Predictable paging in real-time systems: a compiler approach[*]

Isabelle Puaut   Damien Hardy
Université Européenne de Bretagne / IRISA, Rennes, France

Abstract

Conventionally, the use of virtual memory in real-time systems has been avoided, the main reason being the difficulties it provides to timing analysis. However, there is a trend towards systems where different functions are implemented by concurrent processes. Such systems need spatial separation between processes, which can be easily implemented via the use of the Memory Management Unit (MMU) of commercial processors. In addition, some systems have a limited amount of physical memory available. So far, attempts to provide real-time address spaces have focused on the predictability of virtual to physical address translation and do not implement demand-paging.

In this paper we propose a compiler approach to introduce a predictable form of paging, in which page-in and page-out points are selected at compile-time. The problem under study can be formulated as a graph coloring problem, as in register allocation within compilers. Since the graph coloring problem is NP-complete for more than three colors, we define a heuristic, which in contrast to those used for register allocation, aim at minimizing worst-case performance instead of average-case performance. Experimental results applied on tasks code show that predictability does not come at the price of performance loss as compared to standard (dynamic) demand paging.

1 Introduction

Memory management is a major concern when developing real-time and embedded applications. Predictability issues have resulted in real-time systems being most of the times strictly *static*, avoiding dynamic allocation/deallocation and virtual memory. However, as these systems are getting increasingly large and complex, there is now a need to escape from this strictly static memory management.

In particular, a recent trend towards systems where different functions are implemented by concurrent processes can be observed, for instance in Integrated Modular Avionics (IMA) systems or for the automotive industry. Such systems need spatial separation between processes, which can be easily implemented via the use of the Memory Management Unit (MMU) of commercial processors. Moreover, cost constraints may limit the amount of physical memory available.

Virtual memory consists in using hardware support (MMU, Memory Management Unit) to compute at run-time where an address (called *virtual* address) is located in *physical* memory. The virtual address space of a program is divided up into fixed-size units called *pages*. The mapping between virtual pages and physical pages is stored in data structures scanned by the MMU at every memory access (page tables stored in RAM and a fully-associative cache named TLB, for Translation Look-aside Buffer to speed-up accesses to page tables). When a program attempts to reference an unmapped page, the MMU notices that the page is unmapped and traps to the operating system; such a trap is called a *page fault*. Upon a page fault, the operating system loads the page from disk (*page-in*). Symmetrically, when there is no free physical page anymore, a *replacement* policy implemented by the operating system selects one physical page to evict from main memory (*page-out*). Modified pages have to be copied back to disk before being evicted; this is done either in the page fault handler or by an independent process depending on the operating system. The interests of virtual memory are twofold: *(i)* it provides spatial protection between processes, since each process has a private page table; *(ii)* it allows to execute tasks whose address space is larger than the capacity of physical memory, since pages are paged-in and out on demand, in a transparent manner to the programmer.

In real-time systems, it is crucial to prove that tasks will meet their temporal constraints in all situations, including the worst-case situation. Therefore, *predictability* of performance is as important as average-case performance. One should be able to predict the Worst-Case Execution Time (WCET) of pieces of software for the system timing validation [13]. Virtual memory raises predictability issues at two levels:

– Level of *address translation*: getting the mapping between virtual to physical pages requires a TLB lookup

[*]This study was partially supported by the french National Research Agency project Mascotte (ANR-05-PDIT-018-01)

plus possibly a page table lookup if the mapping is absent from the TLB. The duration of address translation is hard-to-predict, because: (i) not all mappings can be stored in the TLB because of its limited capacity, thus it is difficult to know which mappings will be served by the TLB and which ones will require a page table lookup; (ii) the TLB is shared between concurrent processes;

– Level of *paging activity*: knowing whether or not a reference to a virtual page will result in a page fault is hard to predict. This is because physical memory is shared between concurrent processes, and in general any physical page regardless of its owner process may be selected by the page replacement algorithm. In addition, the replacement algorithm is never strict Least Recently Used (LRU), because it would be too costly to maintain the ordering of page references using current MMUs. Furthermore, common replacement policies may be arbitrarily complex and in general not, or not enough, documented, because they are implemented in software inside the operating system. For instance, a process may be used to update the disk for dirty pages; some physical pages may be temporarily locked during a page-in or a page-out.

So far, attempts to provide real-time address spaces have focused on the predictability of virtual to physical address translation [11, 2]. Demand-paging is carefully avoided: all physical pages are voluntarily created in memory at process load-time, or wired in memory, to avoid unpredictability due to page faults. Surprisingly, little effort has been devoted to reconciliate the benefits of the paging activity (in particular its ability to execute programs larger than main memory) and predictability. Providing some form of *predictable paging* seems to us very important, in a context where the volume of software embedded into devices grows and cost considerations limit the amount of available memory in some systems.

This paper makes a first step in that direction. We propose a compiler approach to introduce a predictable form of paging, in which page-in and page-out points of virtual pages are selected at compile-time, thanks to the static knowledge of possible references to virtual pages. Our approach operates on a single task[1] and currently considers references to *code* only. The problem under study can be expressed as a graph coloring problem, heavily used in compilers for register allocation. Since the graph coloring problem is NP-complete for more than three colors, we define a heuristic, which in contrast to those used for register allocation, aims at minimizing the worst-case performance instead of the average-case performance. Experimental results applied on tasks code show that predictability does not come at the price of performance loss as compared to standard demand paging.

[1]To be used in a multi-task system, one may use our approach in combination with a memory partitioning scheme, like the one presented in [15].

The rest of the paper is organized as follows. Related work is surveyed in Section 2. Section 3 formulates the problem of off-line selection of page-in and page-out points as a graph coloring problem and proposes a WCET-oriented graph coloring heuristic. Experimental results applied on code are given in Section 4. Implementation issues and directions for future work are dealt with in Section 5.

2 Related work

A very predictable approach called *overlaying* [12] was used before the hardware support for virtual memory became common. The software is divided into pieces called overlays. When an overlay is needed, the overlay is explicitly loaded into memory by the program, overwriting an overlay that was no longer needed. Overlaying techniques, while highly predictable, were in most systems non automatic, requiring manual work from the programmer to define the overlays.

Virtual memory appeared in the sixties to provide spatial isolation between concurrent processes and allow programs larger than the amount of physical memory to execute. The definition of efficient page replacement algorithms has received considerable attention in the seventies. The optimal page replacement algorithm as defined in [1] evicts the page that *will* not be used for the longest time. Obviously, optimal replacement cannot be implemented in practice because it requires an exact knowledge of future memory accesses. Instead, existing page replacement strategies exploit the knowledge of past references to guess future ones. The mostly used replacement algorithms are approximations of the Least Recently Used (LRU) replacement. LRU evicts the page that has not been used for the longest time. Approximations of LRU are used instead of strict LRU because strict LRU would be too costly to implement using standard hardware. Indeed, most MMUs use only 2 bits per page: U (for Used) and M (for Modified). The U (resp. M) bit is set by the MMU at every reference (resp. modification); these two bits are reset by software to implement efficient in the average-case but approximated LRU replacement. The difference of our work with existing page replacement strategies is that we predetermine page-in and page-out points at compile-time rather than at run-time as in standard demand paging.

So far, demand paging is avoided in real-time operating systems. Demand paging simply cannot be implemented in real-time operating systems running on processors without MMU. For processors with a MMU, some systems like Spring [10] use the MMU for protection between processes only. In Spring, all the pages of a program are loaded at process start such that pages faults do not occur. Furthermore, the number of pages used by a process is limited, such that all address translations are served by the TLB without resorting to page table lookups. Other real-time systems like RT-Mach [16] and real-time extensions of POSIX provide a system call to wire pages in memory for real-time tasks. VxWorks [17] provides a library to control address translation, but does not provide any support for demand paging. Bennett and Audsley [2] focus on page table structure for address translation predictability. Our work builds on these studies ensuring predictability of address translation, and focuses on the predictability of the paging activity.

One may view the predictability issues caused by paging systems as identical to those raised by caches. Many methods have been designed in the last years to estimate worst-case execution times on architectures with instructions and/or data caches [9, 5], for different cache structures and replacement policies. The tightest predictions are obtained for LRU replacement. In contrast, pseudo round-robin and random replacement yield to looser timing estimates [6]. Analysis methods originally defined for caches cannot be directly transposed to paging systems. The main reason is that page replacement policies are more sophisticated and less documented than cache replacement policies because they are software-implemented. Moreover due to hardware-software interactions, page replacement policies are not strict LRU. Thus, to the best of our knowledge no attempt to statically analyze page replacement policies has been made so far. In this paper, for the above-mentioned reasons, we do not try to predict the worst-case behavior of dynamic paging. Instead, we predict page in and page out points at compile time thanks to the knowledge of possible future page references.

Graph coloring was recently used by Li et al in [8] for automatically managing transfers between scratchpad memory and off-chip memory, with performance considerations in mind. In contrast, our work focuses on transfers between RAM and disk and is predictability-oriented rather than performance-oriented.

3 Predictable paging: a graph coloring approach

This section is devoted to the modeling of predictable paging as a graph coloring problem. We first make an informal parallel between predictable paging and graph coloring in paragraph 3.1. Paragraphs 3.2, 3.3 and 3.4 then describe our algorithm for static selection of page-in and page-out points in more details.

3.1 Informal description

Assuming that referenced virtual pages are known statically, which is common in real-time systems for predictability considerations, it is possible to define the program regions where these pages are used. We will term such regions *Webs*. A web for a virtual page *vp* is the set of basic blocks that reference *vp* (see Fig. 1.a, in which three virtual pages are used).

Two webs are said to *interfere* if their intersection is not empty. An *interference graph* can then be defined: a node in the interference graph corresponds to a web, and an (undirected) edge corresponds to an interference between two webs (see Fig. 1.b).

Defining a mapping between virtual and physical pages amounts to assigning a physical page to every web, assuming a limited number of physical pages. It is equivalent to coloring the interference graph, with a limited number of colors, one per physical page (see Fig. 1.c, with two physical pages represented by colors black and grey). Obviously, it might happen that the interference graph is not colorable. Then, webs have to be split, resulting in extra page-ins and page-outs. This iterative process has to be repeated until the interference graph becomes colorable.

The mapping between virtual and physical memory pages is a straightforward result of the coloring process (see rectangles in Fig. 1.d). Similarly, the location of page-in and page-out points is a direct result of the coloring: page-in points are on the web incoming edges, and page-out points are on the web outgoing edges (see small bullets in Fig. 1.d, shown for virtual page 2 only).

This problem is similar to register allocation in compilers [3], where webs represent variable usage and colors the processor physical registers. Spill code is the register allocation equivalent of page-ins/page-outs in our problem.

In the following, we will use the term N-colorable to note an interference graph colorable using N colors. The degree of a node in the interference graph will denote its number of neighbors in the interference graph.

3.2 Webs and interference graph

Webs and interference graph are defined for every task. An inter-procedural Control Flow Graph (CFG) is constructed at compile-time. There is one node per basic block and an edge for every possible sequence between two basic blocks (caused by conditional and unconditional branches, function calls and function returns). The set of virtual pages that may be referenced by a basic block is assumed to be known at compile-time.

Let us note $G = (V, E)$ the program CFG, with V the set of basic blocks and E the set of transitions between basic blocks.

Ideally, if the number of physical pages was large enough, a virtual page should be paged-in before its very first use, and paged-out after its last use only, regardless of the references to the virtual pages in-between. As a consequence, the start point of the coloring process is the interfer-

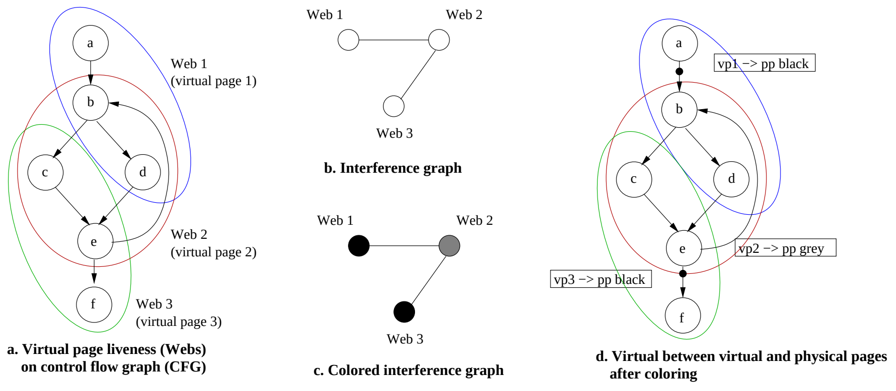


Figure 1. Predictable paging formulated as a graph coloring problem

ence graph between the largest live-ranges of virtual pages, called *maximal webs* in the following. Maximal webs are split in the course of the coloring process when the interference graph is not colorable.

More formally, a maximal web for a virtual page vp is defined as the set of basic blocks either using a virtual page vp or belonging to an execution path between two basic blocks using vp (see figure 2). Let $pages(x)$ denote the set of virtual pages used by basic block x, and $sucs(x)$ denote the set of direct or indirect successors of x in the CFG, the maximal web of virtual page vp is defined as follows:

$$
\begin{aligned}
&maxweb(vp) = \{x \in V \mid vp \in pages(x) \vee \\
&\quad \exists (y,z) \in V \mid vp \in pages(y) \wedge vp \in pages(z) \\
&\qquad \wedge x \in sucs(y) \wedge z \in sucs(x)\}
\end{aligned}
$$

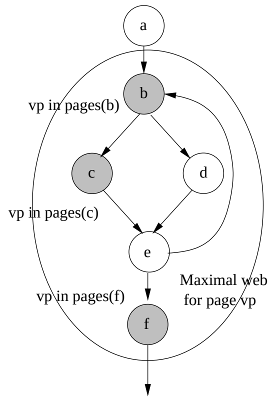


Figure 2. Maximal webs

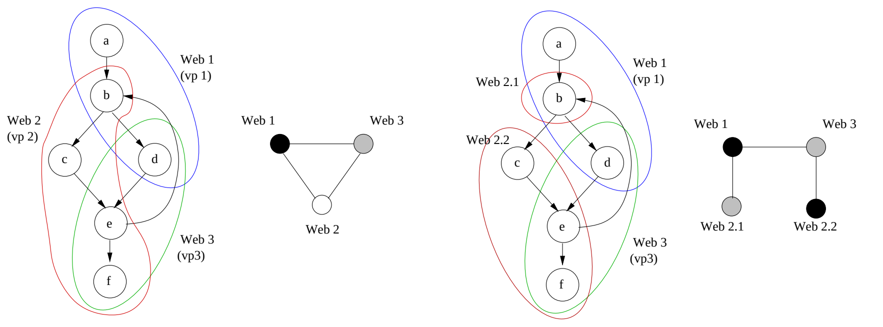
3.3 Graph coloring algorithm

The start point of the coloring algorithm is the set of maximal webs. Once a web is colored, its assigned color is not changed (greedy algorithm). Every web is assigned an integer *weight* used as a heuristic in the coloring process, and defining the ordering of web coloring. In the following, constant *nbcol* will represent the number of colors (number of physical pages). The algorithm assumes that every basic block uses a number of virtual pages lower or equal than *nbcol*, but obviously it supports a total number of used virtual pages much larger than *nbcol*.

The algorithm for coloring the interference graph is sketched below. The data structures used by the coloring algorithm are first built (lines 6 and 7). Function *Assign-Weight* called at line 8 assigns a weight to every maximal web (see paragraph 3.4 for a description of the weight functions). The algorithm then iteratively tries to color the interference graph through a call to function *Color* (loop at lines 10 to 14). Function *splitWeb* splits the web having caused the coloring process to fail, if any.

```
 1  var CFG: tCFG;      // Control flow graph
 2  IG: tIG;            // Interference graph
 3  Webs: set of tWeb; // Set of webs
 4  ncWeb: bool;        // First non colorable web
 5  begin
 6    Webs := BuildMaximalWebs(CFG);
 7    IG := BuildInterferenceGraph(Webs);
 8    AssignWeight(IG);
 9    ncWeb := Color(IG,nbcol);
10    while (ncWeb $\neq$ NULL) do
11          IG := splitWeb(ncWeb,IG,nbcol);
12          AssignWeight(IG);
13          ncWeb := Color(IG,nbcol);
14    done
15  end
```

The pseudo code of function *Color* is presented below. Functions *getWebsGreaterOrEqual* (resp. *getWebsLowerThan*) returns the set of webs in the interference graph with a degree greater or equal to (resp. lower than) parameter *nbcol*. Function *Color* scans and colors webs by decreasing weight value (loop in lines 11 to 14). Function *AssignColor* called at line 13 assigns a color to web w, such that the assigned color is different from those of the inter-



a. Non 2−colorable interference graph

b. After splitting Web 2, the graph becomes 2−colorable

Figure 3. Web splitting

fering webs, if possible. The second loop colors naturally colorable webs, those with a degree lower or equal to the number of colors.

```
funct Color(IG: tIG, nbcol: integer): Web
  var w1,w2: tWeb;   // Particular webs
  colorable: bool;   // True if IG colorable
  ncIG: tIG;         // Not yet colored subset of IG
  Webs: set of tWeb; // Set of webs
  begin
    // Color webs with degree $\geq$ nbcol
    colorable := true;
    Webs := getWebsGreaterOrEqual(IG,nbcol);
    ncIG := notColored(Webs);
    while (¬empty(Webs) and colorable) do
         w1 := RemoveWebWithMaxWeight(Webs);
         colorable := AssignColor(w1,nbcol);
    done
    // Color remaining webs
    Webs := getWebsLowerThan(ncIG,nbcol);
    while (¬empty(Webs)) do
         w2 := RemoveFirstWeb(Webs);
         AssignColor(w2,nbcol);
    done
    if (¬colorable) then return w1; else return NULL;
  end
```

The web splitting procedure *splitWeb*, whose algorithm is not detailed for space considerations, splits the first non colorable web detected by procedure *Color*. Let us assume that a particular web w is to be split and interferes with a set of already colored webs. Procedure *splitWeb* extracts from w the fully connected sub-web containing the smaller set of interfering nodes. An illustration of procedure *splitWeb* is given in figure 3. Assuming that webs 1 and 3 are already colored, the interference graph is not 2-colorable and thus web 2 has to be split. The smallest set of nodes interfering with webs 1 and 3 is $\{b\}$, which is excluded from web 2, thus split in webs 2.1 and 2.2. In this small example the resulting interference graph becomes 2-colorable.

3.4 Heuristics for web coloring

Webs are colored by decreasing weight order. As our target applications have real-time constraints, we are primarily interested in minimizing their worst-case timing requirements. As a consequence, the weight function for a web w, called $W_{wcet}(w)$ hereafter, accounts for the impact of the web on the task worst-case execution time. $W_{wcet}(w)$ is defined as follows:

$$W_{wcet}(w) = \sum_{x \in w | vp \in pages(x)} frequency(x) \qquad (1)$$

where vp is the virtual page associated to web w and $frequency(x)$ is the number of references to basic block x along the program worst-case execution path (WCEP). Execution frequencies along the worst-case execution path are a direct result of WCET estimation tools using Integer Linear Programming (ILP) to estimate WCETs [14]. Since the WCEP may change due to coloring decisions, it is re-evaluated regularly. The re-evaluation period can be parameterized from 1 (re-evaluation at every coloring) to ∞ (no re-evaluation).

A second weight function, named $W_{nesting}(w)$ was also defined for comparison purpose. Contrary to W_{wcet}, $W_{nesting}(w)$ does not use any frequency information and thus can be used without any WCET estimation tool available. $W_{nesting}(w)$ is a common heuristic used in compilers for register allocation [3]. It favors webs with deeply nested basic blocks. $W_{nesting}(w)$ is defined as follows:

$$W_{nesting}(w) = \sum_{x \in w | vp \in pages(x)} 10^{nesting(x)} \qquad (2)$$

with $nesting(x)$ the nesting level of basic block x. The nesting level of a basic block in the main function is 1. A basic block in the loop body of a loop l enclosed in a loop l' is assigned a nesting level of $nesting(l') + 1$.

Name	Description	Code size (bytes)
minver	Matrix inversion for 3x3 floating point matrices	4516
compress	Compression of a 128 x 128 pixel image using discrete cosine transform	3056
matmult	Multiplication of two 50x50 integer matrices	804
crc	CRC (Cyclic Redundancy Check)	1232
jfdctint	integer implementation of the forward DCT (Discrete Cosine Transform)	3604
qurt	the root computation of a quadratic equation	1748
fft	Fast Fourier Transform	3520

Table 1. Benchmark characteristics

4 Performance evaluation

We are interested in evaluating the *worst-case* timing behavior of programs. Estimation of WCETs is completed using static program analysis. Experimental conditions are described in paragraph 4.1. Experimental results are given in paragraph 4.2.

4.1 Experimental setup

WCET estimation. Our experiments were conducted on MIPS R2000/R3000 binary code. The WCETs of tasks are computed by the Heptane[2] static WCET analysis tool [4]. One may configure Heptane to estimate WCETs using either: a tree-based method, through a bottom-up traversal of the syntactic tree of the analyzed C programs; an IPET (Implicit Path Enumeration Technique) method, generating a set of linear constraints from the program control-flow graph. Here, the IPET WCET estimation method is used, because we are interested in the frequency of basic blocks along the WCEP, which is a direct result of IPET estimation methods.

Heptane includes hardware modeling capabilities to estimate WCETs for programs running on architectures with instruction caches, (in-order) pipeline, simple branch prediction. In this paper, the hardware analysis phase of Heptane is bypassed and a constant 1 cycle execution time per instruction is considered. A page-in time of 1 million cycles is assumed. The page-out delays are zero because only code pages are considered.

Unless explicitly stated, the W_{wcet} weight function is used, and the WCEP is not re-evaluated during graph coloring. We use pages of 128 bytes; page size is small to stress the paging activity even on rather small benchmarks.

Benchmarks. The experiments were conducted on seven benchmarks, whose features are summarized in Table 1. All benchmarks but compress are benchmarks maintained by the Mälardalen WCET research (http://www.mrtc.mdh.se/projects/wcet/benchmarks.html). Compress is from the UTDSP Benchmark (http://www.eecg.toronto.edu/).

[2]Heptane is an open-source static WCET analysis tool available at *http://www.irisa.fr/aces/software/software.html*.

4.2 Results

The main performance metric used in the following paragraphs is the number of page-ins along the worst-case execution path. Such a number is a direct output of the Heptane WCET estimation tool.

Influence of number of physical pages. The left part of figure 4 depicts the impact of the number of physical pages on the number of page-ins along the WCEP. For space considerations, results for only four benchmarks are given, the raw numbers for all benchmarks are given in an appendix available on demand. The figure shows that the smaller the number of physical pages, the higher the number of page-ins along the WCEP.

The right part of figure 4 gives the *measured* number of page faults for a demand-paging system using a LRU replacement policy, obtained using a small operating system running on a simulated MIPS processor[3]. A comparison of the two figures shows the same evolution of the number of page loads when making the number of physical pages vary. In particular, the number of page-ins gets unacceptably high for a small number of physical pages (trashing phenomenon). The measured number of page-ins is in most cases lower than the estimated one, because WCET estimation tools estimate the longest path executed and not only one path. Furthermore, WCET estimation tools may overestimate the length of the WCEP (e.g. overestimation of number of loop iterations like in the *fft* application, having nested non-rectangular loops). All in all, except for *fft* the number of page-ins is close to the one of a dynamic paging system, which shows predictability does not come at the price of performance loss.

Predictable paging vs analysis of LRU replacement. We have introduced our predictable paging technique because current page replacement policies used in real-time

[3]Nachos web site, http://www.cs.washington.edu/homes/tom/nachos/

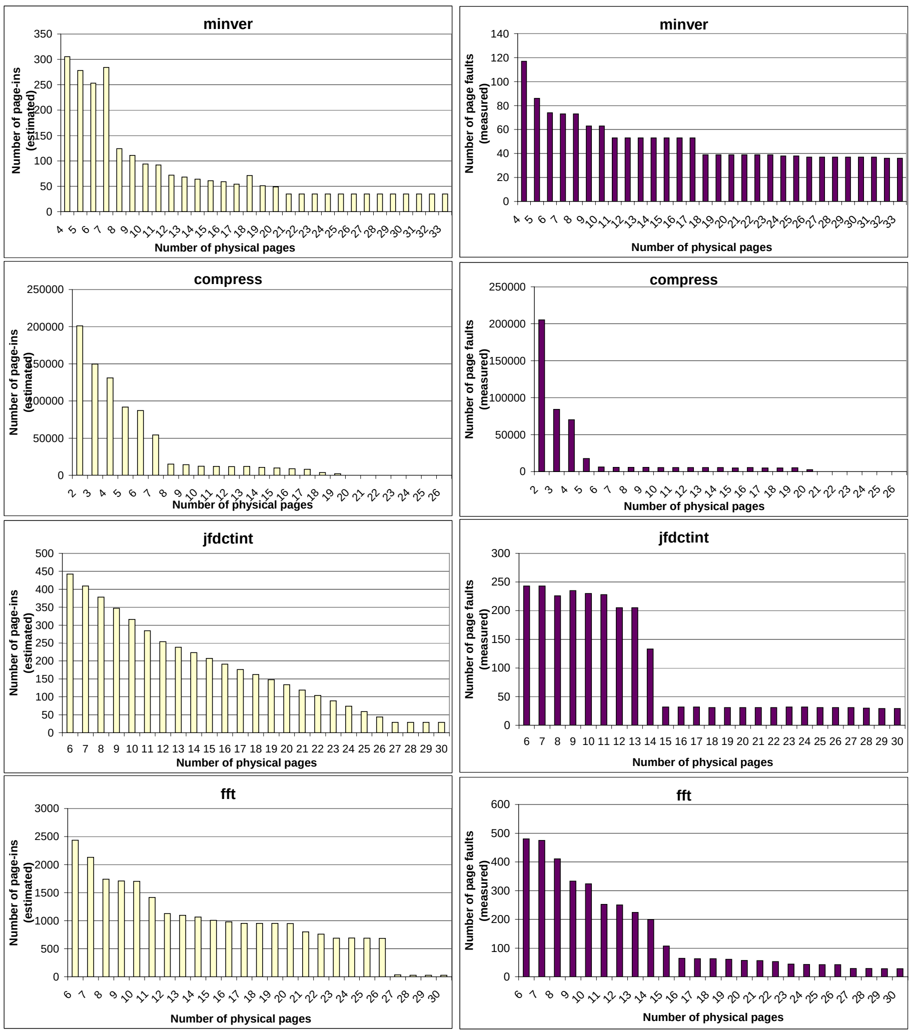


Figure 4. Impact of number of physical pages on number of page-ins

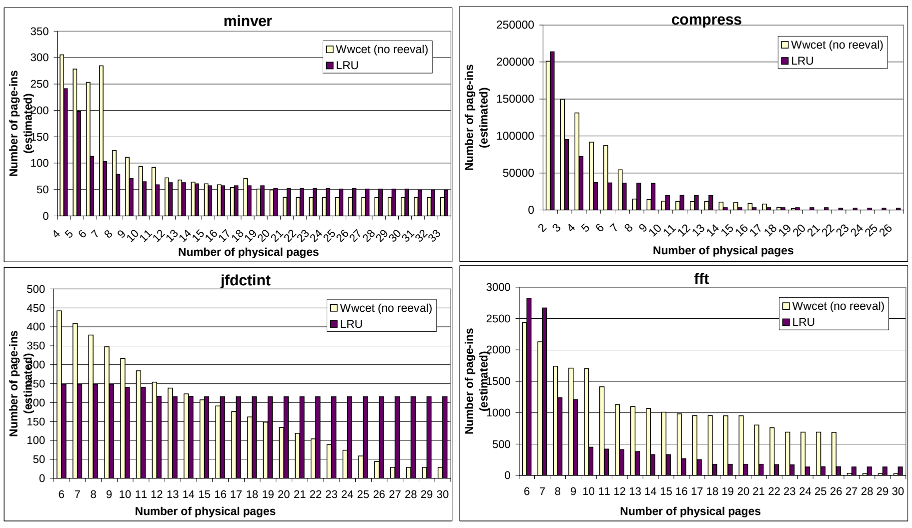


Figure 5. Predictable paging vs analysis of LRU replacement

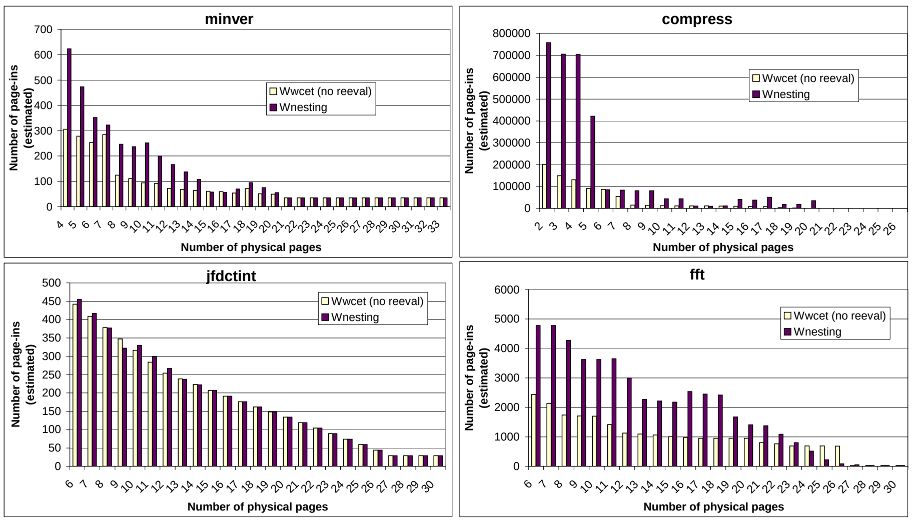


Figure 6. Impact of weight function

operating systems are not predictable enough. As a consequence, it is not possible to compare WCETs of programs with state-of-the-art page replacement policies with our predictable paging method. Thus, we have compared our proposal with a static analysis LRU page replacement, a highly predictable replacement policy. Our analysis of LRU page replacement uses Heptane static instruction cache analysis method for fully associative caches. The cache analysis method of Heptane (see [4] for details) is based on F. Mueller's *static cache simulation* [9]. Results are expressed in Fig. 5 in terms of number of page-ins along the worst-case execution path.

When the memory is not too scarce, our predictable paging yields to lower WCET estimate than LRU. We have observed on small examples two situations explaining the pessimism of the analysis of LRU page replacement:

- Circular access to a set of pages of cardinal P within a loop, on a system with less than P physical pages. In that situation, LRU replacement behaves poorly because every evicted page will be reused shortly after in the loop. This deficiency, detailed in [7] is a deficiency of LRU replacement itself and not the static analysis of LRU replacement.
- Classifications as *misses* of references to pages accessed both in the body of a loop and the loop exit. In that situations, the static analysis of LRU page replacement considers that the loop may iterate zero times and thus the page may have to be loaded from disk. This pessimism may become important in the case of nested loops. Here, the problem is with the static analysis of LRU replacement and not with LRU replacement itself. This problem could be fixed if a lower bound of the number of iterations of loops was provided to the WCET estimation tool.

When the number of physical pages is extremely low, in most cases the analysis of LRU yields to tighter WCET estimates than our scheme. A closer analysis of the sources of pessimism of our proposal in that situations is still needed and is left for future work.

Impact of weight function. The two weight functions W_{wcet} and $W_{nesting}$ presented in paragraph 3.4 have been implemented and tested. Figure 6 gives the number of page-ins along the worst-case execution path for these two weight functions.

Except on very rare cases, the number of estimated page faults is much lower when using the W_{wcet} heuristic than when using $W_{nesting}$. Using frequency information is thus valuable for obtaining as tight WCET estimates.

The numbers given in the appendix show that re-evaluating the worst-case execution path during the graph coloring has no impact. This is not surprising for the tasks with little data-dependencies (matmult, jfdctint) but needs further investigations for the others.

5 Implementation issues and future work

Some hardware and/or operating system support is required to fully implement our proposal. The first requirement is to have support for executing code (here, page-ins and page-outs) at specific code locations. This could be done by using hardware debug registers or operating system support for debug, if any. Another requirement is to have support for changing translation information. This is expected to be straightforward for operating systems with page locking facilities like in RT-Mach, real-time extensions of POSIX, or a library to control address translation like in VxWorks. Further work is required to evaluate the implementation cost of our proposal, in particular in presence of shared libraries/code/memory segments or multiple threads sharing the same address space.

The algorithm for off-line selection of page-in and page-out points is independent of the type of pages referenced (code, data), as far as referenced pages are known at compile time. The difficulty of applying our scheme to data comes from the identification of data pages referenced by every instruction, in case data addresses are computed dynamically (accesses to arrays, stack allocated data, dynamically allocated data). The identification of referenced pages need not be exact, it is sufficient that all pages that *may* be referenced are known. Our ongoing work evaluates the practical feasibility of identifying possibly referenced data pages, and to quantify the negative impact of an imprecise knowledge of data references.

Furthermore, to be used in hard real-time systems, disks (or any other classes of secondary storage) with predictable access times are required. This is a direction for future research.

Finally, this paper has focused on a single task, and has left open the choice of the number of pages assigned to every task. In [15] an algorithm for optimally partitioning two-level memory and minimize the task utilization is described. Such an algorithm can be used to select the number of physical pages assigned to each task such as to minimize utilization. A research direction would be to improve that algorithm to optimize schedulability rather than utilization.

Acknowledgments. The authors would like to thank E. Abgrall who implemented the algorithm together with D. Hardy, as well as J. F. Deverge and K. Brifault for their comments on earlier drafts of this paper.

References

[1] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[2] M. D. Bennett and N. C. Audsley. Predictable and efficient virtual addressing for safety-critical real-time systems. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 183–190, Delft, The Netherlands, June 2001.

[3] G. J. Chaitin. Register allocation and spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–105, 1982.

[4] A. Colin and I. Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, The Netherlands, June 2001.

[5] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for real-life processor. In *Proceedings of the first international workshop on embedded software (EMSOFT 2001)*, volume 2211 of *Lecture Notes in Computer Sciences*, pages 469–485, Tahoe City, CA, USA, Oct. 2001.

[6] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7), July 2003.

[7] B. Juurlink. Approximating the optimal replacement algorithm. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 313–319, 2004.

[8] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[9] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, May 2000.

[10] D. Niehaus. *Program Representation and Execution in Real-Time Multiprocessor Systems*. PhD thesis, University of Massachusetts, Feb. 1994.

[11] D. Niehaus, E. Nahum, J. Stankovic, and K. Ramamritham. Architecture and OS support for predictable real-time systems. Technical report, University of Massachusetts, Mar. 1992.

[12] R. J. Pankhurst. Program overlay techniques. *Communications of the ACM*, 11(2):119–125, Feb. 1968.

[13] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, May 2000. Guest Editorial.

[14] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universitat, Institut fur Technische Informatik, Wien, Apr. 1995.

[15] J. E. Sasinowski and J. K. Strosnider. A dynamic programming algorithm for cache/memory partitioning for real-time systems. *IEEE Transactions on Computers*, 42(8):997–1001, Aug. 1993.

[16] H. Tokuda, T. Nakajima, and T. Rao. Real-time mach: Towards predictable real-time systems. In *Proceedings of the USENIX Mach Workshop*, Oct. 1990.

[17] WindRiver Systems. *VxWorks programmer's guide*, Nov. 1998.

A Appendix: Raw numbers

In the following tables, the first column indicates the number of pages used. The next three columns then give the estimated number of page-ins along the worst-case execution path using respectively: the W_{wcet} weight function with and without re-evaluation of the WCEP (R and no-R) and the $W_{nesting}$ weight function. Column LRU gives the estimeted number of page-ins along the worst-case execution path when assuming a LRU page replacement policy. Finally, the last column gives the measured number of page faults for a LRU page replacement policy when following one execution path.

minver	Wwcet	Wwcet	Wnesting	LRU	Measured
page nb	(R)	(no-R)			
4	305	305	623	241	117
5	278	278	473	198	86
6	253	253	352	113	74
7	284	284	322	103	73
8	124	124	246	79	73
9	111	111	236	71	63
10	94	94	252	65	63
11	92	92	199	59	53
12	72	72	166	63	53
13	68	68	137	63	53
14	64	64	108	61	53
15	61	61	58	57	53
16	59	59	56	57	53
17	54	54	70	57	53
18	71	71	95	57	39
19	51	51	75	57	39
20	49	49	55	52	39
21	35	35	35	52	39
22	35	35	35	52	39
23	35	35	35	52	39
24	35	35	35	52	38
25	35	35	35	51	38
26	35	35	35	52	37
27	35	35	35	51	37
28	35	35	35	51	37
29	35	35	35	51	37
30	35	35	35	51	37
31	35	35	35	49	37
32	35	35	35	49	36
33	35	35	35	49	36

compress	Wwcet	Wwcet	Wnesting	LRU	Measured
page nb	(R)	(no-R)			
2	201142	201142	758112	213793	205333
3	149706	149706	704797	95433	84179
4	131146	131146	704262	72265	70135
5	91723	91723	421127	37202	17685
6	87116	87116	85512	36626	6180
7	54349	54349	83721	36370	5638
8	14926	14926	80906	36370	5638
9	14143	14143	80394	36115	5624
10	12063	12063	44299	19731	5382
11	11807	11807	44300	19731	5382
12	11552	11552	10765	19475	5382
13	11809	11809	9742	19475	5382
14	10529	10529	11023	3096	5381
15	9762	9762	41489	3096	5006
16	8739	8739	37650	3096	5382
17	7956	7956	51219	3096	5043
18	3606	3606	18452	3096	4912
19	2071	2071	18453	3096	5100
20	25	25	34839	3096	2550
21	24	24	24	3096	25
22	24	24	24	2584	25
23	24	24	24	2584	25
24	24	24	24	2584	25
25	24	24	24	2584	24
26	24	24	24	2584	24

qurt	Wwcet	Wwcet	Wnesting	LRU	Measured
page nb	(R)	(no-R)			
2	388	388	385	387	116
3	271	271	386	330	103
4	158	158	327	270	98
5	99	99	156	270	45
6	97	97	148	159	44
7	89	89	148	156	42
8	89	89	146	102	42
9	87	87	144	100	42
10	84	84	141	43	41
11	25	25	82	42	40
12	23	23	77	40	29
13	18	18	19	39	20
14	14	14	14	40	15
15	14	14	14	39	14
16	14	14	14	37	14

matmult	Wwcet	Wwcet	Wnesting	LRU	Measured
page nb	(R)	(no-R)			
3	7657	7657	257655	132604	12504
4	5158	5158	252506	5060	2504
5	7	7	7	2512	7
6	7	7	7	2512	7
7	7	7	7	13	7
8	7	7	7	13	7

crc	Wwcet	Wwcet	Wnesting	LRU	Measured
page nb	(R)	(no-R)			
2	1376	1376	1376	1165	1669
3	782	782	862	863	1288
4	12	12	607	525	1286
5	11	11	523	524	12
6	10	10	10	525	12
7	10	10	10	523	12
8	10	10	10	523	12
9	10	10	10	523	12
10	10	10	10	12	10
11	10	10	10	12	10

jfdctint	Wwcet	Wwcet	Wnesting	LRU	Measured
page nb	(R)	(no-R)			
6	442	442	455	248	243
7	409	409	417	248	243
8	378	378	377	248	226
9	347	347	322	248	235
10	316	316	330	240	230
11	284	284	299	240	228
12	254	254	267	217	205
13	238	238	237	215	205
14	223	223	222	216	133
15	207	207	207	215	32
16	191	191	191	215	32
17	176	176	176	215	32
18	162	162	162	215	31
19	148	148	148	215	31
20	134	134	134	215	31
21	119	119	119	215	31
22	104	104	104	215	31
23	89	89	89	215	32
24	74	74	74	215	32
25	59	59	59	215	31
26	44	44	44	215	31
27	29	29	29	215	31
28	29	29	29	215	30
29	29	29	29	215	29
30	29	29	29	215	29

fft	Wwcet	Wwcet	Wnesting	LRU	Measured
page nb	(R)	(no-R)			
6	2436	2436	4783	2825	480
7	2131	2131	4778	2669	475
8	1740	1740	4276	1237	411
9	1709	1709	3627	1207	333
10	1701	1701	3626	449	324
11	1414	1414	3655	419	252
12	1127	1127	2992	411	250
13	1096	1096	2271	379	224
14	1065	1065	2220	331	199
15	1007	1007	2177	331	107
16	980	980	2537	267	64
17	953	953	2450	251	63
18	952	952	2421	177	63
19	951	951	1675	177	61
20	950	950	1406	177	57
21	802	802	1373	179	56
22	761	761	1086	171	53
23	690	690	799	170	44
24	691	691	512	135	43
25	689	689	225	137	42
26	686	686	82	137	42
27	35	35	55	136	29
28	28	28	28	135	29
29	28	28	28	135	28
30	28	28	28	136	28