Predictable code and data paging for real time systems *

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

Abstract

There is a need for using virtual memory in real-time applications: using virtual addressing provides isolation between concurrent processes; in addition, paging allows the execution of applications whose size is larger than main memory capacity, which is useful in embedded systems where main memory is expensive and thus scarce. However, virtual memory is generally avoided when developing real-time and embedded applications due to predictability issues. In this paper we propose a predictable paging system in which the page loading and page eviction points are selected at compile-time. The contents of main memory is selected using an Integer Linear Programming (ILP) formulation. Our approach is applied to code, static data and stack regions of individual tasks. We show that the time required for selecting memory contents is reasonable for all applications including the largest ones, demonstrating the scalability of our approach. Experimental results compare our approach with a previous one, based on graph coloring. It shows that quality of page allocation is generally improved, with an average improvement of 30% over the previous approach. Another comparison with a state-of-the-art demand-paging system shows that predictability does not come at the price of performance loss.

1. Introduction

The use of virtual memory and demand paging when developing real-time and embedded applications is generally avoided because of the difficulties of predicting the dynamic behavior of paging activity. Instead, applications are loaded in main memory at program start, and virtual addressing is not used. Nevertheless, as many embedded systems are getting increasingly large and complex, there is now a need to overcome this strict static memory management.

Hardware support for virtual memory is now present in many embedded commercial processors. Virtual memory hardware provides address mapping and memory protection for processes. A process's virtual address space is divided into fixed-size pages, which are mapped at run-time to physical pages. Page mapping information is stored in main memory in *page tables*, scanned by the Memory Management Unit (MMU)[1]. In systems with standard demand paging, when a program attempts to reference an unmapped page, a *page fault* occurs, and the operating system's page-fault handler loads the page from disk on demand (*page-in*). Symmetrically, when there is no free physical page anymore, a software-implemented *replacement policy* selects one virtual page to evict from main memory (*page-out*). Virtual memory hardware allows to implement isolation between processes. Moreover, it permits to execute tasks whose address space is larger than the capacity of main memory, which is particularly interesting in embedded systems with stringent cost constraints.

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 [25]. Virtual memory raises predictability issues at two levels:

- Level of *address translation*: the duration of the address translation is hard-to-predict due to the presence of a TLB, because the TLB replacement policy may not always be well documented or not amenable to static analysis [17].
- Level of *paging activity*: knowing whether or not a reference to a virtual page will result in a page fault is hard to predict: *(i)* page replacement policies are often more complex than cache replacement policies (software-implemented in the operating system), and thus are far from the highly predictable strict Least Recently Used (LRU) [17], *(ii)* main 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.

So far, attempts to provide real-time address spaces have focused on the predictability of virtual to physical address translation [12, 3]. In this paper, we propose a predictable compiler-directed memory management technique, in which points where code and data pages are loaded from secondary storage (disk/flash) and paged out to secondary storage are selected at compile-time transparently. The selection of page

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

[1] Most MMUs include a Translation Lookaside Buffer (TLB), which is a small fully-associative cache used to speed up lookup of the page table.

load/unload points is achieved using a 0/1 Integer Linear Programming (ILP) formulation. Experimental results on a real size application are provided to compare our ILP formulation with a previous approach for predictable paging, which is named hereafter *graph coloring approach*, that we have previously described in [16]. The comparison shows that the ILP formulation yields a better quality of page allocation than our previous work. Moreover, although by similarity to register allocation the page allocation problem can be shown to be NP-complete [7], the ILP problem is demonstrated to be solved in reasonable time even for the biggest tasks. Finally, a comparison of our approach with a state-of-the-art demand-paging system shows that the predictability of our scheme does not come at the price of a performance loss. To the best of our knowledge, this work is the first work allowing a *transparent* (compiler-controlled) and *predictable* use of paging hardware for real-time applications, applicable both to code and data (stack and static data).

The selection of page load/unload points in our approach requires that I/O operations to disk/flash have a bounded (but not necessarily tight) latency. The design of I/O systems with predictable and low I/O latency is considered outside the scope of this paper. The interested reader is referred to [1] for an example of real-time disk management for soft real-time systems.

In order to be independent from the task scheduling policy, our compiler-directed memory management scheme operates on individual tasks. It is intended to be used together with a memory partitioning scheme such as [18] to partition the main memory among the application tasks.

The rest of the paper is organized as follows. Related work is surveyed in Section 2. Section 3 first presents how addresses, required to select the contents of main memory at compile-time, are statically computed; Section 3 then details our ILP formulation of the page allocation problem. Experimental results are given in Section 4. Finally, Section 5 summaries the paper contributions and gives directions for future work.

2. Related work

The definition of efficient page replacement algorithms has received considerable attention in the operating systems community in the seventies. Since optimal page replacement, as defined in [2], cannot be implemented in practice because it requires an exact knowledge of future memory accesses, existing page replacement strategies exploit the knowledge of past references to guess future ones. In contrast to existing page replacement policies, page replacement in our approach is decided at compile-time for the sake of predictability.

So far, demand paging is avoided in real-time operating systems. For processors with a MMU, some systems like Spring [11] use it for protection between processes only. In Spring, all the pages of a program are loaded at process start such that page 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 [21] and real-time extensions of POSIX provide a system call to wire pages in memory for real-time tasks. [14] proposes a page lock/release mechanism for out-of-core embedded applications. The approach automatically inserts paging hints used by the replacement algorithm to keep/discard pages. Since a part of the paging activity is still under operating system support through a dynamic method, the approach is not directly usable in real-time systems. A very predictable approach called *overlaying* [13] was used before the hardware support for virtual memory became common. Unfortunately, overlaying techniques, while highly predictable, were in most systems non automatic, requiring manual work of the programmer to define the overlays. [6] proposes a manual overlaying approach suited to real-time systems, applied to code and static data. In our case, the selection is automatic and stack-allocated variables are supported.

One may view the predictability issues caused by paging as identical to those raised by caches. Many methods have been designed to estimate WCETs on architectures with instruction and/or data caches [10, 20], for different cache structures and replacement policies. The tightest predictions are obtained for LRU replacement. In contrast, pseudo round-robin and random replacement yield looser timing estimates [17]. 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. 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 and use a compiler-directed approach instead.

Compiler-directed memory management is not new. At the lower levels of the memory hierarchy, *cache locking* schemes [22, 15] select cache contents at compile time. The specifics of cache locking as compared our approach lies in the structure of cache memories, imposing constraints on the locations of memory blocks into the cache (in direct-mapped and set-associative caches). [5, 23] proposed ILP formulations for data allocation in scratchpad memory to minimize the WCET and respectively to reduce the energy consumption.

At the higher level of the memory hierarchy, a compiler-directed control of the paging activity is proposed in [9]. In contrast to our work, [9] focuses on numerical programs (only considers accesses to arrays inside loops) and does not target real-time applications. In [16] we have proposed a compiler-directed page allocation scheme for code regions only. In that previous work, the problem under study was formulated as a graph coloring problem. Compared to our previous work, this paper improves the quality of page allocation and is now able to control paging of code, stack and static data regions.

3. Predictable paging using an ILP formulation

In this section, we describe our ILP formulation which determines at compile-time, code/data pages to be loaded and evicted at run-time. First, we present the framework we have developed to analyze the addresses of data, which are necessary to deal with stack pages and static data pages. Then, we show our program representation and detail our ILP formulation. Finally, implementation considerations are discussed.

3.1. Static address analysis

Static analysis of *code* addresses is achieved in a straightforward manner through an analysis of the binary file. References to *data* are more difficult to extract statically. Static analysis of data references has been the subject of many previous works. For instance, in [5], the compiler is modified to extract the information created during the compilation process. [24] uses data-flow analysis at the assembly level. Like [24] our address analyzer operates on disassembled binaries, thus avoiding any modification of the compiler. Since we are interested in accesses to pages, we do not need an analysis granularity as fine as in [24]. For instance, we do not distinguish loop inductions variables. We do not try to distinguish individual accesses to array elements or local variables within a stack frame; the granularity of our analyzer is the entire array or entire stack frame, thus yielding low address analysis time.

The address analysis is *safe* in the sense that it catches all accesses to static and local data. The analysis is context-insensitive in the sense that it does not distinguish differences between two executions contexts (e.g. calling contexts of functions). This assumption, made for the sake of implementation simplicity, turned out to be realistic for the class of applications we have analyzed (see Section 4), where we find just a few different contexts of execution. Our current implementation does not include pointer analysis, that is left for future work.

We have applied our address analyzer on MIPS assembly code. Nevertheless, this approach can be easily ported to other architectures.

3.1.1 Stack analysis

The stack analysis we have designed assumes an acyclic call graph, which is common in real-time systems, where recursion raises predictability issues. The analysis relies on the knowledge of the stack base address and the size of all function stack frames, obtained by scanning the first instruction of every function which is responsible for allocating the stack frame.

Knowing stack base and size of individual stack frames (Figure 1.a), the analysis consists in propagating the address of the stack frame for every callee of every function along the acyclic call graph.

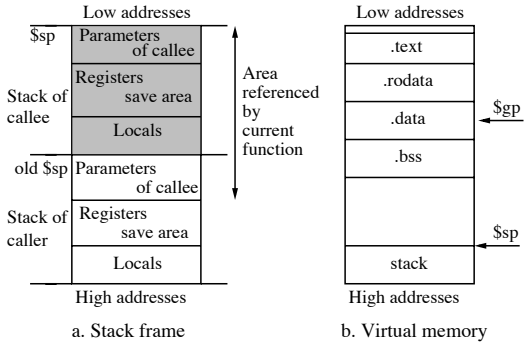


Figure 1. MIPS memory organization

3.1.2 Static data analysis

We analyze the contents of each register before and after every instruction by an intra basic block data-flow analysis. Since we do not currently support pointers, it is sufficient to analyze register contents to compute the load and store addresses and thus no analysis of memory contents is required. We determine accesses of load and store instructions with the following data-flow equations, where $RC_{in}(inst)$ (respectively $RC_{out}(inst)$) is the contents of each register before (respectively after) the execution of instruction $inst$:

$$RC_{in}(inst) = \begin{cases} RC_{out}(inst\prime) \text{ with } inst\prime \text{ the} \\ \text{preceding instruction of the} \\ \text{basic block if any} \\ \\ \{(r, \perp) \mid r \in (Reg \setminus \{\$gp, \$sp\})\} \\ \quad \cup \{(\$gp, gp), (\$sp, sp)\} otherwise \end{cases}$$

$$RC_{out}(inst) = (RC_{in}(inst) \setminus kill(inst)) \cup gen(inst)$$

In the equations, Reg is the set of MIPS registers, $gen(inst)$ is the set of pairs $(\$r, v)$ generated by instruction $inst$, where the pair $(\$r, v)$ represents a value v contained in register $\$r$. $kill(inst)$ is the set of pairs $(\$r, v)$ erased by instruction $inst$. $(\$gp, gp)$ and $(\$sp, sp)$ are two particular pairs which represent the fact that the content of $\$gp$ and $\$sp$ are considered like constants during the analysis. Register $\$gp$ is a register containing a global pointer to the data section (Figure 1.b) determined at compile-time and $\$sp$ is the stack pointer computed by the stack analysis. Finally, $\perp$ represents an invalid register content. It is used as an initial register value for the first instruction of every basic block. It allows us to check that pointers are not used within basic blocks.

$kill$ and gen functions are defined for all instructions. In the following for the sake of conciseness we concentrate on a small set of representative instructions. As an illustration, the store instruction $store\ \$r, @mem$ stores the contents of register $\$r$ at the memory address $@mem$. This instruction does not modify any register content. Thus, $gen(store\ \$r, @mem) = \emptyset$ and $kill(store\ \$r, @mem) = \emptyset$.

The add instruction $add\ \$r1, \$r2, v$ adds the contents of register $\$r2$ with value v (constant or register contents) and stores

the result in register $\$r1$. $kill$ and gen for this instruction are defined as follows:

$$gen(add\ \$r1, \$r2, v) = \begin{cases} \{(\$r1, v)\} & \text{if } \$r2 = \top \text{ and } v \text{ is a constant or a value in a register } v \\ \{(\$r1, \bot)\} & \text{if } \$r2 = \bot \text{ or } v \text{ is a register containing } \bot \\ \{(\$r1, [\$r2] + v)\} & \text{otherwise} \end{cases}$$

$$kill(add\ \$r1, \$r2, v) = \{(\$r1, x) \mid x \in Val\}$$

where Val is the set of all possible values including $\bot$ and $\top$. $\top$ represents a correct but unknown value (any possible value but $\bot$). We use $\top$ to specify the contents of a register after a load from memory because memory contents are not analyzed (e.g. induction variable in stack frame)

With this registers contents analysis, we are able to determine for each load and store an interval of addresses. For static data, we use the symbol table to determine this interval and for local data we use the stack frame as interval. Intervals of addresses are transformed into sets of potentially referenced virtual pages, used by the ILP formulation (see § 3.3).

3.2. Program representation

We represent a program with an inter-procedural Control Flow Graph (CFG) (central part of Figure 2) constructed at compile-time from disassembled code. There is one node per basic block and one edge per possible sequence between two basic blocks (caused by conditional and unconditional branches, function calls and function returns). We assume that every basic block uses a number of virtual pages lower than or equal to the number of physical pages allocated to the program. But obviously, the number of virtual pages used by the entire program can be larger than the number of physical pages.

Let us note $G = (V, E)$ the program CFG, with V the set of nodes representing the basic blocks and E the set of transitions between these nodes. We add two virtual nodes called $first$ and $last$ added respectively before the entry point and after the program return point(s).

3.3. ILP formulation

Our ILP formulation for the page allocation problem consists of determining at compile-time the points in the CFG where virtual pages will be paged-in and paged-out from/to secondary storage (flash/disk). This allocation must ensure that all referenced virtual pages of every basic block will always be present in main memory when used. In the context of real-time systems, we want to minimize the WCET by reducing the number of page-ins/page-outs along the Worst-Case Execution Path (WCEP). This is done by keeping in memory virtual pages which are not strictly necessary for a basic block in case there are some free physical pages available and the page is reused. The ILP formulation is based on initial knowledge of execution frequencies of edges belonging to the WCEP.

Distinctions are made between initialized and non-initialized pages (stack and data) in order to avoid unnecessary page loading for non-initialized pages. Symmetrically, detection of last references to a virtual page allows to deallocate the page instead of writing it back to secondary storage. Note that this second optimization cannot be implemented in demand-paging systems due to lack of knowledge of future references to pages.

In the following, symbols in capitals denote the inputs of the ILP problem, while lower-case symbols denote the problem variables. The inputs of the ILP formulation are given in Table 1. Values LOADCOST, ALLOCCOST, WRITECOST and DEALLOCCOST include the time required to change the page mappings.

The binary variables of the ILP formulation are given in Table 2. The central variable of the ILP formulation and main output of the ILP problem is variable vp_i^p, determining if virtual page p has to be kept in main memory while executing basic block i.

Variable $load_{(i,j)}^p$ indicates that page p has to be loaded in main memory along edge (i, j), either from secondary storage if the page has an initial value in secondary storage, or simply allocated otherwise. Companion variables $loadfd_{(i,j)}^p$ and $alloc_{(i,j)}^p$ represent these two complementary subcases. Variable def_i^p, evaluated during ILP solving, serves at distinguishing between these two subcases.

Management of page unloading is achieved using symmetrical variables. $unload_{(i,j)}^p$ specifies that page p has to be evicted. Companion variables $unloadw_{(i,j)}^p$ and $dealloc_{(i,j)}^p$ specify if the page has to be written back to secondary storage or simply deallocated. Variables $write_i^p$ and $dead_i^p$ serve at identifying this second case.

Objective function. The objective of the ILP formulation is to reduce the impact of page-ins and page-outs by keeping in main memory the virtual pages which are most frequently used along the WCEP. The objective function aims at minimizing the sum of contributions to the WCET of all page-ins and page-outs along the WCEP, with the distinction between allocation and load from secondary storage, and with the distinction between write-back to secondary storage and deallocation. It can be expressed as follows:

$$\sum_{(i,j)\in E} \sum_{p\in \mathrm{VP}} \mathrm{F}_{(i,j)} * (\quad \mathrm{LOADCOST} * loadfd_{(i,j)}^p + \mathrm{ALLOCCOST} * alloc_{(i,j)}^p + \mathrm{WRITECOST} * unloadw_{(i,j)}^p + \mathrm{DEALLOCCOST} * dealloc_{(i,j)}^p)$$

The ILP formulation needs some extra constraints to avoid inconsistencies between variables. Two classes of constraints,

Name	Definition
N	Number of physical pages
VP	Set of code and data virtual pages of the task
DATAVP	Set of virtual data pages of the task
NOINITVP	Set of virtual pages containing stack pages and non-initialized data pages of the task (NOINITVP $\subseteq$ DATAVP $\subset$ VP)
USEDVP$_i$	Set of virtual pages referenced by basic block i. In other words, pages that must be present in main memory during the execution of basic block i
F$_{(i,j)}$	Execution frequency along the WCEP of edge (i,j). Details on the computation of F$_{(i,j)}$ are given in Section 4.1.
LOADCOST	Worst-case time required to load one virtual page from secondary storage into main memory
ALLOCCOST	Worst-case time required to allocate one virtual page into main memory
WRITECOST	Worst-case time required to write back one virtual page from main memory
DEALLOCCOST	Worst-case time required to deallocate one virtual page from main memory
PRED$_i$	Set of predecessors of basic block i in CFG
SUCC$_i$	Set of successors of basic block i in CFG
MODVP$_i$	Set of virtual pages modified by basic block i (MODVP$_i$ $\subset$ USEDVP$_i$)

Table 1. Inputs of the ILP formulation

Name	Definition
vp_i^p	Equal to 1 if page p is present in main memory while executing basic block i
$load_{(i,j)}^p$	Equal to 1 if page p is paged-in along edge (i,j)
$loadfd_{(i,j)}^p$	Equal to 1 if page p is loaded from secondary storage along the edge (i,j)
$alloc_{(i,j)}^p$	Equal to 1 if page p is allocated in main memory along edge (i,j)
def_i^p	Equal to 1 if page p is not referenced by any path from the entry point to basic block i (i included)
$unload_{(i,j)}^p$	Equal to 1 if page p is evicted from main memory along edge (i,j)
$unloadw_{(i,j)}^p$	Equal to 1 if page p is written back to secondary storage along edge (i,j)
$dealloc_{(i,j)}^p$	Equal to 1 if page p is deallocated from main memory along edge (i,j)
$write_i^p$	Equal to 1 if page p is modified along at least a path leading to basic block i since its allocation
$dead_i^p$	Equal to 1 if page p is referenced by any path starting at i (i included)

Table 2. Variables of the ILP formulation

given below, express *(i)* the main memory limitation, and *(ii)* constraints on page usage.

Main memory limitation. Regarding main memory limitation, for each node of the CFG, the number of virtual pages present in main memory must be lower than or equal to the number of physical pages N:

$$\forall i \in V,\ \sum_{p \in VP} vp_i^p \leq \text{N} \tag{1}$$

Page loading constraints. A page-in of a virtual page p along edge (i,j) occurs when p is not present in main memory for node i (i.e. $vp_i^p = 0$) and p is present for node j (i.e. $vp_j^p = 1$). This boolean condition can be expressed as follows:

$$\begin{aligned} &\forall (i,j) \in E, \forall p \in VP \\ load_{(i,j)}^p &\leq 1 - vp_i^p \\ load_{(i,j)}^p &\leq vp_j^p \\ load_{(i,j)}^p &\geq vp_j^p - vp_i^p \end{aligned} \tag{2}$$

To ensure the presence in memory of the virtual pages needed by the execution of a basic block i, vp_i^p is set to 1 for each basic block i which references p ($p \in$ USEDVP$_i$). vp_{first}^p and vp_{last}^p are set to 0.

$$\begin{aligned} &\forall p \in \text{USEDVP}_i,\ \forall i \in (V \setminus \{first, last\}),\ vp_i^p = 1 \\ &\forall p \in \text{VP},\ vp_{first}^p = 0,\ vp_{last}^p = 0 \end{aligned} \tag{3}$$

The contents of vp_i^p when $p \notin$ USEDVP$_i$ is the result of the resolution of the ILP problem. Some pages may be kept in memory, although not used, to reduce the paging activity (e.g. when a page is used twice in a loop, see example in Figure 2).

The distinction between a page allocation and a load from secondary storage is made by the def_i^p variable. An allocation occurs when a non-initialized virtual page p was not referenced before. This can be expressed as follows:

$$\begin{aligned} &\forall (i,j) \in E, \forall p \in \text{VP}, \\ alloc_{(i,j)}^p &\geq load_{(i,j)}^p + def_i^p - 1 \\ alloc_{(i,j)}^p &\leq load_{(i,j)}^p \\ alloc_{(i,j)}^p &\leq def_i^p \end{aligned} \tag{4}$$

A load from secondary storage occurs when an allocation is not possible:

$$\forall (i,j) \in E,\ \forall p \in \text{VP},$$
$$loadfd^p_{(i,j)} = load^p_{(i,j)} - alloc^p_{(i,j)} \quad (5)$$

Let us now define the constraints on def^p_i. This variable is set to 1 for basic block $first$ only if the virtual page is a non-initialized page:

$$\forall p \in \text{NOINITVP},\ def^p_{first} = 1$$
$$\forall p \notin \text{NOINITVP},\ def^p_{first} = 0 \quad (6)$$

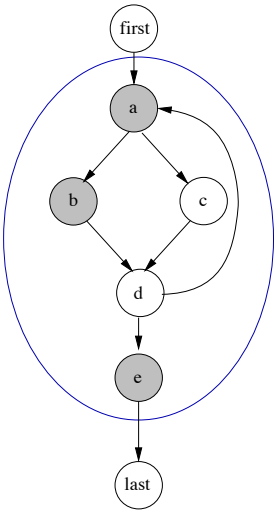
We want to propagate this information into the CFG if the non-initialized page is not referenced along all the paths:

$$\forall i \in V, \forall p \in \text{VP},$$

$$\begin{aligned} def^p_i &\geq (\sum_{k \in \text{PRED}_i} def^p_k) + (1 - vp^p_i) - \mid \text{PRED}_i \mid \\ def^p_i &\leq 1 - vp^p_i \\ def^p_i &\leq def^p_k, \forall k \in \text{PRED}_i \end{aligned} \quad (7)$$

□

Symmetrical constraints, omitted for space considerations, are generated for handling page unloading from main memory.



ILP initialization

$load^p_{(first,a)} = ?$
$load^p_{(a,b)} = ?$
$load^p_{(a,c)} = ?$
$load^p_{(b,d)} = ?$
$load^p_{(c,d)} = ?$
$load^p_{(d,e)} = ?$
$load^p_{(e,last)} = ?$
$vp^p_{first} = 0$
$vp^p_a = 1$
$vp^p_b = 1$
$vp^p_c = ?$
$vp^p_d = ?$
$vp^p_e = 1$
$vp^p_{last} = 0$

ILP results

$load^p_{(first,a)} = \mathbf{1}$
$load^p_{(a,b)} = \mathbf{0}$
$load^p_{(a,c)} = \mathbf{0}$
$load^p_{(b,d)} = \mathbf{0}$
$load^p_{(c,d)} = \mathbf{0}$
$load^p_{(d,e)} = \mathbf{0}$
$load^p_{(e,last)} = \mathbf{0}$
$vp^p_{first} = 0$
$vp^p_a = 1$
$vp^p_b = 1$
$vp^p_c = \mathbf{1}$
$vp^p_d = \mathbf{1}$
$vp^p_e = 1$
$vp^p_{last} = 0$

Figure 2. Example of ILP system and results

Figure 2 illustrates the result of the ILP process for a virtual page p referenced by basic blocks a, b and e (depicted in grey). The left part shows the initialization to 1 of a subset of variables vp^p_i, those representing pages strictly needed by the basic blocks. The right part gives the results of the ILP problem in the case where all basic block have enough free pages available to keep page p in memory during the loop.

3.4. Implementation issues

Calls to page load/unload routines have to be inserted into application code at compile-time. In order that the application memory map is not changed after the computation of paging points (which would invalidate the result of the selection), the most straigthforward solution is to reserve space for every possible page load/unload point at the assembly level. To limit the space requirements of such a simple approach, a possible improvement would be to put restrictions on the possible page load/unload points (e.g. function entry or loop header).

4. Experimental results

In this section, we evaluate the quality of our page allocation scheme by examining the impact of the paging activity on the worst-case execution path. The performance metric is the *number of costly paging actions* (page load from secondary storage and write backs to secondary storage, along the worst-case execution path), obtained using static analysis. We also execute the task in order to compare the number of paging actions with the one of a standard paging system using a pseudo LRU page replacement policy. The objective is then to analyze whether or not predictability results in a loss of performance. Finally, the time complexity of the selection of paging points is evaluated. We first describe the experimental conditions and then we give and analyze experimental results.

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] timing analyzer [4], more precisely its Implicit Path Enumeration Technique (IPET). IPET methods compute WCET estimates through the generation of an ILP problem aiming at identifying the longest path in a piece of code, see for instance [8]. IPET methods have the ability to produce execution frequencies along (one of) the longest execution paths. Computation of input parameter $\text{F}_{(i,j)}$ considers a page load/unload cost for a page p at the frontiers of the biggest connected usage areas of p.

Without loss of generality, the low-level analysis phase (instruction caches, branch prediction...) of Heptane is bypassed and a constant of 1 cycle execution time per instruction is considered. This choice is made because the latency of accesses to secondary storage is several orders of magnitude larger than the one of instruction execution. We thus observed that a precise modeling of the processor hardware was unnecessary because it has no impact on page allocation.

A page-in (respectively page-out) time of 1 million cycles is assumed for loading/writing back a page from/to secondary storage, including modifications of page mapping information.

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

Name	Description	code size (bytes)	.data size (bytes)	.bss size (bytes)
compress	Compression of a 128 x 128 pixel image using discrete cosine transform	3064	0	70656
crc	CRC (Cyclic Redundancy Check)	1236	274	768
jfdctint	Integer implementation of the forward DCT (Discrete Cosine Transform)	3204	0	256
qurt	The root computation of a quadratic equation	1760	4	56
fft	Fast Fourier Transform	3624	64	64
minver	Matrix inversion for 3x3 floating point matrices	4604	72	408
statemate	Automatically generated code by STARC (*STAtechart Real-time-Code generator*)	11704	0	290
task1	Confidential	20560	208	662
task2	Confidential	8092	380	3
task3	Confidential	24844	232	463

Table 3. Benchmark characteristics

Other numerical values have been tested with the same results in terms of number of paging operations. The allocation (respectively deallocation) of a page in main memory is assumed to take 100 cycles. We use pages of 512 bytes for the smaller benchmarks; page size is voluntarily small to stress the paging activity. On larger tasks (bottom three tasks of Table 3), we set the page size to 1024 bytes.

Tools. The ILP problem is solved by the commercial ILP solver CPLEX 10.0[3] on an Intel Pentium 4 3.6 GHz with 2 GB of RAM.

The measure of the paging activity is done on top of the Nachos educational operating system[4], extended with a state-of-the-art page replacement policy, namely the well-known pseudo-LRU clock algorithm [19]. Nachos runs on top of a simulated MIPS processor.

Benchmarks. The experiments were conducted on seven small benchmarks and three tasks from a larger real application (see Table 3). 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/). The real tasks are part of the case study provided by the automotive industrial partner of the Mascotte ANR project[5] to the project participants.

4.2. Results

Impact of approach for predictable paging on worst-case performance. The performance metric used is the number of costly paging operations along the WCEP. The lower is this number, the better is the page allocation quality. We compare the results of the ILP formulation detailed in this paper with an extension to data of our previous graph coloring approach. The main idea of the graph coloring approach (see [16] for details) is to color an interference graph, in which nodes represent areas of use of virtual pages (called *webs*) and in which colors represent physical pages. The greedy coloring process starts with the largest usage areas, which are split when the graph is not colorable. The splitting procedure selects the web to be split according to the web *weight* representing the impact of the web on the WCET. The weight of a web associated to a page p is defined as the sum of the frequencies along the WCEP of each basic block referencing p.

Name	Average improvement ratio	Computation Time Average in second
compress	57.36%	216.88
crc	30.68%	0.08
jfdctint	30.59%	0.17
qurt	3.13%	0.11
fft	54.75%	0.45
minver	52.21%	1.5
statemate	13.08%	238.99
task1	11.47%	88.18
task2	29.46%	5.33
task3	27.41%	246.18

Table 4. Benchmarks results

The results of the comparison are summarized in Table 4. Values in the table are average improvement ratios for different numbers of physical pages (average of $\frac{nb\_paging\_ops_{coloring} - nb\_paging\_ops_{ILP}}{nb\_paging\_ops_{coloring}}$) with the number of pages in the interval $[max(\mid \text{USEDVP}_i \mid), \mid \text{VP} \mid]$.

Figures 3 and 4 detail the results. The X axis gives the number of physical pages allocated to the task, while the Y axis reports the number of costly paging actions.

The ILP formulation yields a better allocation results than the graph coloring approach regardless of the number of physical pages available. An average reduction of 30% of the number of paging actions along the WCEP is observed.

Moreover, in almost all cases, the number of paging actions when using the ILP formulation decreases monotonically with the number of physical pages. In contrast, we observe a higher number of non-monotonic behaviors when using the coloring approach because coloring is greedy. The only observed non-

[3]ILOG CPLEX - High-performance software for mathematical programming and optimization: *http://www.ilog.com/products/cplex/*.

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

[5]http://www.projet-mascotte.org/

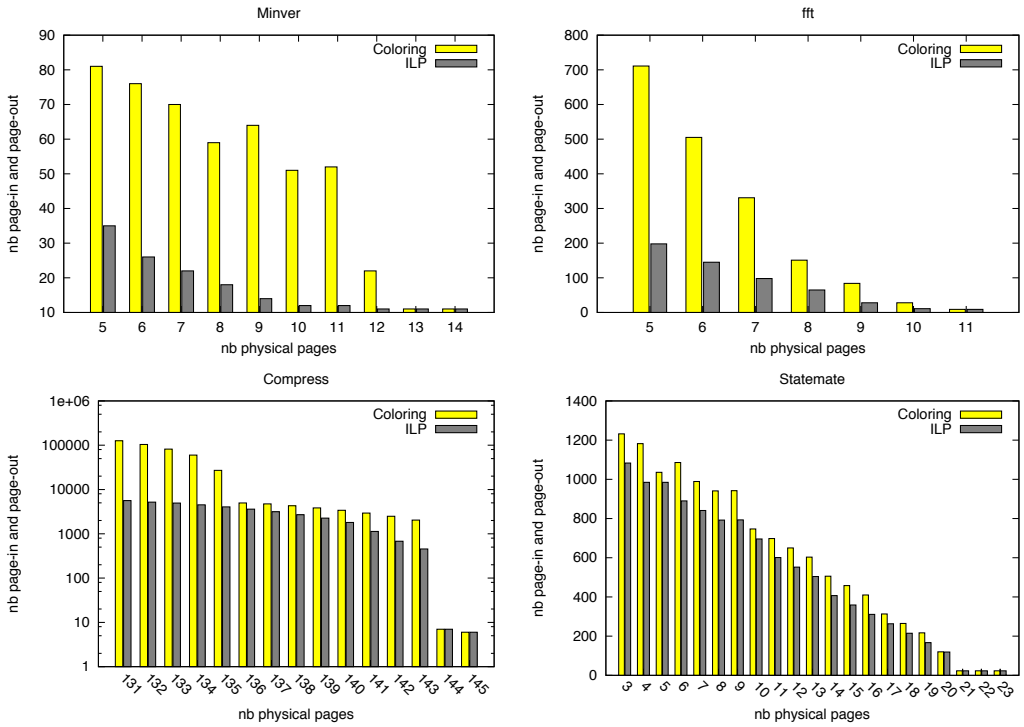


Figure 3. Comparison of predictable paging approaches for benchmarks

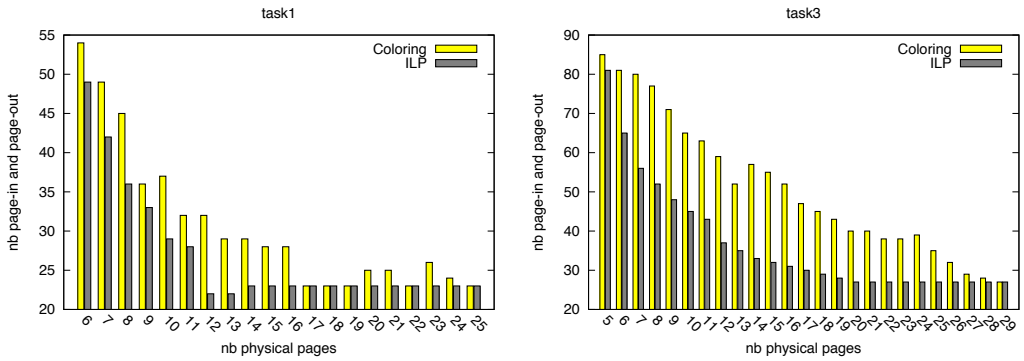


Figure 4. Comparison of predictable paging approaches for real application

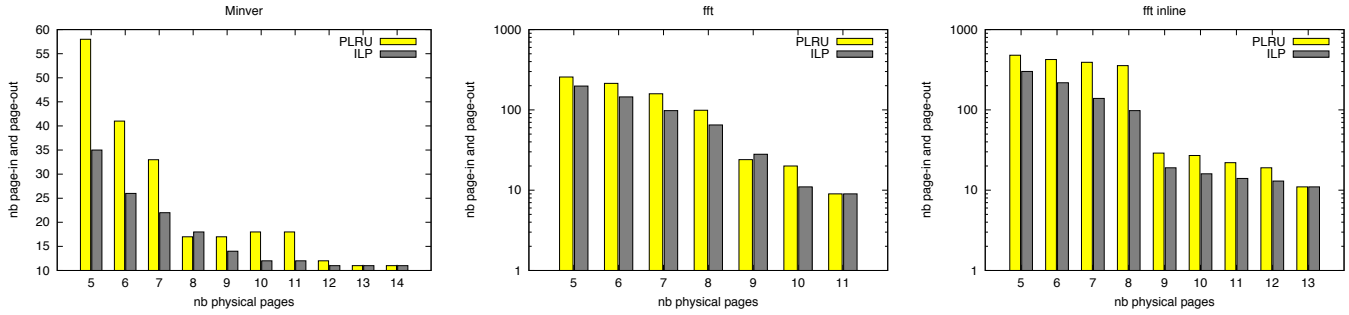


Figure 5. Comparison of predictable paging (ILP) with demand-paging

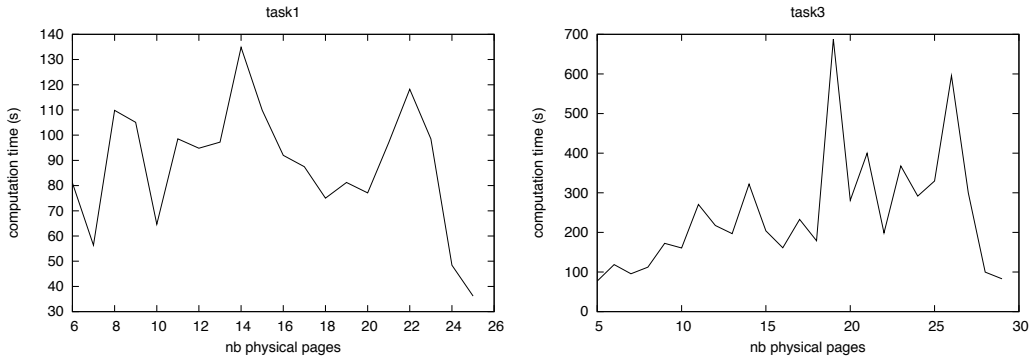


Figure 6. Computation time of the ILP problem for real application

monotonic behavior of the ILP formulation occurs for *task1* for a small number of memory sizes. This rare phenomenon is due to a change of the WCEP during the page allocation process, resulting in a suboptimal page allocation. To overcome this problem, we have transposed the iterative approach described in [16]. Its application sligthly improved the results (one less paging operation) at the cost of a significantly increased computation time.

For some benchmarks there is a huge difference between the two approaches (up to 75% for *fft* and 90% for *compress* for some memory sizes). This large difference can be explained by the presence of deeply nested loops. The graph coloring heuristic assigns an important weight to the webs associated with pages referenced in these loops. As a consequence, these webs are never split and the pages referenced in deeply nested loops are kept in memory during almost all program execution. The ILP formulation in contrast reduces the regions where these pages are kept in memory.

Finally, we observe that when using the ILP formulation on real-size tasks, the minimum number of paging operations is reached when using a number of physical pages lower than the number of virtual pages used. In other terms, it is possible to attain the same worst-case performance as if the task was entirely fitting in main memory, with a smaller number of pages. This observation is important during the design process of real system to avoid overestimation of memory resources.

Comparison with dynamic execution of pseudo LRU. Here we execute the five simplest benchmarks in their worst-case scenario and compare the number of costly paging operations with the one of the ILP approach. Only results on the simplest benchmarks are reported because of the difficulties to identify worst-case execution scenarios for the most complex codes. Results are given in Figure 5 for *minver* and *fft*. Globally, the number of paging operations when using ILP and demand-paging are very close to each other for all the experimented benchmarks, showing that predictability does not come at the price of performance loss.

Although the results do not show significant differences of performance between our predictable paging scheme and standard demand paging, some potential sources of performance loss in our approach have to be highlighted:

- Our approach is non-contextual which may result in performance loss in the case of functions with multiple call points. This occurs for instance in the *fft* benchmark, causing our compiler-directed scheme to have slightly worse results than PLRU for some memory sizes. A solution could be to inline the functions with multiple call points, which solves the problem for *fft* as shown in the rightmost drawing in Fig. 5.
- The granularity of the address analysis (entire arrays and stack frames) might also impact performance, in particular in case of irregular accesses to arrays in loops. We encountered this problem in a previous version of the *minver* benchmark, in which the size of a local array was erroneously too large. Since the address analysis considered that the whole array was accessed, the performance of our compiler-directed paging scheme was then slightly worse than the one of demand paging. This issue can be addressed by using a more precise (and thus more complex) address analysis.

Computation time of the ILP system. Figure 6 gives the computation time required for solving the ILP system depending on the number of physical pages available in the system for the two biggest real size tasks (see Table 4 for all numbers). Although the allocation problem is NP-complete, the computation time is reasonable even on real size tasks.

Unlike the graph coloring approach, the ILP formulation computes the page load/unload points, but not the physical page associated to every virtual page. Deciding of the mapping between virtual and physical pages is also an NP-complete problem. So far, we have tried to solve this page allocation problem using either graph coloring or an ILP formulation. Both approaches turned out to be too time-consuming to be used on real applications. Instead of defining page mappings at compile-time, we suggest to define page mappings at runtime using predictable data structures like in [3].

5. Conclusion and Future work

In this paper we have proposed an ILP formulation to introduce a predictable form of paging for real-time embedded systems. This approach considers code, static data and stack pages. Quality of results are better than our previous approach (30% on average and up to 90% in a task with deeply nested loops). In comparison to standard demand-paging systems, the predictability of our scheme was shown not to result in performance loss. In terms of computation time, experimental results have shown that this approach is usable on real size applications.

Future researches include decreasing the computation time by formulating the ILP problems at a granularity larger than the basic block (for instance loops or entire functions). Another straightforward improvement would be to define a similar ILP formulation for locking the TLB contents to have both low and predictable address translation times. Implementing pointer analysis would increase the applicability of our approach. Finally, implementation issues in a real-time operating systems have to be addressed, as well as multitasking issues, like for instance memory partitioning and selection of memory partition sizes.

Acknowledgments. The authors would like to thank J. F. Deverge, R. Guziolowski and the anonymous reviewers for their comments on earlier drafts of this paper.

References

[1] H. Abbott, R.K.; Garcia-Molina. Scheduling i/o requests with deadlines: A performance evaluation. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 113–124, Dec. 1990.

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

[3] 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.

[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] J. Deverge and I. Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *Proc. of the 19th Euromicro Conference on Real-Time Systems*, pages 179–190, Pisa, Italy, July 2007.

[6] T. Geelen. Dynamic loading in a real-time system: An overlaying technique using virtual memory. Technical report, Philips, Aug. 2005.

[7] D. W. Goodwin and K. D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996.

[8] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 30 of *ACM SIGPLAN Notices*, pages 88–98, New York, NY, Nov. 1995.

[9] M. Malkawi and J. Patel. Compiler directed memory management policy for numerical programs. *SIGOPS Oper. Syst. Rev.*, 19(5):97–106, 1985.

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

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

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

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

[14] A. Patil and N. Audsley. An efficient page lock/release os mechanism for out-of-core embedded applications. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, pages 81–88, 2007.

[15] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 217–226, Dresden, Germany, July 2006.

[16] I. Puaut and D. Hardy. Predictable paging in real-time systems: a compiler approach. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 169–178, Pisa, Italy, July 2007.

[17] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[18] 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.

[19] A. S. Tanenbaum. *Operating systems: design and implementation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[20] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3):157–179, 2000.

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

[22] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS03)*, Cancun, Mexico, 2003.

[23] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Transactions on Very Large Scale Integration Systems*, 14(8):802–815, Aug. 2006.

[24] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2-3):209–233, 1999.

[25] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. accepted for ACM Transactions on Embedded Computing Systems (TECS), 2007.