servers, drivers, kernel, and users) they can communicate with, and privileges were added to restrict the operations that servers/drivers can carry out. While these improvements are to be commended, and the description of why they were added may be valuable, they are not directly related to the issue of implementing usermode drivers. Moreover, a reader familiar with the state of the art in microkernel design may be left wondering why these features were not in the kernel in the first place! They describe well established methods in microkernel construction and little contribution is made here.

3. Discussion of Evaluation

Some initial testing and optimisation of the file system performance is presented and discussed, with experiments on the throughput of fixed and increasing block sizes in the new driver being compared to a fixed, low, block size in the old driver. This testing appears to show a 5% to 20% overhead of user-mode drivers compared to in-kernel drivers, though the exact figures are not discussed. It is surmised that this overhead can be mitigated by tuning other variables, such as the block size: this is a well established method. Some figures are also provided regarding the boot time of Minix 3 and the use of a RAM Disk, though it is not clear why. There is some discussion of possible future optimisations, which really just states that reducing the number of IPCs to the kernel is a good thing.

Apart from this, no scientific performance analysis or testing is performed at all. Their evaluation concludes with the claims that reliability of the operating system has been improved because the number and consequence of bugs has been reduced, and common failures can be recovered from, however absolutely no data is provided to warrant these claims. For example, while it is noted that drivers can be restarted, tested, and debugged without rebooting, which is good, no results of testing that utilises this mechanism are provided.

4. Major Shortcomings

This paper suffers from several major shortcomings:

- *Poor scientific method:* As described above, very little testing was undertaken and no data is provided to warrant the claims made.
- *Poor overall implementation technique:* a further contribution to this poor scientific method is that the combined approach of improving microkernel dependability and moving drivers to userland means we can't test the effectiveness of either in isolation. The user-mode drivers in Linux work that is referenced is a better example of analysing the impact of

moving device drivers to userland. Furthermore, while the improvements to the microkernel are commendable, it is a worry that the microkernel did not have these facilities in the first place, and it is probably not within the scope of this work to implement them. It is almost like attempting to evaluate the moving of device drivers to user mode in Linux but then overhauling the scheduling algorithm and virtual memory subsystems as part of the work.

- *Unoriginality*: Many of the techniques proposed to increase the reliability of operating systems are well established; some of these techniques are referenced in the discussion of related work but the authors fail to disclose just how related the work is. The authors claim that this is the first UNIX clone that is "far more fault tolerant than normal UNIX systems", though no data is provided regarding the fault tolerance of the implemented system and hence this claim is unfounded. POSIX conformance in a microkernel might be slightly novel, however the level of conformance is not stated and no further details are provided.

5. Conclusion

Unfortunately, there is very little of originality or novelty in this paper; while the overall ideas of implementing a microkernel and/or moving device drivers to userspace to improve reliability are sound, they are well established. While the opportunity to compare a microkernel with device drivers within the kernel (Minix2) to one with user-mode drivers (Minix3) may have been of some utility, unfortunately no real evaluation or performance analysis was undertaken and hence this opportunity was not exploited.

In general, the paper provides a nice description of some of the problems that this work encountered in attempting to achieve its goals, which may be of relevance to those who might encounter similar problems, and serves as a good introduction to the reasoning behind microkernel design for the lay reader. However, its poor scientific technique results in it contributing little to the field of operating system research as a whole.

That said, some minor contributions were made: the description of classes of dependencies between device drivers and the kernel might be useful elsewhere, and the reincarnation server appears novel, but its most novel aspects appear to be described in a different paper.