# Efficient Memory Control for Avionics Systems

Pinchas Weisberg, Yair Wiseman
Computer Science Department
Bar-Ilan University
Ramat-Gan 52900
Israel
{pinchas,wiseman}@cs.biu.ac.il

**ABSTRACT**

Modern aircrafts have seen a significant growth of avionics systems. Most of new aircrafts have systems for navigation, automatic flight control, collision avoidance systems, flight data recorder, weather radar system as well as communication and monitoring systems. All of these systems use embedded computer systems with increasing memory requirements. In order to reduce the electricity consumed by these systems, we suggest detecting the portions of the computer memory that is actually used. The other portion can be temporarily shut down and turn them on again when they are needed. Such a shut down can notably reduce the electricity consumption of the avionics systems.

## 1. INTRODUCTION

The software of Avionics systems has become more complex and larger during last years [20]. The programs of advanced avionics systems require larger memory allocation; however the maximum allocation is more often than not unneeded and most of the physical memory is usually unused [25].

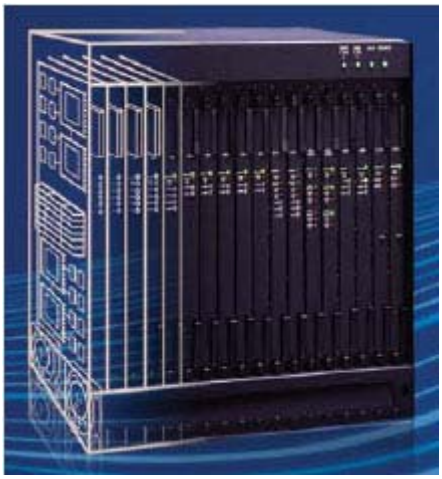

Figure 1. The rad-hard 72-MBit QDR II+ static random access memories of Cypress Semiconductor Corp.

The memory of avionics system is typically small and there are many techniques how to make it smaller e.g. [24]. An example of set of memory cards for avionics systems of Cypress Semiconductor Corp. can be seen in Figure 1. The cards should be suitable for use in high temperatures, withstand an impact of

accelerations much higher than g-force and resist a potential intense radiation so each card contains at most just 8MB which is much less than the standard SRAM cards we are familiar with. We would like to delve into the problem of how to detect unused portions of this memory so they can be temporarily shut down.Execution of a typical program can be divided into a sequence of phases - periods during which the program executes in some locality. Phase shift detection is useful in reducing architectural simulations, dynamic hardware configuration and online adaptive optimizations.

Many phase detection methods divide program execution into fixed length intervals and monitor some characteristics of program behavior during each interval. A phase shift is detected when there is a difference in the measured metric. Other phase detection methods are based on a high frequency of misses. When a program moves to a new locality, it misses the segments of memory that comprise the new locality.

This paper proposes to detect phases by monitoring Page Fault Frequency - the rate at which a process misses pages and generates page faults. When the frequency of page faults is high, it is an indication of a transition from one program phase to another. PFF phase detection can be handled by the OS and does not need offline profiling or special hardware support.

Phases are a property of a process, not of the system as a whole. In a multiprogramming environment execution of processes is interleaved. To detect phases online in a multiprogramming system, each process must be tracked individually. We describe an implementation of PFF phase detection that keeps phase information in the task structure of the process and calculates paging frequency in terms of its virtual execution time.

## 2. RELATED WORKS

The term *locality* describes the observed tendency of programs to refer to a small portion of their address space during significant periods of their execution time. Phases of locality are periods of program execution that have stable or slow changing of locality. The execution of a typical program can be described as a sequence of phases of locality

$$(l_1, t_1), (l_2, t_2), \ldots, (l_i, t_i), \ldots$$

where $l_i$ is the set of segments referenced during phase $i$ and $t_i$ is the duration of the phase. Segments are blocks of contiguous locations in the address space of the program. They are either units of information used in managing memory, such as pages and cache lines, or distinct entities in the source code detected by the compiler, such as loops and functions. A transition from one locality to another is not gradual, but rather characterized by excessive loading of segments that are needed for the execution to proceed in the next locality. Segments that are loaded early in the execution of a phase will be referenced with high probability during the rest of the phase. This is the principle underlying the performance of virtual memory

and caches.

To improve the performance of virtual memory, Denning [7] proposed the working set model for the behavior of programs. The known notation $W(t, \tau)$ denotes the set of pages referenced during a window of fixed size $\tau$ preceding time $t$. The working set size $\omega(t, \tau)$ is the number of pages in the working set. To minimize page faults, memory management must allocate to a process a quantity of pages that is enough to contain its current locality. In a multiprogrammed environment, there should be space in memory for the locality of every active process.

The phase behavior of an executing program is exhibited not only by its working set but also by microarchitecture dependent characteristics such as instructions per cycle (IPC), branch prediction and cache miss rate. Research has shown a correlation between phases of working sets and phases of hardware metrics [8], and a correlation between the phases of hardware metrics themselves [17]. The reason for these links is that program behavior is dependent on the code which is executed; and as a result of changes in the working set, program behavior changes, too. It was also found that most programs have repetitive behavior, with similar phases recurring during their execution.

Detecting the boundaries between phases has several applications. Reconfigurable hardware can be dynamically tuned for better performance and energy saving [8, 19, 3]. When a phase shift is detected, retuning is performed by trying a number of configurations and selecting the optimal one. When a repeated phase is identified, a saved optimal configuration is installed. Phase detection can be exploited to reduce architectural simulation time [18]. Instead of simulating an entire program run, a program can be divided into clusters of intervals having similar behavior. Then only some intervals representative of those clusters need to be simulated. Adaptive optimizations can benefit from program phases by triggering re-optimization when program execution changes significantly [12], or flushing a cache of code fragments on phase shift as done in Dynamo [2].

Many phase detection methods work by dividing program execution into fixed length intervals and monitoring some characteristics of program behavior during each interval [3, 8, 16, 18]. At the end of an interval, the measured value is compared with that of the previous interval. If the values differ by more than a predefined threshold, a phase change is detected. Hardware based approaches use physical characteristics: branch frequency, branch misprediction, cache miss rates, and IPC as a metric to identify phases of program behavior [3, 9]. Code based methods analyze the behavior of programs in terms of the code executed over time [8, 18, 19].

Dhodapkar and Smith [8] use working set signatures for controlling multi-configuration hardware. The working set touched during a fixed interval of program execution is collected by special hardware to form a working set signature. After each interval, software is invoked to compute the relative signature distance with

respect to the previous signature. If a working set change is detected, the hardware is reconfigured.

Sherwood et al. [18] use Basic Block distribution analysis to reduce architectural simulation to selected intervals. To find the phases of behavior of a program, its execution is divided into fixed length intervals. Basic Block Vectors are used to represent the frequency with which Basic Blocks of code are executed during a given interval. The Manhattan distance between vectors is then calculated to find the similarity between intervals, and similar intervals are grouped into a phase by a clustering algorithm. A program phase is regularly defined as a contiguous interval of execution during which a program metric is stable. They extend the notion of a phase to include all similar sections of execution regardless of temporal adjacency.

In a later paper, Sherwood et al. [19] describe an on-line method for phase detection and prediction that can be used for power management. To approximate the tracking of Basic Blocks used in the offline approach, they use special hardware to track the program counter of every committed branch and the number of instructions committed between the current branch and the last branch. After each profiling interval, the executed section is classified into a matching phase.

Hind et al. [13] formalized the problem of phase detection as an operation that takes as input a profile of program behavior and the following two parameters:
*Granularity* - specifies how a profile is partitioned into fixed length units.
*Similarity* - a function to compute if units of comparison are similar.
They demonstrated that changes to the values of these parameters can lead to the detection of significantly different phases.

Instead of finding similarity between windows of execution, phase detection can be based on high frequency of misses. When a program moves to a new locality it misses the segments of memory that comprise the new locality, unless they are already there from a previous reference.

HP Dynamo [2] is a dynamic optimization system that improves the performance of an instruction stream. It interprets the instruction stream until a hot instruction trace is identified. At that point, Dynamo generates an optimized version of the trace and inserts it into a software code cache. Subsequent encounters of the hot trace will cause control to jump to the corresponding cached fragment. To avoid the overhead of LRU storage, Dynamo employs a flushing heuristic to periodically remove cold traces from the fragment cache. A sharp rise in new fragment creation is an indication of a significant change in the working set of the program that is currently in the fragment cache. A complete fragment cache flush is triggered whenever Dynamo recognizes a sharp increase in the fragment creation rate.

Ratanaworabhan and Burtscher [14] proposed a phase detection method that is based on high frequency misses of Basic Blocks. To detect phases, an application is profiled on some input to generate a trace of Basic Blocks identifiers. The trace is then read and inserted into an infinite-size cache of Basic Blocks

identifiers, while misses in this cache are monitored. As a program transitions into a new phase, it starts a new working set of Basic Blocks which causes closely spaced misses in the cache. A transition that is followed by a burst of misses is identified as signaling a phase change, and a Basic Blocks signature for that transition is recorded. Recurring phases do not incur misses in the cache; they are detected by comparing each Basic Blocks transition to previously recorded signatures. Two signatures match if 90% of their Basic Blocks are the same. Although their method breaks ties with execution windows and a threshold to make a phase change decision, other parameters are introduced into the operation of phase detection. To define closely spaced misses, there is a need to decide on the number of misses and the time interval during which they occur. There is also a need for a parameter that defines the comparison of two signatures.

Another miss triggered method was suggested by Watanabe et al. [22], with the aim of visualizing objects in an object-oriented program. An LRU cache is employed for observing objects that are working for the current phase. When the cache is frequently updated, the beginning of a new phase is recognized.

Phase detection can be performed offline by profiling a training input or dynamically online. When done offline, phases may be different at run-time if input data has been changed, if the program has been optimized or if it is running on a different hardware. Also, the source code may not be available for offline analysis. Online methods do not need a prior profiling step and detect phases specific to the current execution. However, online detection adds time and space overhead to a program's execution, and usually needs additional hardware support.

This paper presents an online phase detection method which is based on monitoring Page Fault Frequency (PFF) - the rate at which a process generates page faults. Chu and Opderbeck [5] suggested the use of PFF as a page replacement algorithm. To monitor PFF they measure the time elapsed between the last and current page faults and compare it to a critical inter page fault time. If the page fault frequency lies above a given critical level, PFF replacement increases the amount of allocated memory. PFF replacement considers high frequency of page faults as an indication of an increase in the current working set. However, there is a major flaw in the PFF approach - it does not perform well when there is a shift to a new locality. During the transition periods, there is a rapid succession of page faults that causes PFF replacement to swell the resident set before the pages of the old locality are expelled [6, 21].

In contrast, PFF phase detection considers high PFF as signaling a transition to another working set and indicating a program phase shift. PFF phase tracking can be done as part of the OS management of time and memory and does not require multiple runs or additional hardware support. To detect phases according to PFF, the OS checks the number of page faults that occur during each time interval of the tracked process. When this number exceeds a threshold, it will signify a phase change. We propose implementing PFF phase detection as a kernel service. A process that needs to be phase tracked will request this service from the OS via a system call. When requesting the service, the process will pass to the OS parameters such as the interval length, the number of free pages and a function to call when a phase shift is detected (e.g. reconfiguring the hardware).

## 3. PFF PHASE DETECTION

Batson and Madison [4] defined a phase as a period during which a program accesses a subset of its address space. Following such a stable period, the program transitions to a different component will have a different subset of information. When moving to the next phase, the program must load its code and data into memory, unless it is already there. A phase change is thus typically characterized by a short interval during which there is high paging activity to bring in the new working set.

PFF phase detection uses this observation to detect the boundaries between phases. Figure 2 describes program execution according to this model, the executing program transitions among phases which consist of a short transition period followed by a stable period. For example, when a program calls a function it enters a phase. First it generates page faults in order to bring into memory the code necessary for the execution of the function; then it continues to execute the function without missing pages. To use PFF for phase detection, the executed program is divided into intervals of execution and page faults are counted for each interval. If the numbers of page faults that occur during an interval exceed a predefined threshold, then that interval signifies a phase shift. Page faults may occur on consecutive intervals; in that case the consecutive intervals are consolidated and considered as a continuous transition to a single phase.

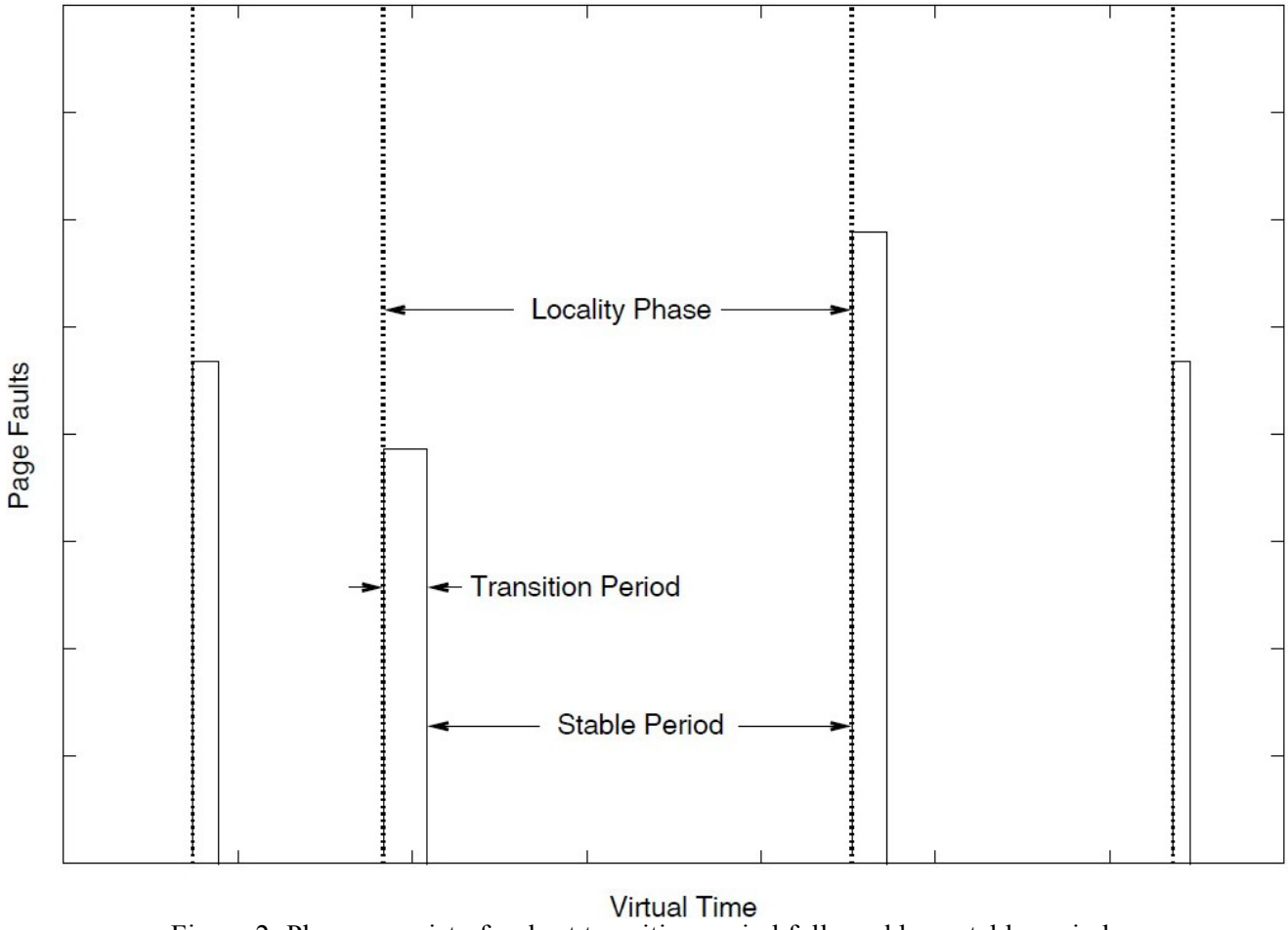


Figure 2: Phases consist of a short transition period followed by a stable period

As noted above, phases that are detected on the basis of similarity between windows are dependent on parameters [13]. Likewise, the method we propose, which is based on high miss frequency, must be provided with appropriate parameters for its operation. The length of the interval affects the number of detected phases. When the length is too large, consecutive phases become merged and the number of phases falls sharply. Reducing the length needlessly adds overhead to the detection algorithm. To ignore sporadic page faults, a threshold can be set on the number of faults that signify a page shift.

Another essential parameter in determining the results of PFF phase detection is the amount of memory available to the running program. If the amount is too large, a recurring phase or a phase consisting of pages that were referenced in the past may not be detected. Since the pages of the new phase may already be in memory, their reference will not cause page faults. For example, if the process is allocated memory which is enough to contain its maximum RSS, no recurring phase can be detected. If there is little free memory, pages comprising the current locality will be paged out and paged in during a single phase thus signifying false phases. Since an OS will allocate to a process all available memory that it needs, we must restrict the RSS virtually. Below, a way to virtually restrict RSS will be shown and the parameters will be discussed quantitatively.

The fact that different parameters give different phases does not mean phase detection is not useful. Phases as well as locality do not have an exact definition. At the extremes, locality may mean the whole program on the one hand and every program command on the other, in between there is a hierarchy of localities. Although locality is not well defined it works well in caches and virtual memory, we can not imagine the performance of current computers without them. Phase detection can also be useful for the applications mentioned earlier if tuned properly.

To visualize page faults as they occur in the OS, we instrumented the Linux memory manager to print the user time of the occurrence of a page fault to the kernel buffer. For this experiment we used a Pentium 4 machine having 512MB ram and running Linux-2.6.8. Since we were interested only in page faults that were generated by the test programs, the code was instrumented to prints page fault information only for a user with a specific user id (1000); and the test programs were run as a user having this user id. The kernel function that was modified is fault.c. If the current process (task) is run by user id 1000, the instrumented memory manager prints to the kernel log buffer the user time of the process that caused the page fault. The kernel differentiates between minor and major page faults. Faults for pages that are already loaded in memory are minor; faults for pages that have to be brought from disk are major. For the purpose of phase detection, there is no difference between them. They both indicate a page that is missing from the current working set.

In order to direct the kernel buffer output to a file, we have modified the sylog.conf file. To visualize phases according to PFF, the process must be forced to generate page faults even if those pages were loaded in previous phases. This can be done by restricting physically or logically the memory available to

a process. The OS will have to evict old pages in order to load new ones. In this experiment, to restrict free memory available to the process, we used a program that locks pages in memory according to a number of megabytes it receives as a parameter. Locked memory is not paged out to the swap area and can not be used by other processes. The quantity of the remaining free memory could be checked with the command free. Not all free memory is allocated to the process; some is reserved by the OS. The pages that are allocated to the process are shared by its code and data.

Graphs displaying the behavior of page faults and phases for two SPEC 2000 programs, gzip and mcf, are shown in Figures 3a and 3b. The x-axis denotes user time in terms of system clock ticks attributed to the process running this program. The y-axis represents the number of page faults that occurred during that clock tick. The programs were run to completion with reference input and free memory (as reported by free) was restricted to 50MB. Both programs have numerous faults during their initialization (we slightly moved the origin of the graphs to the right). The graphs demonstrate that execution of a program is characterized by short intervals of paging activity for loading the working set of new localities that are followed by stable periods with insignificant paging.

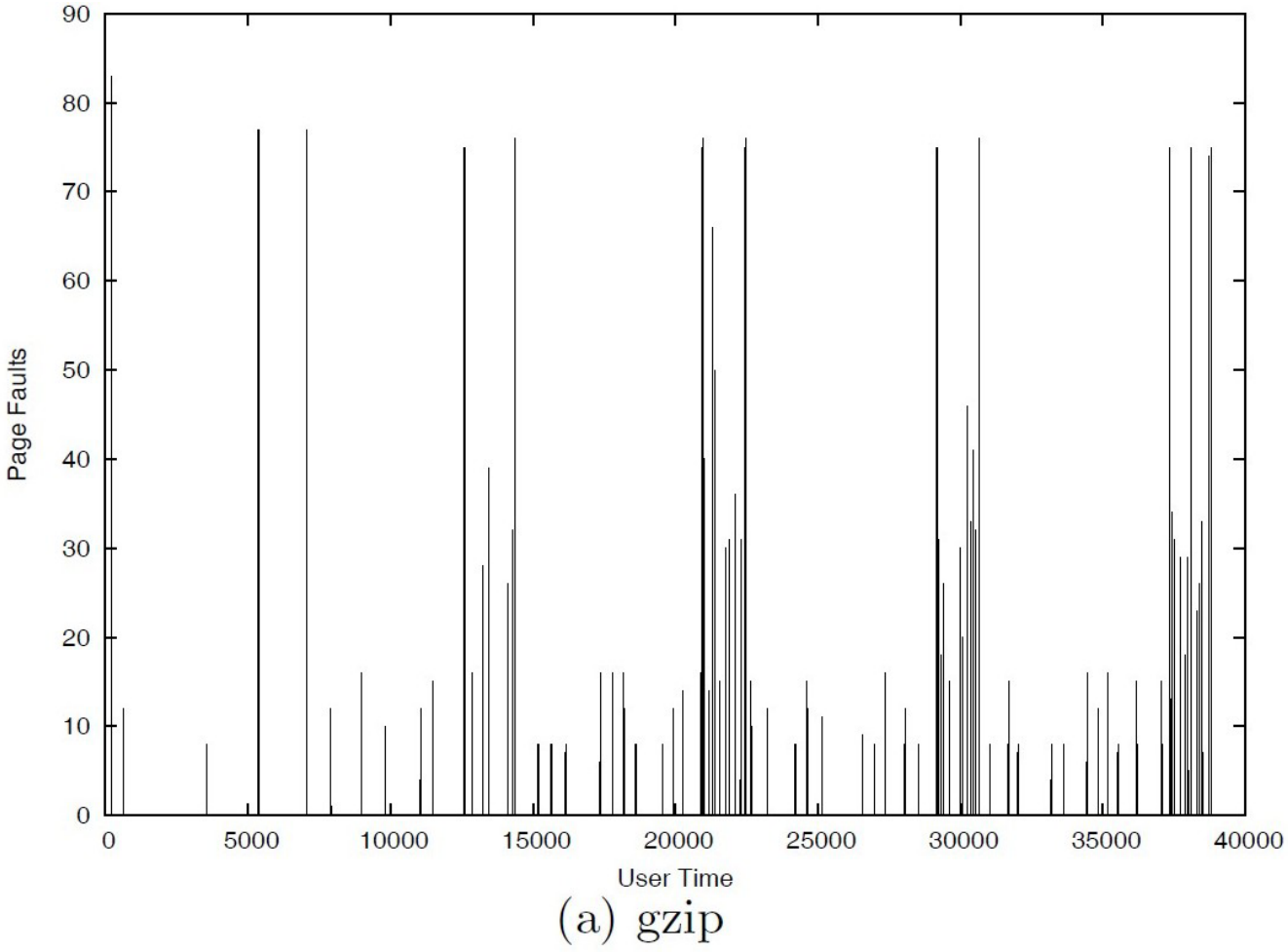


Figure 3a: Number of page faults occurring during each tick of process user time. Free memory was limited to 50MB - running gzip

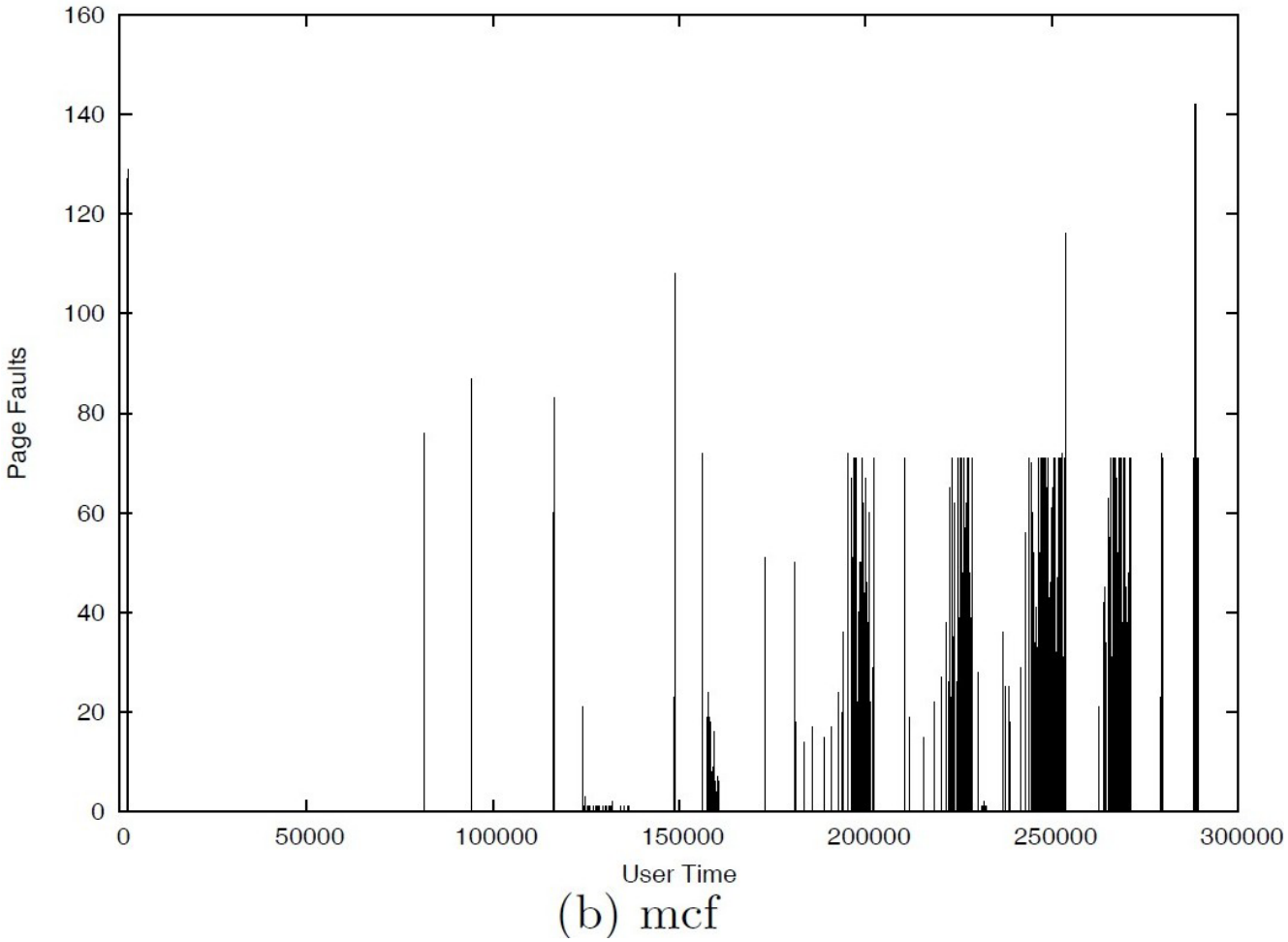


Figure 3b: Number of page faults occurring during each tick of process user time. Free memory was limited to 50MB - running mcf

There are zones in the graphs that seem as though constant paging is taking place, e.g. mcf at 250000 ticks. Figure 4a shows a zoom-in view of mcf revealing that the zone consists of just short intervals of paging. In Figure 4a we can also see that it is common for page faults to occur in bursts that last more than one tick. Intuitively, as a program starts to execute in a new locality, it references more and more pages of its working set until it reaches a stable state. It continues to execute for some time in this locality and then moves on to another locality. To eliminate the false phases that would result if each tick were considered separately, consecutive clock ticks that have page faults are merged as shown in Figure 4b and therefore indicate the beginning of a single phase.

In the experiment, we used memory locking to restrict memory. However, to practically use page faults for phase detection, we do not want to lock pages and thus prevent the application from using available memory. We need a mechanism that precisely controls the amount of memory a process can use without incurring page faults, while restricting additional memory only virtually. In addition, the time interval we used was the system clock tick. A typical frequency of the system timer in the Linux kernel is currently 1000 ticks per second. For a computer running at 10,000 MIPS, it means 10M instructions per interval. We will see later that this interval may be too large.

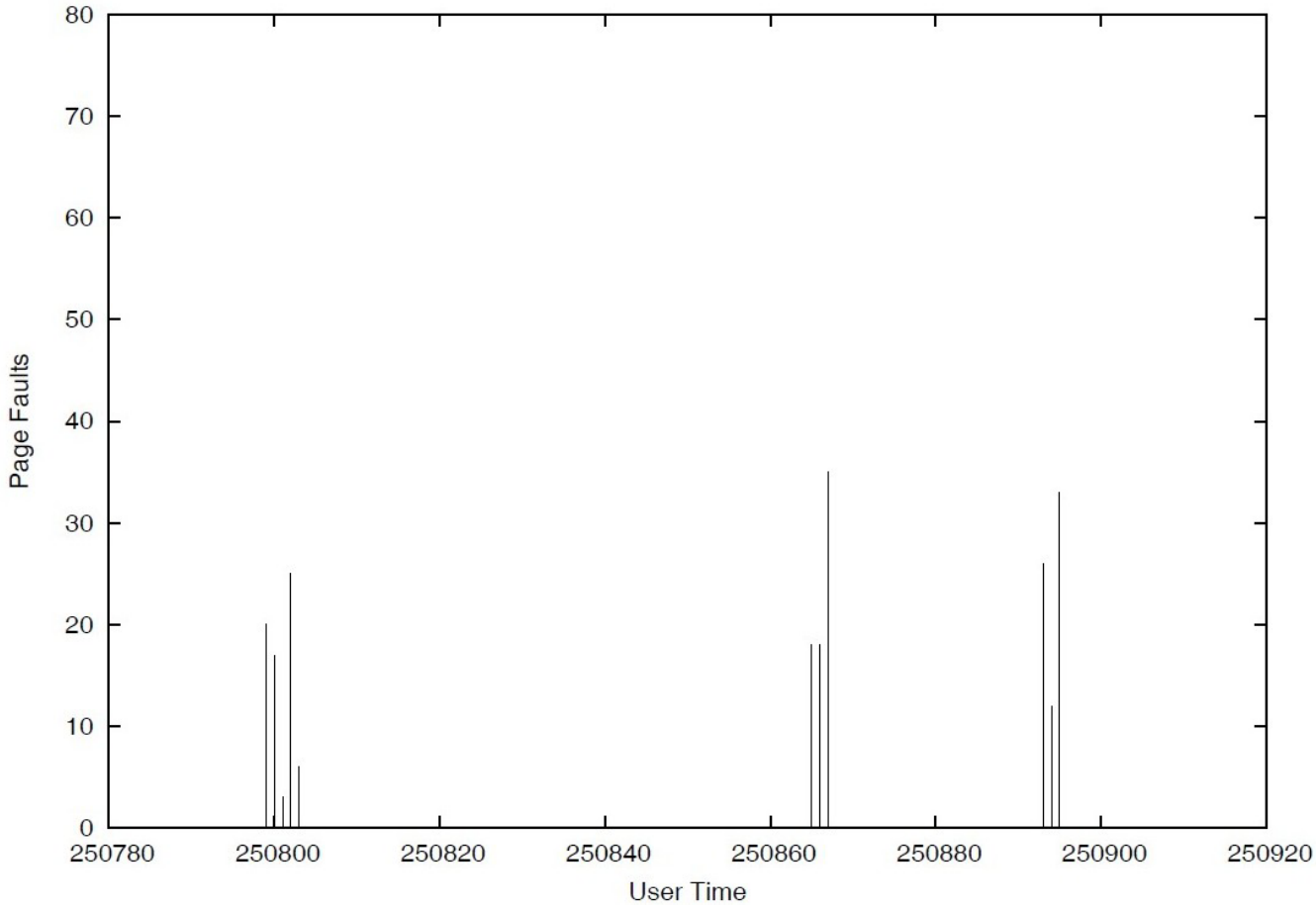


Figure 4a: A zoom into an interval of mcf – consecutive ticks with page faults

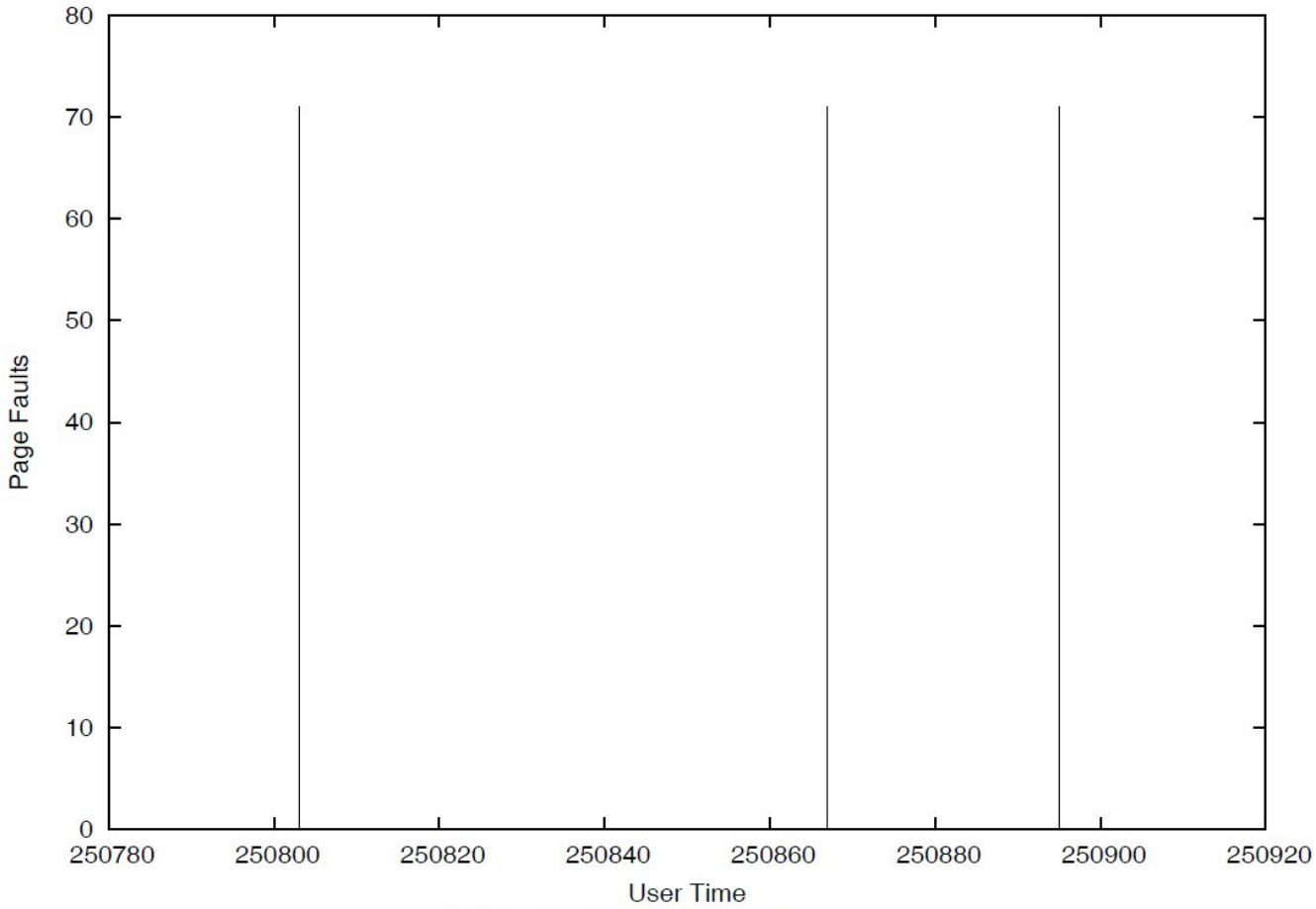


Figure 4b: A zoom into an interval of mcf – after merging.

### 3.1 Amount of Free Pages

An essential parameter in determining the results of PFF phase detection is the amount of memory available to the running program. If the amount is too large, a phase consisting of pages that were previously loaded may not be detected. Its pages may already be in memory and their reference will not cause page faults. If there is little free memory, pages comprising the current locality will be paged out and paged in during a phase, thus signifying false phases. On the other hand, too little memory will cause consecutive intervals to generate page faults, the intervals will be merged; thus reducing the number of detected phases.

The proper number of pages is the amount containing the locality of the phase that is currently detected, but this amount is changing throughout the execution of the program. In the following analysis, we will see that for phases that do not recur closely to be detected, the amount needs to be within a range but the exact number is not significant.

As a program is executing, transition occurs from locality to locality. Let

$$. . . , l_i , . . . , l_j , . . .$$

be the sequence of sets of pages referenced during each phase of locality and assume the sets are disjoint. Let

$$. . .,|l_i|, . . .,|l_j|, . . .$$

be the number of pages of the corresponding locality sets. Then, in order to detect the transition to phase lj, the following has to be true:

$$RSS \leq \sum_{k=i+1}^{j-1} | l_k |$$

If the next phase to be detected is a recurring phase, then the amount of free memory must not be larger than the combined amount of memory occupied by all phases separating the previous occurrence from the current occurrence. This assures us that the previous occurrence was evicted from memory.

And in order to detect lj as only one phase, the following condition is also needed:

$$RSS \geq |l_j|.$$

The amount of free memory must not be smaller than the resident set of the detected phase; otherwise pages will be swapped during its execution causing it to be detected as more then one phase.

As long as the amount of free pages available to the program is within the range

$$| l_j | \leq RSS \leq \sum_{k=i+1}^{j-1} | l_k |$$

all phases will be detected.

The need to restrict memory applies only to pages containing the code of the program, not to pages of data.

The PFF phase detection algorithm we describe is based on paging that result from executing new code not from referencing new data. In the next section, we discuss how to achieve memory restriction virtually.

### 3.2 Virtual Resident Set Size

A mechanism to virtually limit free pages to some number can be implemented by using the valid bit of page table entries. The valid bit is used by Operating Systems to simulate the referenced bit. Operating Systems usually use the referenced bit during page replacement to find pages that were not recently used. In architectures that do not support the referenced bit in hardware [23], Operating Systems simulate it in software by turning of the valid bit and examining this bit instead. When the page is referenced, a page fault occurs and the Page Fault Handler sets back the valid bit. Turning of the valid bit can be used in our case to limit the number of pages that can be referenced without forcing a page fault. The Operating System will turn of the valid bit of all code pages of the tracked process, except for the number of free pages. However, the free pages need to be those pages containing the recent locality. If the pages of the process could be arranged in LRU order, then we could turn of the valid bit of all pages except those at the head of the list. To achieve this, we can incorporate the management of the valid bit in the page replacement of the Operating System.

Now we describe how it can be done in Linux. Page replacement in Linux is global and is based on two LRU-like lists, called the active list and the inactive list. The objective is for the active list to contain the working set of all processes and for the inactive list to contain reclaim candidates [11, 10]. Each physical page in Linux is represented by a page descriptor; the lru fields in the page descriptor stores pointers to the next and previous elements of the LRU lists.

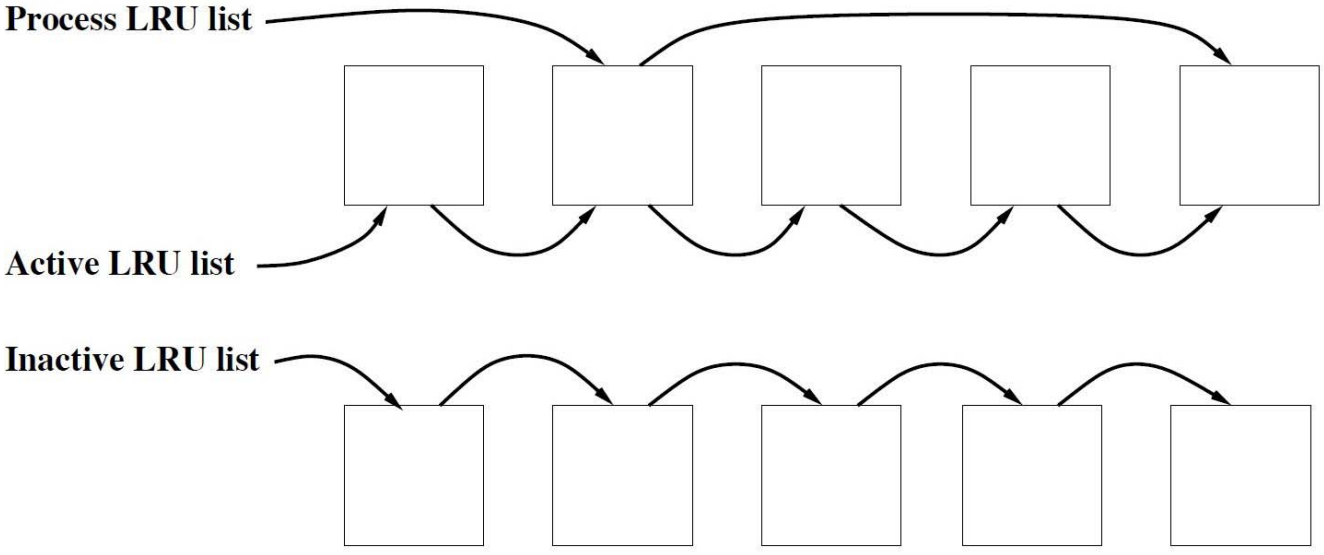


Figure 5: Pages belonging to processes are grouped into two lists, the active list and the inactive list. A list of pages belonging to a specific process was added.

To keep the pages of a specific process in LRU order, its pages can be inserted into a list by adding a process_lru pointer field to the page descriptor as illustrated in Figure 5. Using this pointer, an LRU-like

list of active pages belonging to a specific process can be maintained. When the kernel moves a referenced page of the active list to the head of the list, it will also move it to the first position of the process-lru list. To keep the number of free pages of a process bounded, pages that are in the head of the active process-lru list will be marked valid by setting the valid bit in their corresponding PTEs, other pages will be marked invalid. Whenever a new page is inserted at the top of the list, the valid page furthest from the top will be marked invalid, thus keeping the number of valid pages constant. The process-lru list has to be maintained only for pages of the code of the process not for pages of data. Code and data can be distinguished because they belong to different memory areas of the process. This limiting mechanism does have an overhead in the extra page faults that are generated and in managing the process-lru list.

### 3.3 Tracking PFF per process

Phases depend on the code executed by a process and are thus a property of the process executing the code and not of the system as a whole. In a multiprogramming environment, execution of processes is interleaved; therefore there is no meaning in detecting system phases. A process may be preempted within a phase because its time slice expired or for other reasons and rescheduled later to continue the phase. Tracking phases has to be done per process not per system. To detect the phases of a specific process according to PFF, we have to consider only page faults generated by that process and calculate time intervals in terms of its processing time.

The method we use to detect a phase change is to divide program execution into intervals and monitor the number of page faults that occur during each interval. If the number of page faults that occur during an interval exceeds a predefined threshold, the interval is considered a faulting interval; otherwise it is considered a non-faulting interval. The threshold is set to filter out sporadic faults. One or more faulting intervals that follow a non-faulting interval define the start of a new phase. We keep the information needed to track phases in the task structure of the tracked process and consider time intervals during which the tracked process runs.

The period of the timer needs to be adjusted to the time it takes to run the number of instructions that were chosen for the interval. However, the resolution of the regular Linux timer is not adequate. The frequency of the system timer is between 100 and 1000 ticks per second giving at best a resolution of one millisecond. For computers running at 10,000 MIPS, a higher resolution timer is needed. Fortunately, high resolution timers with microsecond resolution are already supported by the Linux kernel.

## 4. EXPERIMENTS

To find out how the amount of free pages and interval length influence PFF phase detection and to evaluate the usefulness of the phases detected, we used several tools from the SimpleScalar toolset [1]. To track the dynamic behavior of the simulated programs, the tools have been modified to print interval statistics per a specified number of instructions. We configured the tools to report statistics at every interval of 100K committed instructions and derived longer intervals from the output of this interval. We used sim-cache to simulate PFF phase detection, sim-bpred and sim-outorder to examine the detected phases.

We simulated ten programs from the SPEC CPU2000 benchmark suite using binaries precompiled for the Alpha ISA. They are the six integer programs crafty, eon, gcc, gzip, perlbmk and vpr and the four floating point programs applu, equake, galgel and mesa. All programs were run from start to completion with reference inputs.

### 4.1 Choosing the size of free memory

The sim-cache simulator has been configured to simulate a fully associative L1 instruction cache with an LRU replacement policy. The block size of the cache was set to 4096 and the number of cache lines varied from 4 to 256. All other caches were disabled as their output was irrelevant to our experiments.

In the memory hierarchy, the relation between disk storage and main memory is like that between main memory and cache. Virtual memory uses a page size of typically 4K bytes, the pages are fully associative and its replacement policy is usually LRU-like. The misses reported by sim-cache for a specified number of cache lines as configured above correspond to the page faults that would occur to a process running with an equivalent number of physical pages.

The output of sim-cache gave us a series of the page faults that occurred during each interval of the executing program. By looking for one or more faulting intervals that follow a non faulting interval, we can find the number and length of phases.

### 4.2 Free Pages and Interval Length

We tried various combinations of free pages and intervals to determine the appropriate parameters for PFF phase detection. Different parameters give different number of phases and phase length. The intuition behind this is that since a program comprises a hierarchy of phases, as free memory is reduced, the resolution of detected phases gets finer. The desired number and length of phases is dependent on the application for which the program is partitioned into phases. For the purpose of dynamically tuning cache sizes, reconfiguration latency has to be taken into account; phases must be longer than this latency. We also have to consider the overhead of switching between different cache configurations. Changing a cache parameter is likely to cause flushing of dirty entries; phases must execute long enough to benefit from reconfiguration. On the other hand, adaptive optimizations are related to program constructs and may profit from smaller phases.

Figure 6 shows graphically, page faults that occur during each interval of 100K instructions throughout the execution of the program crafty. Subfigures (a) - (f) correspond to running the program with the number of free pages ranging from 4 to 128. As the number of page frame is increased the number of page faults decrease, until at 128 pages (512K) almost the whole program is read into memory at the start of execution and there are hardly any page faults from then on.

Table 1 shows the results of phase detection for the ten programs listed above. Each row gives the number

of detected phases when restricting free memory to a different number of 4K pages. The number of free pages is at all times a power of 2, ranging from 4 pages to 128 pages. Each program has three rows showing the results when using intervals of 100K, 1M and 10M instructions.

We saw in Figure 6 that the frequency of page faults increases with diminishing free memory, as expected; however, when we look at the first raw of Table 1 we see that the program crafty reaches the maximum of detected phases when using 16 pages, 4 and 8 pages detect just few phases. It turns out that when using 4 and 8 pages for the execution of crafty, the system suffers from thrashing [15]. Since free memory is smaller than the working set, pages that are moved out of memory to make room for new pages are brought in soon after. Because almost all intervals contain many page faults, the algorithm can hardly find a faulting interval that follows a non faulting interval. On the other hand, using 128 and 256 pages gives also a small number of phases. A large part of the program is read into memory in an early stage of its execution and stays there, referencing phases later does not impose page faults. Figure 7 shows graphically the number of phases detected for free pages ranging from 4 to 128 pages. Because the programs are of different length and different phase behavior, for each program the maximum of phases detected was normalized to 100.

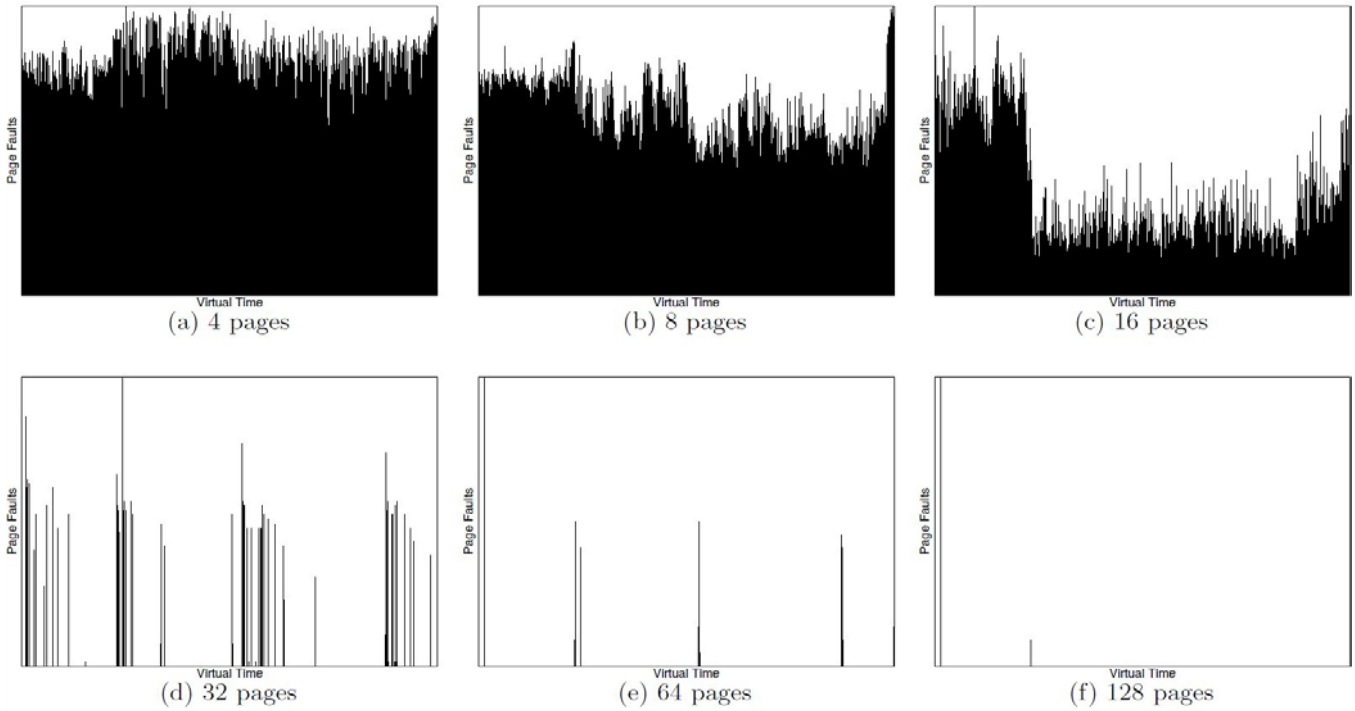


Figure 6: Number of page faults that occur during each interval of 100K instructions when execution the program crafty with free pages ranging from 4 to 128

We see from Table 1 and from Figure 7 that PFF phase detection is quite sensitive to the parameter of free pages. This means that in order to use it effectively, the parameter needs to be known in advance and passed to the algorithm. As an alternative, the rate of detected phases can be monitored to dynamically adjust the parameter of free pages to a value within the range 4 to 32.

| Program | Interval | 4 Pages | 8 Pages | 16 Pages | 32 Pages | 64 Pages | 128 Pages | 256 Pages |
|---|---|---|---|---|---|---|---|---|
| | 100K | 7 | 7 | 157797 | 301 | 26 | 8 | 8 |
| crafty | 1M | 1 | 1 | 1300 | 193 | 13 | 3 | 3 |
| | 10M | 0 | 0 | 0 | 99 | 11 | 2 | 2 |
| | 100K | 7019 | 7373 | 7373 | 7024 | 6323 | 25 | 25 |
| applu | 1M | 4911 | 5265 | 5265 | 4916 | 4565 | 17 | 17 |
| | 10M | 1379 | 2435 | 2435 | 2435 | 2435 | 9 | 9 |
| | 100K | 2 | 65 | 48 | 9640 | 14 | 12 | 10 |
| eon | 1M | 0 | 0 | 1 | 5628 | 4 | 3 | 3 |
| | 10M | 0 | 0 | 1 | 716 | 4 | 3 | 3 |
| | 100K | 13199 | 443 | 1303 | 8 | 7 | 7 | 7 |
| equake | 1M | 10396 | 403 | 1262 | 7 | 7 | 7 | 7 |
| | 10M | 3816 | 266 | 260 | 6 | 6 | 6 | 6 |
| | 100K | 7550 | 4051 | 583 | 344 | 51 | 26 | 26 |
| galgel | 1M | 6377 | 3366 | 569 | 332 | 48 | 25 | 25 |
| | 10M | 1329 | 1381 | 342 | 202 | 36 | 21 | 21 |
| | 100K | 7153 | 5302 | 3607 | 7650 | 2934 | 1269 | 987 |
| gcc | 1M | 49 | 385 | 397 | 503 | 408 | 365 | 321 |
| | 10M | 4 | 15 | 36 | 24 | 72 | 86 | 81 |
| | 100K | 85275 | 15311 | 33 | 5 | 5 | 5 | 5 |
| gzip | 1M | 20772 | 10217 | 21 | 3 | 3 | 3 | 3 |
| | 10M | 11 | 18 | 21 | 3 | 3 | 3 | 3 |
| | 100K | 2002 | 2002 | 151006 | 3005 | 7 | 6 | 6 |
| mesa | 1M | 1147 | 1147 | 103914 | 2150 | 5 | 4 | 4 |
| | 10M | 0 | 0 | 2 | 1002 | 4 | 3 | 3 |
| | 100K | 357 | 373 | 696 | 5699 | 3822 | 36 | 20 |
| perlbmk | 1M | 21 | 26 | 32 | 32 | 1129 | 13 | 9 |
| | 10M | 7 | 8 | 11 | 14 | 309 | 6 | 4 |
| | 100K | 2215 | 2120 | 70 | 29 | 14 | 14 | 14 |
| vpr | 1M | 1088 | 885 | 47 | 22 | 12 | 12 | 12 |
| | 10M | 522 | 424 | 16 | 11 | 8 | 8 | 8 |

Table 1: Number of detected phases for free memory ranging from 4 to 128 pages and for intervals of 100K, 1M and 10M instructions.

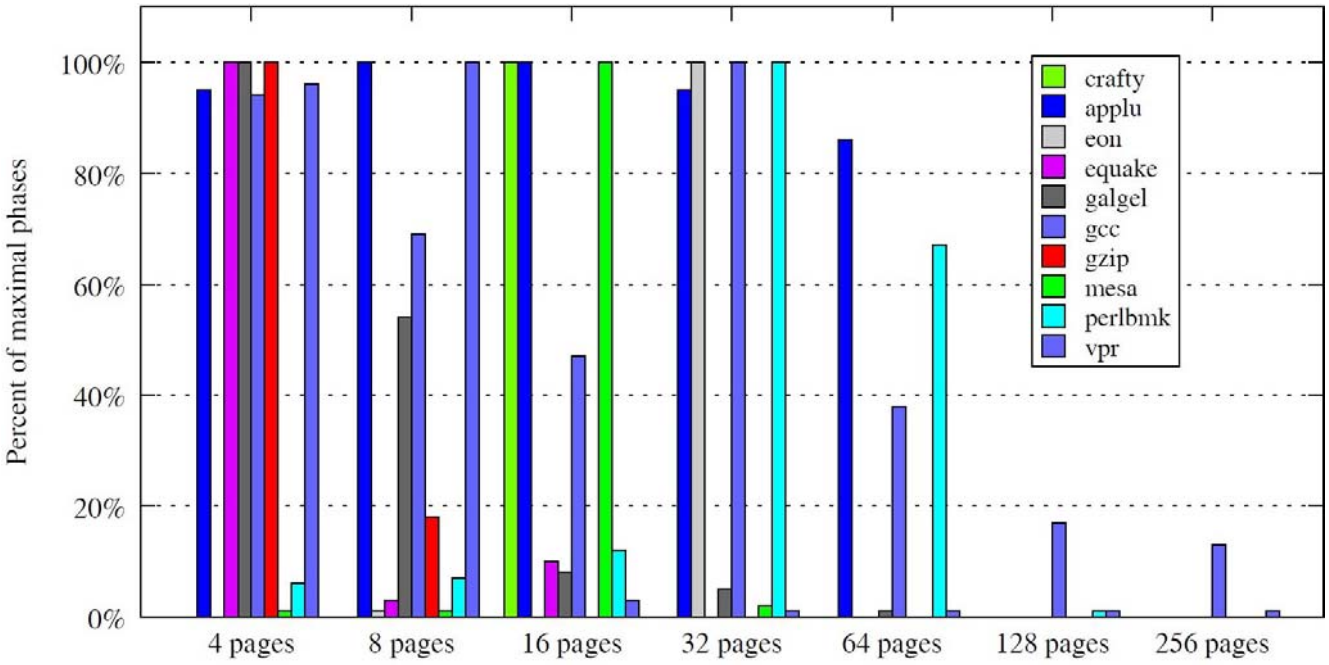


Figure 7: Number of phases for 4 to 128 free pages. For each program the maximum of phases detected was normalized to 100.

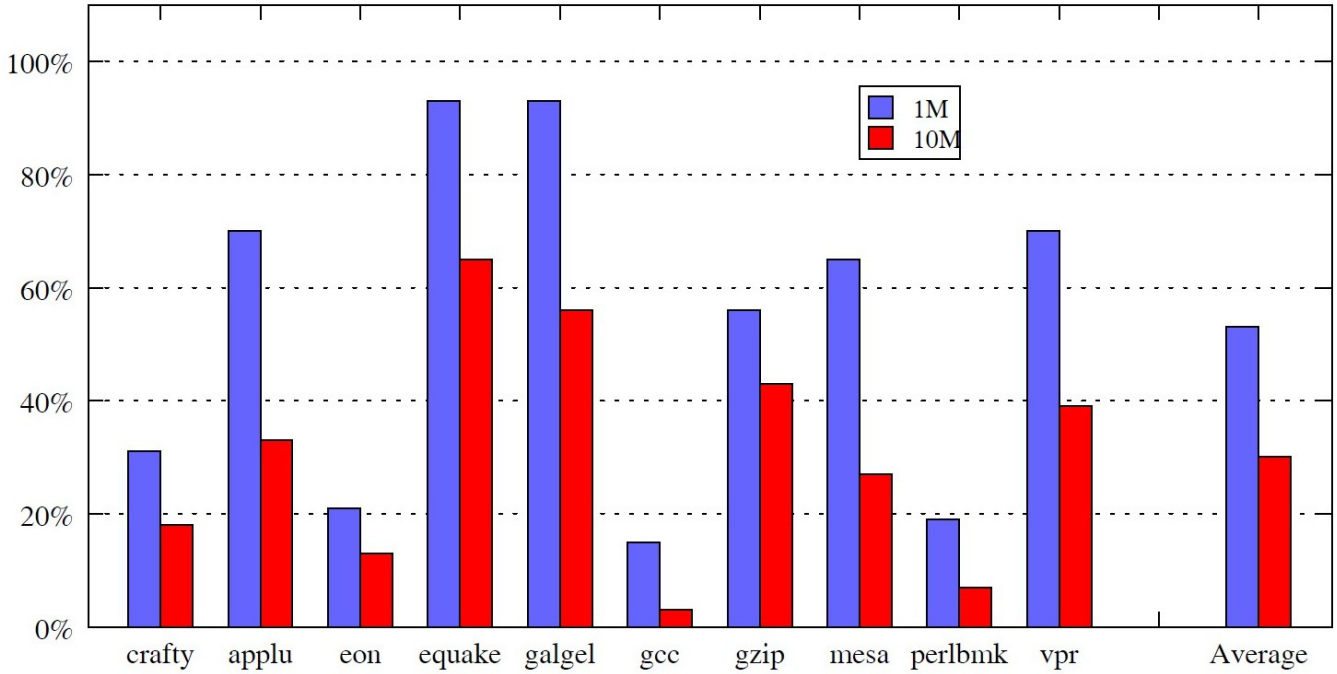


Figure 8a: Percent of phases detected when using 1M and 10M intervals relative to 100K intervals – for each program the average of all free page sizes 4 - 128.

Another important parameter is the interval length. Table 1 shows results for three interval lengths 100K, 1M and 10M instructions. Figures 8a and 8b show graphically the percent of phases detected when using 1M and 10M intervals relative to 100K intervals. Figure 8a shows for each program the average of all free page sizes 4 – 128; whereas Figure 8b shows for each program the average of the three page sizes giving the maximal number of phases. Because page sizes that produce only few intervals are not of much use,

the results shown in Figure 8b are more reliable. We see that the number of detected phases falls on the average to 50% when using 1M intervals and to 19% when using 10M intervals. The reason is that half of the phases detected by PFF with an interval of 100K are shorter than 1M instructions and most are shorter than 10M instructions. When a phase is shorter than the interval, it can not be detected by the algorithm because page faults generated by the next interval prevent it. Since intervals are formed by the locality of program constructs such as loops and functions, the results found above are reasonable

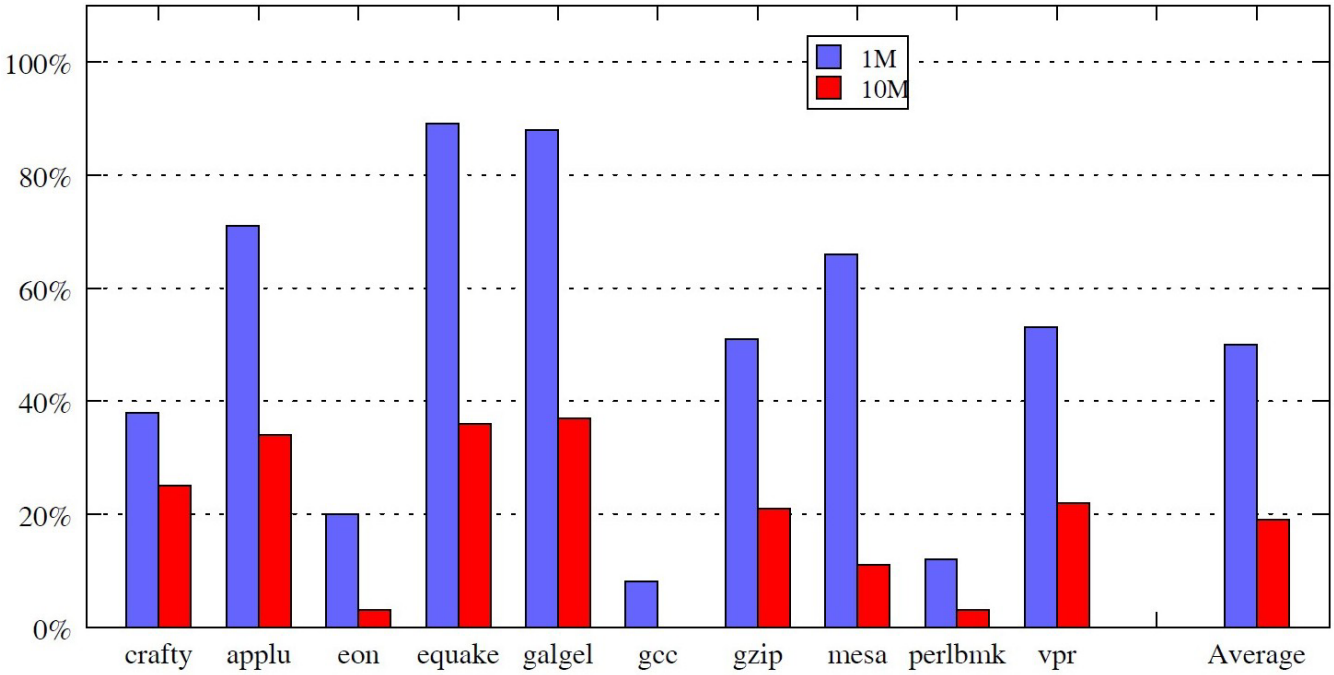


Figure 8b: Percent of phases detected when using 1M and 10M intervals relative to 100K intervals – for each program the average of the three page sizes giving the maximal number of phases.

### 4.3 Similarity of Metrics within Phases

To evaluate the effectiveness of phases detected by PFF, we used sim-bpred to measure the instructions per branch (IPB) metric for the ten SPEC CPU2000 programs that we examined. The output of sim-bpred was a series of IPB values for intervals of 100K instructions. For each group of intervals detected by the output of sim-cache as a phase, we calculated the coefficient of variation (CV) (the ratio of the standard deviation to the mean) in IPB. We then calculated the average of the CVs of all phases and compared it to the CV of all program intervals. For each program, the output of IPB was partitioned into phases according to the output of sim-cache that gave maximal phases. The results in percent are shown in Figure 9a. The average of all programs is 41% for the whole programs and 11% for the phases, a reduction to about a fourth.

As a further evaluation, we used the timing simulator sim-outorder to measure the instructions per cycle (IPC) metric. For this test we did not need the output of sim-cache since sim-outorder simulates the cache. We configured the cache of sim-outorder for each program with the same cache configuration we used for sim-cache that gave maximal phases. All other parameters of sim-outorder (e.g. branch predictor) were not

changed from their default value. The output of sim-outorder was a series of misses and IPC values for intervals of 100K instructions. Using this output we partitioned the program into phases and calculated the average of the CVs in IPC of all phases and compared it to the CV of all program intervals. The results in percent are shown in Figure 9b. The average of all programs is 39% for the whole programs and 12% for the phases, a reduction to about a fourth.

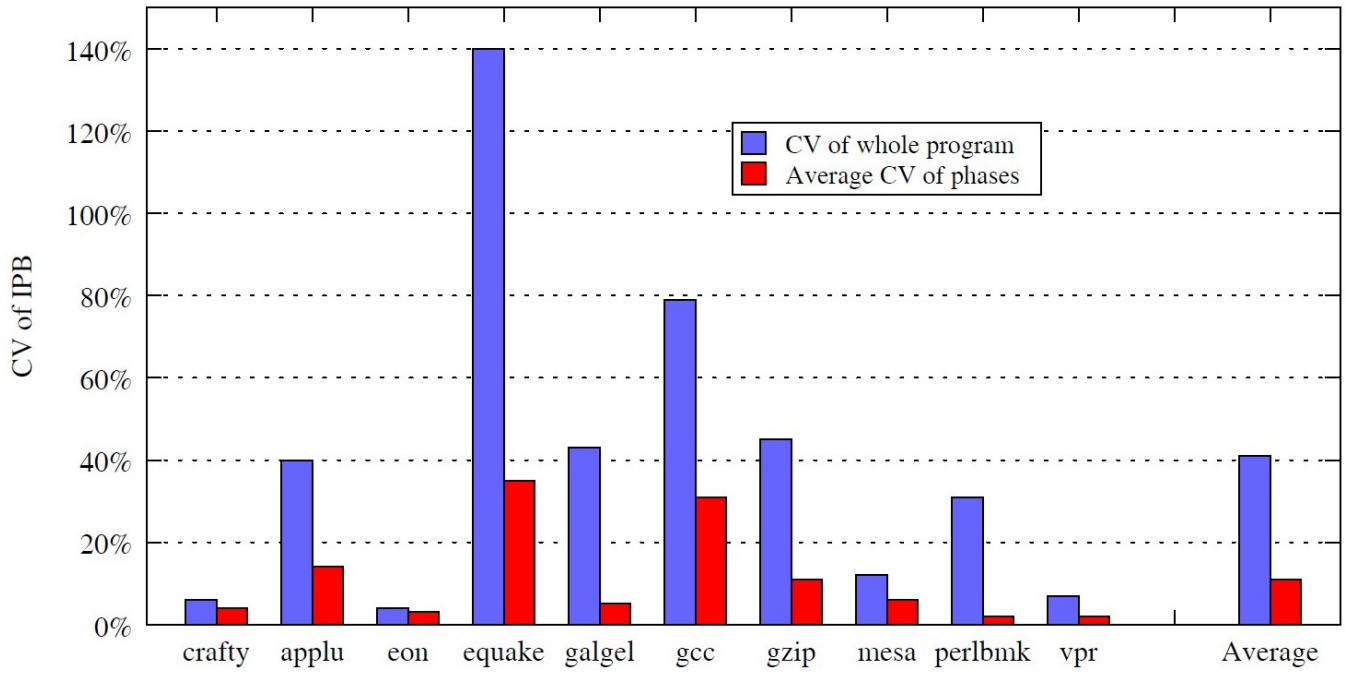


Figure 9a: Average CV of all phases and CV of whole program. – IPB

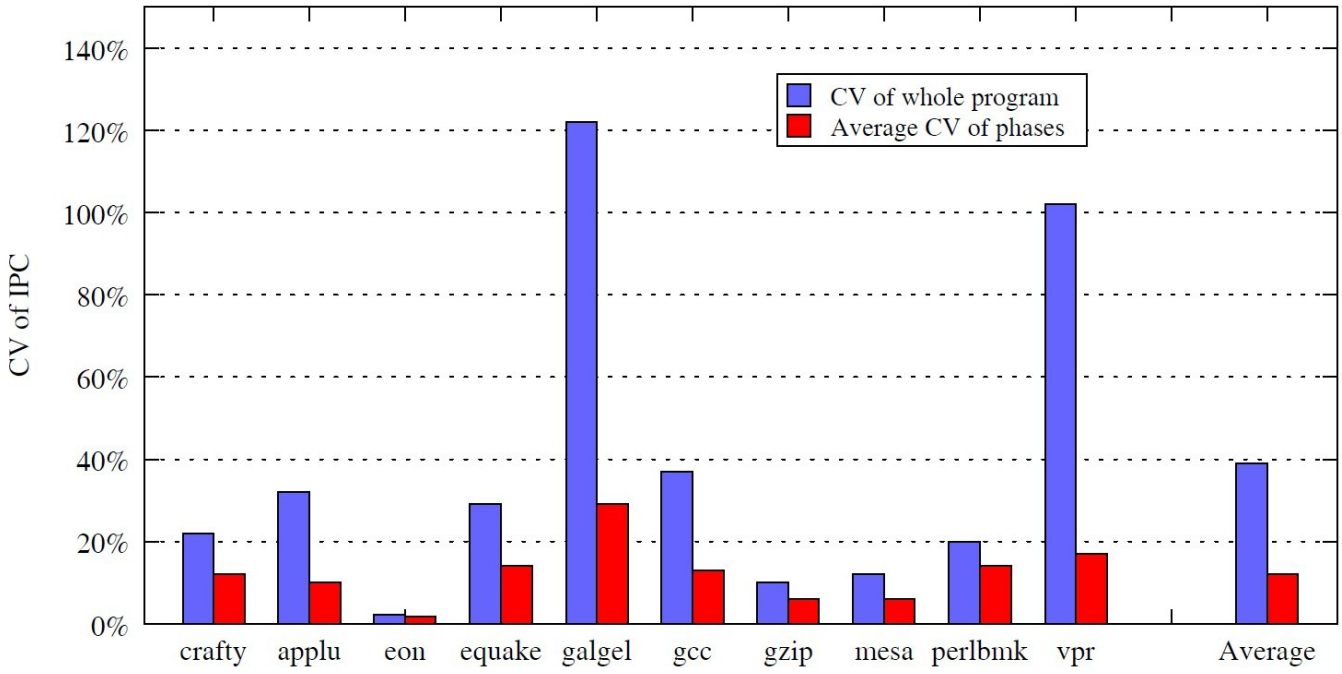


Figure 9b: Average CV of all phases and CV of whole program. – IPC

## 5. CONCLUSIONS

The utmost memory capacity of avionics systems is typically superfluous and most their physical memory is more often than not, effectively idle. The actual memory use can be detected by monitoring Page Fault Frequency. We used the propensity of programs to have a high frequency of page faults when a transition from one program phase to another occurs. When a phase switch is detected, an inspection of which portion of the memory can be shut down is taken. There is no need for a special hardware for Page Fault Frequency phase detection; it can be handled by the Operating System. Such a detection system can be very beneficial for an efficient use of memory.

## REFERENCES


[1] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. Computer, 35(2):59-67, 2002.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In ACM
SIGPLAN Notices, pp. 1-12, 2000.

[3] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, pp. 245-257, New York, NY, USA, 2000.

[4] A. P. Batson and A. W. Madison. Measurements of major locality phases in symbolic reference strings. In
SIGMETRICS '76: Proceedings of the 1976 ACM SIGMETRICS conference on Computer performance modeling measurement and evaluation, pp. 75-84, New York, NY, USA, 1976.

[5] W. W. Chu and H. Opderbeck. The page fault frequency replacement algorithm. AFIPS '72 (Fall, part I): Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, part I, pp. 597-609, 1972.

[6] F. D. and Y. yoo Yih. Vsws: The variable-interval sampled working set policy. IEEE Trans. Softw. Eng.,
9(3):299-305, 1983.

[7] P. J. Denning. The working set model for program behavior. Commun. ACM, 11(5):323-333, 1968.

[8] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. SIGARCH Comput. Archit. News, 30(2):233-244, 2002.

[9] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, page 220, Washington, DC, USA, 2003.

[10] M. Gorman. Understanding the Linux Virtual Memory Manager. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[11] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile, pp. 439-450. Morgan Kaufmann, 1994.

[12] T. Kistler and M. Franz. Continuous program optimization: A case study. ACM Trans. Program. Lang. Syst., 25(4):500-548, 2003.

[13] V. R. M. Hind and P. F. Sweeney. Phase detection: A problem classification. Technical Report 22887, IBM Research, 2003.

[14] P. Ratanaworabhan and M. Burtscher. Program phase detection based on critical basic block transitions. In ISPASS '08: Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of

Systems and software, pp. 11-21, Washington, DC, USA, 2008. IEEE Computer Society.

[15] M. Reuven & Y. Wiseman, Medium-Term Scheduler as a Solution for the Thrashing Effect, The Computer Journal, Oxford University Press, Swindon, UK, Vol. 49(3), pp. 297-309, 2006.

[16] X. Shen, Y. Zhong, and C. Ding. Predicting locality phases for dynamic memory optimization. J. Parallel

Distrib. Comput., 67(7):783-796, 2007.

[17] T. Sherwood, Calder, and B. Calder. Time varying behavior of programs. 1999.

[18] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, pp. 45-57, 2002.

[19] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. SIGARCH Comput. Archit. News, 31(2):336-349, 2003.

[20] N. Sözen and E. Merlo, Adapting Software Product Lines for complex certifiable avionics software, 3rd International Workshop on Product Line Approaches in Software Engineering (PLEASE-2012), pp. 21-24, Zurich, Switzerland, June 2012.

[21] W. Stallings. Operating systems (3rd ed.): internals and design principles. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[22] Y. Watanabe, T. Ishio, and K. Inoue. Feature-level phase detection for execution trace using object cache.

In WODA '08: Proceedings of the 2008 international workshop on dynamic analysis, pp. 8-14, New York, NY, USA, 2008.

[23] Y. Wiseman, A Pipeline Chip for Quasi Arithmetic Coding, IEICE Journal - Trans. Fundamentals, Tokyo, Japan, Vol. E84-A No.4, pp. 1034-1041, 2001.

[24] Y. Wiseman and A. Barkai, Smaller Flight Data Recorders, Journal of Aviation Technology and Engineering, Vol. 2(2), pp. 45-55, 2013.

[25] H. Zhou, M. Toburen, E. Rotenberg, and T. Conte, Adaptive Mode Control: A Static-Power-Efficient Cache Design, Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques, Sep. 2001, pp. 61-72.