Predictable paging in real-time systems: an ILP formulation [*]

Damien Hardy Isabelle Puaut
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. In [7] we have proposed a compiler approach to introduce a predictable form of paging, in which page-in and page-out points are selected at compile-time. In [7], the problem under study was formulated as a graph coloring problem. Since the graph coloring problem is NP-complete for more than three colors, we have defined a heuristic aimed at optimizing worst-case performance. In this paper, we give a 0/1 Integer Linear Programming formulation of the same problem. Experimental results are provided to compare both approaches on code regions.

1. Introduction

Memory management is a major concern when developing real-time and embedded applications. Predictability concerns have resulted most of the times in the use of strictly *static* allocation, avoiding dynamic allocation/deallocation of 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 this paper, our focus is on the use of virtual memory.

Virtual memory consists in using hardware support (MMU: Memory Management Unit, TLB: Translation Lookaside Buffer) 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 virtual memory hardware at every memory access. When an unmapped page is referenced, a *page fault* occurs, and the operating system loads the page on demand from disk. Symmetrically, when there is no free physical page anymore, a *replacement* policy implemented by the operating system selects the page to evict. The selection process relies on usage statistics of pages maintained at run-time.

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 [8, 10, 5, 9, 4]. Demand paging raises predictability issues because *(i)* page replacement policies are often complex and far from strict Least Recently Used (LRU), *(ii)* page replacement policies are software-implemented (in the operating system) and are in general not well documented.

So far, attempts to provide real-time address spaces have focused on the predictability of virtual to physical address translation [6, 1]. In [7] we have proposed a compiler approach to introduce a predictable form of paging for code regions, in which page-in and page-out points are selected at compile-time. In that previous work, the problem under study was formulated as a graph coloring problem and a low-complexity algorithm inspired from register allocation in compilers was proposed to solve the page allocation problem. In this paper, we give a 0/1 Integer Linear Programming (ILP) formulation of the same problem. Experimental results are provided to compare both approaches on code regions.

The rest of the paper is organized as follows. Section 2 formulates the problem of off-line selection of page-in and page-out points as a 0/1 ILP problem. Two variations of the ILP formulation are given: the former and simplest one assumes that the task Worst-Case Execution Path (WCEP) is known and does not change during the page allocation process; the latter and more complex one deals with variations of the WCEP during the allocation

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

process. Experimental results applied on code are given in Section 3 and are compared to the experimental results given in [7]. Finally, we conclude in Section 4.

2. Predictable paging: an ILP approach

This section describes the 0/1 Integer Linear Programming (ILP) formulation of the page allocation problem. First, we introduce the program representation and some hypotheses. Then we present our ILP formulation of the page allocation problem. We propose two algorithms based on this ILP formulation, both aimed at minimizing the program WCET.

2.1. Program representation

We represent a program with an inter-procedural Control Flow Graph (CFG) constructed at compile-time. There is one node per basic block and one edge for every possible sequence between two basic blocks (caused by branches, function calls and function returns). We assume that referenced virtual pages of code are known at compile-time. We also consider that every basic block uses a number of virtual pages lower 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. PV is the set of virtual pages of code referenced by the whole program and $\textsc{use}_i$ is the set of virtual pages used by basic block i. (i.e. pages that must be present in physical memory during the execution of basic block i).

2.2. ILP formulation of the page allocation problem

The page allocation problem consists in determining at compile-time the points in the CFG where virtual pages will be paged-in and paged-out. This allocation must ensure that every referenced virtual pages of every basic block are present in physical memory. In the context of real-time systems, we want to minimize the WCET by reducing the number of page-ins/page-outs along the WCEP.

The ILP formulation is based on initial knowledge of edges execution frequencies along the WCEP. These frequencies can be computed by static WCET analysis. We associate to every edge between two basic blocks (i, j) its frequency $\textsc{f}_{(i,j)}$ with i a predecessor of j. In the following, symbols in small capitals denote inputs of the ILP problem, while lower-case symbols denote variables of the ILP problem. The following symbols are used in the ILP formulation:

- $\textsc{n}$ is the number of physical pages
- $\textsc{use}_i$ is the set of virtual pages referenced by basic block i
- $\textsc{f}_{(i,j)}$ is the execution frequency along the WCEP of edge (i, j)
- $\textsc{LoadCost}$ is the worst-case cost in cycles required to load one virtual page into physical memory
- $load^p_{(i,j)}$ is a binary variable which represents a page-in of virtual page p along the edge between the basic blocks i and j

$$load^p_{(i,j)} = \begin{cases} 1 & \text{virtual page } p \text{ must be paged-in} \\ & \text{on edge } (i,j) \\ 0 & \text{otherwise} \end{cases}$$

  There is one variable $load^p_{(i,j)}$ per different tuple (i, j, p) with $(i, j) \in E$ and $p \in PV$.

- pv^p_i is a binary variable which represents the presence in physical memory of virtual page p for basic block i

$$pv^p_i = \begin{cases} 1 & \text{virtual page } p \text{ is present in} \\ & \text{physical memory} \\ 0 & \text{otherwise} \end{cases}$$

  There are as many variables pv^p_i than different tuples (i, p) with $i \in V$ and $p \in PV$.

The idea of the ILP formulation is to reduce the cost of page transfers by keeping into physical memory the virtual pages which are most frequently used along the worst case execution path. The page-outs are not present in this ILP formulation because only virtual pages of code are considered.

The objective fonction to minimize is the sum of contribution to the WCET of all the page-ins along the WCEP with the consideration of their frequencies. It can be expressed as follows:

$$\sum_{(i,j)\in E} \sum_{p\in PV} \textsc{LoadCost} * \textsc{f}_{(i,j)} * load^p_{(i,j)}$$

The ILP formulation needs some extra constraints to avoid inconsistencies between variables. Two classes of constraints, given below, respectively express: *(i)* the physical memory limitation, and *(ii)* constraints on page usage (every virtual page must be loaded before use).

For each node of the CFG, the number of virtual pages present in physical memory must be lower or equal to the number of physical pages $\textsc{n}$:

$$\forall i \in V, \sum_{p\in PV} pv^p_i \leq \textsc{n} \qquad (1)$$

A page-in of a virtual page p along edge (i,j) must occur when p is not present in physical memory for node i (i.e. $pv_i^p = 0$) and p is present for node j (i.e. $pv_j^p = 1$). This boolean condition can be expressed as follows:

$\forall (i,j) \in E, \forall p \in PV$

$$\begin{array}{rcl} load_{(i,j)}^p & \leq & 1 - pv_i^p \\ load_{(i,j)}^p & \leq & pv_j^p \\ load_{(i,j)}^p & \geq & pv_j^p - pv_i^p \end{array} \quad (2)$$

To ensure the presence in memory of the virtual pages needed by the execution of a basic block i, we initialize for each basic block pv_i^p as follows:

$$\forall p \in \text{USE}_i, pv_i^p = 1 \quad (3)$$

The contents of pv_i^p when $p \notin \text{USE}_i$ 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).

2.3. Page allocation process

In this section, we describe two algorithms using our ILP formulation. The former creates an allocation in one pass whereas the latter is a greedy heuristic using an iterative approach.

First, let us define the program regions where virtual pages must be in physical memory. We will term such regions *webs* by analogy to register allocation in compilers. A web for a virtual page p is a set of consecutive basic blocks S included in V that reference p. More formally:

$$\{S,p\} \text{ is a web } \Leftrightarrow \left\{ \begin{array}{l} S \subseteq V \wedge \\ \forall i \in S, [p \in \text{USE}_i \wedge \\ (\forall y \in (pred(i) \cup suc(i))), \\ p \in \text{USE}_y \Rightarrow y \in S)] \end{array} \right.$$

with $pred(x)$ the set of the predecessors of x and $suc(x)$ the set of the successors of x in the CFG. Figure 1 is an example of two different webs associated to the same virtual page p.

One-pass page allocation algorithm

The start point of the first algorithm is the frequency of execution of edges $\text{F}_{(i,j)}$, computed on an initial allocation of pages that uses web boundaries as page transfer points. With the hypothesis $\forall i, \mid \text{USE}_i \mid \leq \text{N}$, such an allocation exists, although not optimal.

The edge frequencies $\text{F}_{(i,j)}$ are used as inputs of the ILP problem given in Section 2.2. Its resolution gives us the most interesting virtual pages to keep into physical memory for every basic block i. As an example, Figure

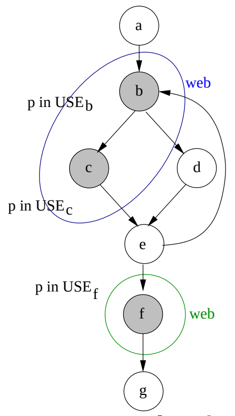

Figure 1. Example of webs

2 illustrates an allocation of a virtual page p with the ILP solution created with Figure 1. It shows that p is kept in physical memory for basic blocks d and e while p is not actually used by these basic blocks.

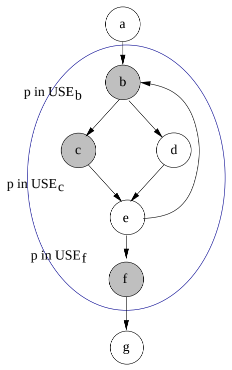

Figure 2. Page allocation after ILP solver

ILP results

$load_{(a,b)}^p = 1$
$load_{(b,c)}^p = 0$
$load_{(b,d)}^p = 0$
$load_{(c,e)}^p = 0$
$load_{(d,e)}^p = 0$
$load_{(e,f)}^p = 0$
$load_{(f,g)}^p = 0$
$pv_a^p = 0$
$pv_b^p = 1$
$pv_c^p = 1$
$pv_d^p = 1$
$pv_e^p = 1$
$pv_f^p = 1$
$pv_g^p = 0$

Iterative page allocation algorithm

The limitation of the one-pass algorithm is that one consider the task worst-case execution path as fixed during the page allocation process. Unfortunately, this is not true in general, because of the existence of multiple paths in the program CFG and because of the impact of the page-in cost on the WCEP.

Due to issue of variability of WCEP [12] during the allocation process, it may be needed to evaluate all possible page allocations to find the optimal allocation. Nevertheless, an exhaustive evaluation would be too time-consuming. We choose instead to use a greedy heuristic with an iterative page allocation process to take into account this variation.

The second algorithm iteratively applies the ILP formulation of Section 2.2. At each step, the program WCEP is re-evaluated. A set of webs is constructed with the consideration that all the virtual pages of the ILP solution (i.e. all the $pv_i^p = 1$) must be present in memory. Webs are assigned weights according to their impact on the WCET:

$$weight_w = \sum_{x \in w.S | w.p \in \mathrm{USE}_x} \sum_{i \in pred(x)} \mathtt{F}_{i,x}$$

Let w be the non-examined yet web, associated to page p, with the highest weight at a step k. Let i be a basic block of w. At step $k+1$, the ILP system is generated and solved, with pv_i^p not a variable anymore but now assigned to the result of the ILP system at step k. This iterative process is repeated until all variables pv_i^p are no longer variables in the ILP system.

3. Experimental results

We are interested in evaluating the worst-case timing behavior of programs. Estimation of WCETs is completed using static program analysis. We describe experimental conditions in Section 3.1 and experimental results are given in Section 3.2.

3.1. Experimental setup

WCET estimation. Our experiments were conducted on MIPS R2000/R3000 binary code. The WCETs of tasks are computed by the Heptane[1] timing analyser [2]. The implicit path enumeration technique WCET computation method of Heptane provides the frequency of edges between basic blocks along the WCEP.

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. A page-in time of 1 million cycles is assumed and we use pages of 128 bytes; page size is small to stress the paging activity even on rather small benchmarks.

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

Benchmarks. The experiments were conducted on six 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/).

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

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

3.2. Results

The main performance metric used 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.

The comparison between our ILP approach and the graph coloring approach [7] are conducted first with the one-pass page allocation algorithm and then with the iterative approaches. The results are presented in Figure 3. The re-evaluation parameter of the WCEP is not strict linear like in [7] but stopped after 10 iterations. This choice is motivated by a need to reduce the execution time of the iterative process without loosing much precision. Indeed, we observed in preliminary experiments that the results of the strict linear and this re-evaluation frequencies are close to each other. This can be explained by the fact that the WCEP variations occur generally during the first page allocations.

The one-pass page allocation results of our ILP formulation is always better than (i.e. have a smaller number of page-ins) or identical to the coloring approach. The improvement is around 30%[3] on average and up to 90% in the best cases.

Similarly to the graph coloring approach, the results of the iterative approach are close to the one-pass page allocation (less than 2% of variation). We explain this by the fact that there is very little data dependent code in the benchmarks. A closer analysis of a larger and more varied set of benchmarks is still needed and is left for future work.

In term of execution time, we observe that the computation time of the solver take just few seconds for every benchmark. Thus, this approach gives an optimizing worst-case performance comparatively to our previous approach without increasing the computation time.

4. Conclusion and Future work

In this paper, we have presented a new approach for the selection of page-in and page-out points to introduce a form of predictable paging in real-time systems.

The results shows a general improvement of 30% on average and up to 90% comparatively to our previous approach based on a graph coloring method. Moreover, this general improvement is not time consuming. In fact, on every benchmark, the computation time to solve the ILP problem was just a few seconds.

In contrast to the graph coloring approach, this formulation is dependent on the type of pages referenced and

[3]This average is made with the average of each benchmark

Name	Description	Code size (bytes)
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

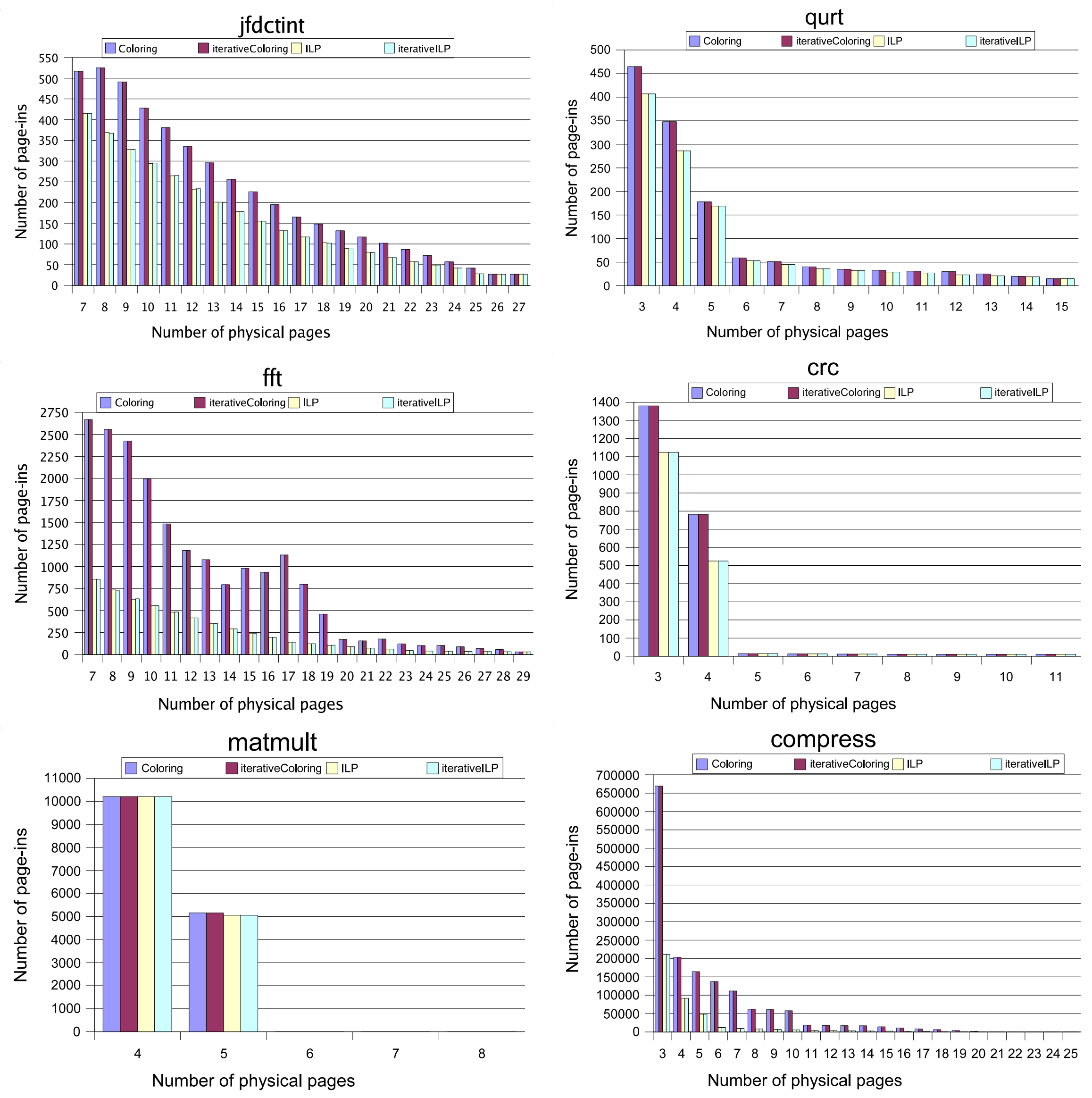

Figure 3. Comparison between ILP and coloring approach

only code pages are considered at this time. Other works [3, 11] proposed ILP formulations for data allocation to minimize the WCET and respectively to reduce the energy consumption. Their approaches are the same in spirit but it is not applicable to the case of data pages because they allocate data in scratchpad memory and do not need to guarantee the presence of them in this stage of the memory hierarchy. We want to extend our formulation to data pages in our future work.

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

References

[1] 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.

[2] 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.

[3] J. Deverge and I. Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, Pisa, Italy, July 2007. To appear.

[4] 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. volume 2211, pages 469–485, Tahoe City, CA, Oct. 2001.

[5] 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.

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

[7] 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*, Pisa, Italy, July 2007. To appear.

[8] 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.

[9] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, Sept. 1989.

[10] P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph based approach. *Real-Time Systems*, 13(1):67–91, July 1997.

[11] 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.

[12] W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller. Improving WCET by optimizing worst-case paths. In *Proceedings of the 11st IEEE Real-Time and Embedded Technology and Applications Symposium*, Mar. 2005.