# Impact of Cache Partitioning on Multi-Tasking Real Time Embedded Systems*

Bach D. Bui, Marco Caccamo, Lui Sha
University of Illinois at Urbana-Champaign
Department of Computer Science
{bachbui2, mcaccamo, lrs}@cs.uiuc.edu

Joseph Martinez
Lockheed Martin Aeronautics Company
Systems Software
joseph.t.martinez@lmco.com

## Abstract

*Cache partitioning techniques have been proposed in the past as a solution for the cache interference problem. Due to qualitative differences with general purpose platforms, real-time embedded systems need to minimize task real-time utilization (function of execution time and period) instead of only minimizing the number of cache misses. In this work, the partitioning problem is presented as an optimization problem whose solution sets the size of each cache partition and assigns tasks to partitions such that system worst-case utilization is minimized thus increasing real-time schedulability. Since the problem is NP-Hard, a genetic algorithm is presented to find a near optimal solution. A case study and experiments show that in a typical real-time embedded system, the proposed algorithm is able to reduce the worst-case utilization by* $15\%$ *(on average) if compared to the case when the system uses a shared cache or a proportional cache partitioned environment.*

## 1. Introduction

Modern real-time embedded platforms are complex integrated systems where several real-time tasks execute in multi-tasking environments and compete for shared resources like cpu, bus, memory, etc. Safety critical avionic systems need to meet stringent temporal constraints and software modules characterized by different criticalities[1] are run in distinct scheduling partitions temporally isolated from each other. The avionic ARINC 653 standard prescribes a partitioned architecture where multiple software partitions can be safely executed on a single CPU by enforcing logical and temporal isolation. It is important to notice that in a CPU partitioned environment it is still possible to experience inter-partition dependencies due to other globally shared hardware resources like cache or bus. More in details, the sequential execution of different scheduling partitions on a mono-CPU platform (see cyclic executive scheduler [6]) causes cache lines of a partition to be invalidated by the execution of another under the common assumption that cache is globally shared. This inter-partition dependency introduces two serious problems: 1) it clearly violates the requirement of temporal isolation required by avionic ARINC 653 standard; 2) the execution time of tasks experience large variance increasing the difficulty of estimating worst-case execution time (WCET). To address the first problem, a predictable bus scheduling policy should be adopted along with a cache partitioning strategy. In the case of the second problem, cache partitioning itself suffices to significantly reduce execution time variance and consequently tasks' WCET. While in [9] Pellizzoni and Caccamo address the bus scheduling problem and propose a solution for COTS-based systems leveraging the idea of a "hardware server", this work focuses on the cache interference problem common to multi-tasking real-time systems (problem 2). As future work, we plan to integrate a cache partitioning technique with predictable bus scheduling to fully address the above mentioned problems and provide a complete solution compliant to the avionic ARINC 653 standard. The focus of this work is on the impact of the last level of cache since its interference affects system performance the most. For example, if we consider the PowerPC processor MPC7410 (widely used for embedded systems) which has $2MB$ two-way associative L2 cache (see Table 1 for further details), the time taken by the system to reload the whole L2 cache is about $655\mu s$. This reloading time directly affects task execution time. A typical partition size of an avionic system compliant to ARINC-653 can be as small as $2ms$. Under this scenario, the execution time increment due to cache interference can be as big as $655\mu s/2ms \approx 33\%$: hence, the multi-task cache interference problem can be severe in embedded systems. In general, the effect of cache interfer-

* This material is based upon work supported by Lockheed Martin Aeronautics and by the NSF under Awards No. CNS0720512, CNS0720702, CNS0613665, CCF0325716. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

1 There exist four criticality levels from most critical to less critical: A, B, C, and D.

ence on task execution time is directly proportional to cache size and CPU clock frequency and inversely proportional to memory bus speed. It is worth noticing that since CPU, memory bus speed, and cache size are constantly increasing in modern computer architectures, it is unlikely that this problem will be less severe in the near future. Another important factor to take into account is that typical embedded applications have a task memory footprint within the range of $50KB - 250KB$ [2]; as a consequence, cache partitioning can be exploited to enhance real-time schedulability by means of reducing tasks' utilization.

Several cache partitioning techniques have been devised [3, 5, 7, 15]; however compared to previous works, the main contribution of this paper is to exploit cache partitioning to improve real-time schedulability while taking into account tasks' criticality. In fact, due to qualitative differences with general purpose platforms, real-time embedded systems need to minimize task real-time utilization (function of execution time and period) instead of only minimizing the number of cache misses. More in details, the partitioning problem is presented as an optimization problem (minimizing worst-case utilization) whose solution is expressed by the size of each cache partition and the assignment of tasks to partitions. Since the problem is NP-Hard, a genetic algorithm is used to find a near optimal solution. A case study and experiments show that in typical real-time embedded systems, the proposed algorithm is able to reduce the worst-case utilization by $15\%$ (on average) if compared to the case when the system uses a shared cache or a proportional cache partitioned environment. Finally, notice that our mechanism is not a replacement for WCET estimation techniques [11, 12, 4]: in fact, it effectively leverages on them to achieve a higher level of efficiency and predictability. The rest of the paper is organized as follows. The next section discusses an overview of related works; Section 3 describes terminology, Section 4 describes the problem formulation and the proposed solution. An approximate utilization lower bound is derived in Section 5. The evaluation of the proposed solution by using real and simulated data is shown in Section 6.

## 2. Related Works

The implementation aspects of cache partitioning techniques are the focus of most of the previous works [3, 15, 5, 7]. SMART, a hardware-based strategy, was proposed by Kirk [3]. The cache memory is divided into equal-sized small segments and one large segment is referred to as shared pool. The large segment is shared by non-critical tasks while the small ones are dedicated to real-time tasks or groups of real-time tasks. Hardware-based cache partitioning has the benefit of being transparent to higher layers thus requiring little software modification. However its major disadvantages, such as having only fix partition sizes and requiring custom-made hardware, make software-based approaches better choices in practice.

The idea of software-based cache partitioning techniques was first proposed by Wolfe in [15]. By means of software, the code and data of a task are logically restricted to only memory portions that map into the cache lines assigned to the task. In essence, if the task memory footprint is larger than its cache partition, its code and data must reside in memory blocks that are regularly fragmented through out the address space. In [5], Liedtke extended Wolfe's idea exploring the use of operating systems to manage cache memory. By mapping virtual to physical memory, an operating system determines the physical address of a process, thus also determines its cache location. In contrast, Mueller [7] investigated using compilers to assign application code into physical memory. During compilation process, code is broken into blocks, blocks are then assigned into appropriate memory portions. Since the code address space is no longer linear, the compiler has to add branches to skip over gaps created by code reallocation. Due to the observation that tasks at the same priority level are scheduled non-preemptively with respect to each other, the author suggested that all tasks can be accommodated by using a number of partitions which is no more than the number of priority levels. Although the argument is true in priority driven systems, it can not be applied to cyclic executive ones.

In this research, we advocate the use of OS-controlled techniques like that one in [5] but we look at the cache partitioning problem from a different angle: the efficiency aspect of the cache partitioning mechanism in terms of system schedulability. To the best of our knowledge, there has not been any research in this direction.

## 3. Terminology and assumptions

In this section we describe the terminology used in this paper. We consider a single processor multi-tasking real-time system $S$ as a pair $S = \left\{\mathcal{T}, K^{size}\right\}$, where $\mathcal{T}$ is a task set of size $N$: $\mathcal{T} = \{\tau_i : i = [1, N]\}$, $K^{size}$ is total number of cache partition units available in the system. Let $\delta$ be the size of a cache partition unit. Note that the value of $\delta$ depends on the cache partitioning technique employed: for example, considering an OS-controlled technique, the value $\delta$ is the page size, e.g. 4KB. In this case, if CPU has $2MB$ cache, then $K^{size} = 2048KB/4KB = 512$ units. Denote as $U_{wc}$ the worst-case system utilization.

Regarding task parameters, each task $\tau_i$ is characterized by a tuple $\tau_i = \left\{p_i, exec_i^C(k), CRT_i(k)\right\}$ where $p_i$ is the task period, $exec_i^C(k)$ is cache-aware execution time, $CRT_i(k)$ is cache reloading time, and $k$ is an integer index that represents cache size $k * \delta$. Functions $exec_i^C(k)$ and $CRT_i(k)$ will be formally defined in the following paragraphs.

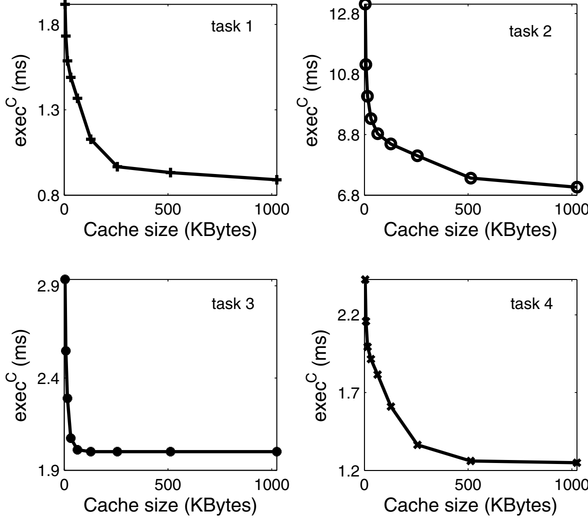


Figure 1: $exec^C$ of avionic applications

**Definition 1** *The cache-aware execution time $exec_i^C(k)$ of task $\tau_i$ is the worst case execution time of the task when it runs alone in a system with cache size $k * \delta$.*

There have been many techniques proposed for measuring single task worst-case execution time including those that take into account cache effect [13]. In this paper we assume that function $exec_i^C$ can be obtained by measuring $\tau_i$'s execution time for different cache sizes by using an available tool. Figure 1 depicts $exec^C$ functions of four tasks in an experimental avionic system. According to the figure, it can be noticed that $exec_i^C(k)$ is composed of two sub-intervals. More precisely, the cache-aware execution time is decreasing in the first sub-interval and becomes almost constant in the second one. Intuitively, we can explain this phenomenon as it follows: when the cache size is smaller than a task's memory footprint, the execution time diminishes as the cache size increases due to a reduction of cache misses; on the other hand, when the cache size is larger than task's memory footprint ($k_i^0$), the execution time is no longer cache-size dependent since the number of cache hits does not change.

Next, we define the cache reloading time function.

**Definition 2** *The cache reloading time function $CRT_i(k)$ is total CPU stall time due to cache misses caused by the preemptions that $\tau_i$ can experience within a period $p_i$.*

$CRT_i(k)$ is the overhead that occurs only in multi-tasking systems and takes into account the effect of cache lines invalidated by preempting tasks. In general, $CRT_i(k)$ depends on the maximum number of times that $\tau_i$ can be preempted and the size of cache partition that $\tau_i$ is using. A good technique to estimate a safe upper bound of $CRT_i(k)$ for fixed priority scheduling is proposed in [12]. This technique can be easily extended for cyclic executive. In this paper, we assume that given a task set $\mathcal{T}$, a technique as described in [12] can be used to find $CRT_i(k)$ for each task $\tau_i$.

| Processor | MPC7410 |
|---|---|
| CPU Speed | 1000Mhz |
| L2 cache | 2MB two-way set associative |
| Memory bus speed | 125Mhz |
| Instruction/Data L1 miss + L2 hit latency | 9/13 CPU-cycles |
| Memory access latency | 17 Memory-cycles / 32 bytes |

Table 1: PowerPC configuration

With the problem at hand, two unknown variables need to be determined for each task $\tau_i$:

- $k_i$: size of the cache partition assigned to the task $\tau_i$
- $a_i$: an indication variable whose value is $a_i = 1$ if task $\tau_i$ uses a private cache partition or 0 otherwise.

For convenience, we call a tuple $[k_i, a_i]$ as an *assignment* for task $\tau_i$, and $\{a_i : \forall i \in [1, N]\}$ as an *arrangement* of tasks. Having defined all elements, it is now possible to compute the $WCET$ of a task $\tau_i$ given an assignment $[k_i, a_i]$:

$$WCET_i(k_i) = exec_i^C(k_i) + (1 - a_i) \times CRT_i(k_i) \quad (1)$$

## 4. The cache partitioning problem

In this section, the partitioning problem is presented as an optimization problem (minimizing worst-case utilization) whose solution is expressed by the size of each cache partition and the assignment of tasks to partitions. A genetic algorithm is introduced to find near optimal solutions and an approximate utilization lower bound is presented that will be used as a comparison metric to evaluate the effectiveness of our genetic algorithm. Before stating the optimization problem, we describe a simple experimental setup with two real-time tasks scheduled according to fixed priorities that shows the impact of the cache interference problem with regard to task utilization.

### 4.1. The impact of cache interference

To have a quantitative measure of the cache interference problem, we conducted an experiment with a real system and its results have strongly motivated this work. In our test-bed we used a DELL with Intel Pentium 4 $1.5GHz$ processor, memory bus speed $400MHz$, and 256KB L2 cache. LinuxRK [8], a real-time kernel developed at CMU, was used as the test-bed operating system. To measure the effect of multi-task cache interference, we used a fixed priority scheduler and two processes running at different priorities: 1) the low priority process $\tau_{low}$ has two different

| $\tau_{low}$ $(ms)$ \ $\tau_{high}$ $(ms)$ | $(1,2)$ | $(2,4)$ | $(5,10)$ |
|---|---|---|---|
| $(5,10)$ | 13.65% | 6.1% | – |
| $(10,20)$ | 13.6% | 6.15% | 2.35% |

Table 2: Task utilization increment.

pairs of execution times[2] and periods: $(5ms, 10ms)$ and $(10ms, 20ms)$; 2) the high priority one $\tau_{high}$ has three different pairs of execution times and periods: $(1ms, 2ms)$, $(2ms, 4ms)$, $(5ms, 10ms)$. For each experiment, there were two runs: during the first one the high priority task limited the number of operations involving a memory access; during the second one, the high priority task was run such that it invalidated as many cache lines as possible of the low priority process. The execution time of $\tau_{low}$ was measured during both runs and $\tau_{low}$'s utilization increment was computed for each experiment. The results when using a MPEG decoder application as the low priority process are shown in Table 2.

According to Table 2, the task utilization increment can be as high as 13% even though the system has a small L2 cache of size 256KB. It is worth noticing that the utilization increment of $\tau_{low}$ is independent of its period (assuming constant task utilization) while it increases inversely proportional to $\tau_{high}$'s period. The results of this simple experiment are in agreement with the case study and extensive simulations of Section 6; in fact, 13% task utilization increment can occur even when $\tau_{low}$ is suffering a rather limited number of preemptions due to a higher priority task.

### 4.2. Genetic algorithm

The cache partitioning problem is now formulated as an optimization problem whose objective function is to minimize the worst-case system utilization under the constraint that the sum of all cache partitions cannot exceed $K^{size}$ (total available cache size) and all the safety critical tasks (i.e., level A and B of task criticality) should be assigned to a private cache (i.e., $a_i = 1$). Note that a less critical task can use either a shared or a private cache partition.

- **Problem Statement**: Given a real-time system $S = \{\mathcal{T}, K^{size}\}$, $\mathcal{T} = \{\tau_i : i = [1..N]\}$, and $\forall i \in [1, N] : \tau_i = \{p_i, exec_i^C(k), CRT_i(k)\}$, find a system configuration $C^{opt} = \{[k_i, a_i] : \forall i \in [1, N]\}$ that minimizes system worst-case utilization $U_{wc}$.

2 Note that these execution times were measured in an experimental setup with very limited cache interference.

The mathematical definition of the mentioned optimization problem follows:

$$\begin{array}{ll} \min & U_{wc} = \sum_i \frac{WCET_i(k_i)}{p_i} \\ \text{s.t.} & \sum_i a_i \cdot k_i + k^{share} \leq K^{size} \\ & \forall i \in [1, N]: \ 0 < k_i \leq K^{size} \\ & \forall i \in [1, N]: \ k^{share} = k_i \ \mid \ a_i = 0 \\ & \forall i \in [1, N]: \ a_i = 1 \mid \tau_i \text{ is a safety critical task} \end{array} \quad (2)$$

Let $U_{wc}^{opt}$ be the minimum value of the objective function. This problem always has a feasible solution. In general, two questions need to be answered for each task: 1) should we put the task in private or shared cache? (except for the case of safety critical tasks). 2) what is the size of the cache partition? Notice that all tasks that are assigned to the shared partition will have the same cache size as a solution for the optimization problem and it is set equal to $k^{share}$. This cache partitioning problem is NP-hard since it can be reduced to knapsack problem in polynomial time.

In the remaining of this section, we describe our genetic algorithm that is used to solve the optimization problem. The algorithm is based on the GENOCOP framework [16] which has been shown to perform surprisingly well on optimization problems with linear constrains [3]. The constraint-handling mechanism of GENOCOP is based on specialized operators that transform feasible individuals into other feasible individuals.

**Algorithm 1** Cache Partitioning Genetic Algorithm

**Input:** $S = \{\mathcal{T}, K^{size}\}$
**Output:** $\{[k_i, a_i] : \forall \tau_i \in \mathcal{T}\}$

```
1: g ← 0 Initialize P(g)
2: while g ≤ G_max do
3:     mutate some individuals of P(g)
4:     cross over some individuals of P(g)
5:     locally optimize P(g)
6:     evaluate P(g)
7:     g ← g + 1
8:     select P(g) from P(g − 1)
9: end while
```

Our solution is shown in Algorithm 1. Specific information of the problem at hand is employed to design the operators which are local optimizer, mutation and crossover. At each generation $g$, a set of individuals (i.e., population $P(g)$) is processed. Individuals are feasible solutions of the optimization problem. A non-negative vector $X$ of fixed length $N + 1$ is used to represent an individual, where: $N$ is number of tasks; and $\forall i \in [1, N]$ if $X[i] > 0$, then $X[i]$

3 Notice that the considered optimization problem involves the evaluation of a non-monotonic function $CRT_i(k)$. As a consequence, solvers as hill climbing or simulated annealing could be easily trapped within a local optimum. Hence, a genetic algorithm was chosen to circumvent this problem.

is the size of the private cache of $\tau_i$, if $X[i] = 0$, then $\tau_i$ uses the shared partition of size $X[N+1]$. For example, $X = [2, 3, 0, 0, 5]$ means $\tau_1$ and $\tau_2$ use two private partitions with $k_1 = 2$ and $k_2 = 3$, whereas $\tau_3$ and $\tau_4$ use the shared partition with $k_3 = k_4 = k^{share} = 5$. Individuals are evaluated (line 6) using Equation 2. The outcome of the evaluation i.e. $U_{wc}$ of each individual is then used to select which individual will be in the next generation. The lower $U_{wc}$ an individual has, higher is the probability that it will survive. Let $U^h_{wc}$ be the utilization of the output configuration i.e. the lowest utilization found by the algorithm. The three operators (i.e., local optimizer, mutation, and crossover) are described in the following sections. In the following algorithms, all random variables generated by instruction $random$ have uniform distribution.

### 4.3. Local Optimizer

Since $exec^C_i(k)$ is a non-increasing function, increasing $X[i]$ always results in a smaller or equal utilization of $\tau_i$ thus a smaller or equal system utilization. This observation leads to the design of a simulated annealing local optimizer as showed in Algorithm 2. Each individual of population $P(g)$ undergoes the local improvement before being evaluated. The original individuals are then replaced by the local optima. This heuristic reduces the search space of the genetic algorithm to only those solutions that are locally optimal.

**Algorithm 2** Simulated Annealing Local Optimizer

**Input:** $X[1..N+1]$
**Output:** $X[1..N+1]$

1: $T \leftarrow T_{max}$
2: **while** $T > T_{min}$ **do**
3: $tries \leftarrow 0$
4: **while** $tries < tries_{max}$ **do**
5: $X' \leftarrow X$
6: $i \leftarrow random[1, N]$ such that $X[i] > 0$
7: $X'[i] \leftarrow X'[i] + random(X'[i], K^{size} - \sum_{j=1, j\neq i}^{j=N} X'[j])$
8: $\Delta U \leftarrow U'_{wc} - U_{wc}$
9: **if** $\Delta U < 0$ and $random[0, 1] < 1/(1 + e^{\Delta U/T})$ **then**
10: $X[i] \leftarrow X'[i]$
11: **end if**
12: $tries \leftarrow tries + 1$
13: **end while**
14: reduce $T$
15: **end while**

At each trial, an $X[i]$ is chosen at random (line 6) and its value is increased by a random amount within its feasible range (line 7) . The new value is accepted with a probability proportional to temperature $T$. Only variables representing private cache are considered (i.e. $X[i] > 0 \quad \forall i \in [1, N]$). Note that enlarging the size of the shared partition, $X[N+1]$, does not necessarily result in reducing utilization since it may increase the cache reloading time. Obviously, this heuristic can only find approximate local optima with respect to a certain *arrangement* of tasks; the global optimum might require a task to use a shared partition instead of a private one (or vice-versa) as assigned by the local optimizer. The mutation and crossover operators are designed to allow the algorithm to search beyond local optima.

### 4.4. Mutation Operator

Algorithm 2 minimizes the utilization by enlarging size of private cache partitions thus the solution is only locally optimal with respect to a certain *arrangement* of tasks. The mutation operator described in Algorithm 3 helps to search beyond local optima by randomly rearranging tasks into a private or shared cache partition. In other words, it creates new landscape where the local optimizer can work.

**Algorithm 3** Mutation Operator

**Input:** $X[1..N+1]$
**Output:** $X[1..N+1]$

1: $flip \leftarrow random\{TAIL, HEAD\}$
2: **if** $flip = TAIL$ **then**
3: $i \leftarrow random[1, N+1]$
4: assign $X[i]$ to a random value in its feasible range
5: **else**
6: $i \leftarrow random[1, N]$ such that $X[i] > 0$
7: $X[i] \leftarrow 0$
8: **end if**

The operator takes as input an individual at the time. The operator randomly chooses a task $\tau_i$ for either modifying its private cache size or rearranging the task into shared cache by assigning 0 to $X[i]$. The size of shared cache partition may also be changed. The operator guarantees that the generated individual is feasible.

### 4.5. Crossover Operator

The crossover operator (Algorithm 4) aims to transfer good chromosome of the current generation to the next one. Algorithm 2 finds, for each individual, a local optimum with respect to its *arrangement* of tasks. After undergoing local optimization, if one individual has lower utilization than another, there is a high probability that the former has a better *arrangement*. In other words, the genetic advantage of an individual over another is implicitly expressed by its *arrangement*. Our design of crossover operator exploits this information.

The operator spawns descendants by randomly combining the *arrangement*s of pairs of predecessors. The algo-

rithm takes two parents as input and produces two children. If $\tau_i$ of one of the parents uses shared cache (i.e. $X[i] = 0$), the *assignment* of the child's $\tau_i$ is the *assignment* of either of the parent's $\tau_i$, otherwise it is the arithmetic crossover (i.e. $b * X_1[i] + (1 - b)X_2[i]$ where $b$ is a continuous uniform random variable taking values within $[0, 1]$). The operator guarantees that if parents are feasible then their children are also feasible.

**Algorithm 4** Crossover Operator

**Input:** $X_1[1..N+1], X_2[1..N+1]$
**Output:** $X_1'[1..N+1], X_2'[1..N+1]$

```
1:  a ← random[0,1]
2:  for i = 1 to N+1 do
3:    if X1[i] = 0 or X2[i] = 0 then
4:      flip ← random{TAIL, HEAD}
5:      if flip = TAIL then
6:        X1'[i] ← X1[i]
7:        X2'[i] ← X2[i]
8:      else
9:        X1'[i] ← X2[i]
10:       X2'[i] ← X1[i]
11:     end if
12:   else
13:     X1'[i] ← b * X1[i] + (1 − b) * X2[i]
14:     X2'[i] ← b * X2[i] + (1 − b) * X1[i]
15:   end if
16: end for
```

## 5. Approximate Utilization Lower Bound

In this section we derive an approximate utilization lower bound for the cache partitioning problem which will be used to evaluate our heuristic solution. The bound gives a good comparison metric to evaluate the effectiveness of a given partitioning scheme and is approximate because it is computed by using $exec_i^C(j)$ and $CRT_i(j)$ functions. In realistic scenarios, these two functions are derived experimentally and have some approximation errors. Hence, the bound is approximate too. A definition of approximate utilization lower bound $U_b$ follows.

**Definition 3** *An approximate utilization lower bound $U_b$ of the cache partitioning problem (defined in Section 4) is a value that satisfies $U_b \leq U_{wc}^{opt}$.*

The bound $U_b$ is easily computed starting from an initial phase that assumes unlimited cache size and all tasks have a private cache size of their memory footprint. Then, at each step the total cache size is reduced either by shrinking the size of a private partition or by moving a task from private cache to shared one: the decision is made in such a way that the increment of total task utilization is minimized at each step. This technique is similar[4] to Q-RAM [10], and the iterative algorithm is executed until the total cache size is reduced to $K^{size}$.

Although the bound value is not exactly $U_{wc}^{opt}$, as will be shown in Section 6, it is still a good measure for the performance of any heuristic solving the analyzed optimization problem. In addition, system designers can also use this bound to predict how much utilization at most can be saved when applying any partitioning heuristic. According to equations 1 and 2, the utilization of a task may increase or decrease depending on its cache attributes like size, private/shared partition and cache reloading time. The derivation of $U_b$ is based on the notion of two distinct utilization differentials per cache unit: $\Delta_i^e(j)$ and $\Delta_i^r(j)$. In fact, $\Delta_i^e(j)$ indicates how a task's utilization varies as a function of the size of its private cache partition, and $\Delta_i^r(j)$ is a normalized index on how task's utilization varies when switching from private to shared cache. The formal definitions of $\Delta_i^e(j)$ and $\Delta_i^r(j)$ follow.

**Definition 4** *The utilization differential $\Delta_i^e(j)$ of task $\tau_i$ is the difference in utilization per cache unit when $\tau_i$'s private cache is reduced from $j$ to $j-1$ units.*

$$\Delta_i^e(j) = \frac{exec_i^C(j-1) - exec_i^C(j)}{p_i} \quad (3)$$

Task $\tau_i$'s utilization may increase when changing from private to shared cache due to the presence of cache reloading time. If $\tau_i$'s private cache size is $j$ then when the switching takes place, the amount of freed cache is $j$ units. Thus we have the following definition of utilization differential $\Delta_i^r(j)$ caused by cache reloading time.

**Definition 5** *The utilization differential $\Delta_i^r(j)$ of task $\tau_i$ is the difference in utilization per cache unit caused by cache reloading time when $\tau_i$ is moved from a private partition of size $j$ to a shared partition of the same size.*

$$\Delta_i^r(j) = \frac{CRT_i(j)}{j * p_i} \quad (4)$$

Having introduced the basic notions, we now describe the algorithmic method for estimating $U_b$ (Algorithm 5). Since $exec_i^C$ is a non increasing function, $\tau_i$ has its smallest utilization when its cache is private and has maximal size (i.e. $[k_i = k_i^0, a_i = 1]$). Consequently, the system utilization is minimized since every task has its own private cache of maximum size. That absolute smallest system utilization ($U_b^{abs}$) and the total needed cache size ($K$) are calculated in line 8 and 9, respectively. This configuration, however, is not feasible when $K > K^{size}$ which holds in most practical systems. Note that after line 10, $U_b = U_b^{abs}$ is a lower bound following Definition 3. Nevertheless, it is not the tightest bound that can be found in polynomial time.

4 Q-RAM starts always from a feasible state with minimum resource allocation, while our algorithm starts from an infeasible state by assuming unlimited resource availability.

A tighter bound is estimated using procedure starting from line 13. Essentially, it reduces the value of $K$ toward that of $K^{size}$ (line 15 and 19). Then for each unit of cache size taken from $K$, the value of $U_b$ is increased (line 16 and 20) such that $U_b$ approaches but **never exceeds** $U_{wc}^{opt}$. This is done by using the smallest values of $\Delta^e$ and $\Delta^r$ to update $U_b$ at each step. The correctness of the procedure is proven in the following paragraphs.

Consider a configuration $C = \{[k_i = k_i^0, a_i = 1] : \forall i \in [1, N]\}$ that uses total cache size $K = \sum_i k_i^0 > K^{size}$, there are three basic operations that can be executed to let $C$ to converge toward $C^{opt}$:

1. reducing the size of any private partition by 1 unit thus reducing $K$ by 1 unit and increasing $U_b$ by a value $\Delta_i^e(j)$.
2. moving a task from its private partition of size $j$ to a shared partition of the same size, thus reducing $K$ by $j$ units and increasing $U_b$ by a value $\Delta_i^r(j) * j$.
3. reducing the size of the shared partition by 1 unit thus reducing $K$ by 1 unit.

Lemma 1 shows that since operation 3 is equivalent to a sequence of the other two, only operation 1 and 2 are needed to compute $U_b$.

**Lemma 1** *Every sequence of operations used to compute $U_b$ can be converted to a sequence of operations 1 and 2.*

**Proof.**
We only need to prove that any sequence of operation 3 can be represented as a sequence of operations 1 and one operation 2. Assume that the final size of the shared partition is $k^{share}$. We can always reduce the size of any task's private partition to $k^{share}$ using operation 1, then by applying operation 2 those tasks can be moved to the shared partition and $U_b$ can be computed without using operation 3. □

Using Lemma 1, we can prove the following theorem that implies the correctness of Algorithm 5

**Theorem 2** *The output of Algorithm 5 ($U_b$) is smaller or equal to $U_{wc}^{opt}$*

**Proof.**
Lemma 1 proves that every transformation applied to compute $U_b$ is composed of sequences of operation 1 and 2. Consider a task $\tau_i$ currently using a private cache of size $j$: for each operation 1 applied to $\tau_i$, $U_b$ increases by $\Delta_i^e(j)$ and $K$ decreases by 1; for each operation 2 applied to $\tau_i$, $U_b$ increases by $\Delta_i^r(j) * j$ and $K$ decreases by $j$. Since Algorithm 5 uses the smallest value among $\Delta^e$ and $\Delta^r$ to update $U_b$ at each step, after the while loop (when $K = K^{size}$), $U_b \leq U_{wc}^{opt}$. The time complexity of the while loop is bounded by $\sum_i k_i^0 - K^{size}$ □

Note that $U_b$ is an utilization lower bound and Algorithm 5 does not produce a feasible solution since it might end up splitting the cache of a task into a private part and a shared one.

**Algorithm 5** $U_b$ Estimation

**Input:** $S = \{\mathcal{T}, K^{size}\}$
**Output:** $U_b$

```
1:  for i = 1 to N do
2:      for j = 1 to k_i^0 do
3:          EXE[sum_{l=1}^{i-1} k_l^0 + j] ← Δ_i^e(j)
4:          REL[sum_{l=1}^{i-1} k_l^0 + j] ← Δ_i^r(j)
5:      end for
6:  end for
7:  sort EXE and REL in decreasing order
8:  U_b^abs ← sum_i exec_i^C(k_i^0)/p_i
9:  K ← sum_i k_i^0
10: U_b ← U_b^abs
11: e ← size of EXE
12: r ← size of REL
13: while K > K^size do
14:     if EXE[e] < REL[r] then
15:         K ← K − 1
16:         U_b ← U_b + EXE[e]
17:         e ← e − 1
18:     else
19:         K ← K − min(K − K^size, j)
20:         U_b ← U_b + min(K − K^size, j) * REL[r]
21:         r ← r − 1
22:     end if
23: end while
```

## 6. Evaluation

This section evaluates the proposed algorithm by using input data taken from real and simulated systems. Although the solution can be applied to systems with any scheduling policy, all the experiments in this section assume a cyclic executive scheduler. This assumption is motivated by the fact that the cyclic executive scheduler is commonly used by avionic industry due to the high criticality of the developed real-time systems. In our evaluation, we are concerned only with the last level of cache as it affects system performance the most. However, it is noted that taking into account other cache levels would not change the performance of the proposed algorithm. We start first by describing the methodology used to generate input data, then we show the experimental results.

### 6.1. Methodology

This section discusses our method to simulate cache access patterns and to generate functions $exec^C$ and $CRT$. In

[14], Thiébaut developed an analytical model of caching behavior. The model is provided in the form of a mathematical function (Equation 5) assigned to each task that dictates the cache miss rate for a given cache size $k$. Although it was originally developed for a fully associative cache, it has been verified that the model generates synthetic miss rates that are very similar to the cache behavior of actual traces across the complete range of cache associativities [14].

$$MissRate(k) = \begin{cases} \frac{1-\frac{k}{A_1}(1-\frac{1}{\theta})-A_2}{1-A_2} & \text{if } k \leq A_1 \\ \frac{\frac{A^\theta}{\theta}k^{(1-\theta)}-A_2}{1-A_2} & \text{if } A_1 < k \leq k^0 \\ 0 & \text{if } k^0 < k \end{cases} \quad (5)$$

$$A_1 = A^{\theta/(\theta-1)} \quad (6)$$

$$A_2 = \frac{A^\theta}{\theta}k^{0^{(1-\theta)}} \quad (7)$$

Cache behavior is function of three task-dependent parameters, $A, \theta, k^0$. $A$ determines the average size of a task's neighborhood (i.e., working set). The higher $A$ is, the larger the task's neighborhood becomes. Notice that a larger working set increases the probability of a cache miss. $\theta$ is the locality parameter which is inversely proportional to the probability of making large jumps. The probability that a memory access visits a new cache line diminishes as $\theta$ increases. It has been shown statistically that real applications have $\theta$ ranging from 1.5 to 3 [14]. It is worth noticing that Equation 5 only models the capacity cache miss. A complete model needs to also encapsulate the effect of compulsory cache miss (i.e., cold cache miss). A system with single or non-preemptive tasks has cache miss rate converges to that of Equation 5. However this is not the case in preemptive multi-tasking systems where compulsory cache miss becomes significant due to frequent preemptions. Equation 8 proposed by Dropsho [1] modifies Equation 5 to calculate the miss rate occurring at the warm-up phase too. The improved model calculates the instantaneous miss rate by using the current number of unique cache entries as the instantaneous effective cache size.

$$InstantRate(m, k) = \begin{cases} MissRate(m) & \text{if } m < k \\ MissRate(k) & \text{otherwise} \end{cases} \quad (8)$$

The inverse of instantaneous miss rate is the average number of memory references required to arrive at the next miss. This information is used to calculate the number of references required to have $M$ misses (Equation 9).

$$Ref(M, k) = \sum_{m=1}^{M} 1/InstantRate(m, k) \quad (9)$$

We are now ready for the calculation of $exec^C$ and $CRT$. Considering task $\tau$ that has total number of memory references $R$, the task's $exec^C$ is calculated according to Equation 10. Note that, in this case we can directly use Equation 5 to approximate cache miss rate since by definition $exec^C$ is the execution time of $\tau$ when running non-preemptively. Hence, $exec^C$ can be computed as it follows:

$$\begin{aligned} exec^C(k) = \; & (1 - Missrate(k)) * R * HitDelay \\ & + Missrate(k) * R * MissDelay, \end{aligned} \quad (10)$$

where $HitDelay$ and $MissDelay$ are the execution times of an instruction when a cache hit or cache miss occur, respectively. The values of these constants depend on the adopted platform. Assume now that $\tau$ runs on a multi-tasking system scheduled by cyclic executive with a time slot of size $s$ (in general, $exec^C$ is longer than $s$): since no assumption is made on the memory access pattern of other tasks, in the worst case $\tau$ has to warm-up the cache again at each resumption. In other words, in order to calculate the number of memory references ($Ref$) taken place in each time slot $s$, Equation 9 must be used and the value of $M$ is such that the induced execution time in that time slot $((Ref(M, k) - M) * HitDelay + M * MissDelay)$ is equal to $s$. Note that in a multi-tasking system, the number of memory references (i.e. instructions) executed in a time slot is less than what can be executed within the same time slot in a non-preemptive or single-task system due to inter-task cache interference problem. Therefore, task $\tau$ would take more than $exec^C$ time to complete its total number of references. By definition, that additional time is captured by $CRT$.

In summary, to generate simulated input data, we generate a set of task-dependent parameters for each task: $A, \theta, k^0$, the size of scheduling time slot $s$, and the total number of task's memory references. $exec^C$ and $CRT$ of each task are then calculated accordingly using Equation 9 and 10. We emphasize that the generated data is only for the purpose of evaluating the performance of Algorithm 1; they do not replace techniques that estimate $exec^C$ and $CRT$. The proposed heuristic can be applied to any practical system whenever information of $exec^C$ and $CRT$ is available by any means.

### 6.2. Case Study

In this section, an experimental avionic system with eight tasks is used to verify the effectiveness of the proposed approach. The system parameters are the same as those in Table 1. The scheduler runs with a major cycle of six time slots: five tasks have their own time slot and three others share one time slot. The size of each time slot is shown in column 2 of Table 4. Tasks' parameters, including number of memory references, memory usage and time slot allocation are shown in Table 3. All tasks have the same period of $16.67ms$. The number of memory hits and misses were measured as a function of the cache size for each task. The traces were then used to find $exec^C$ and $CRT$ functions. $exec^C$ of tasks 1 to 4 are plotted in Figure 1. Al-

| Task | Number of references | Memory usage (KB) | Time slot | Cache size (KB) |
|---|---|---|---|---|
| 1 | 61560 | 8000 | 4 | 508 |
| 2 | 259023 | 33000 | 5 | 644 |
| 3 | 76364 | 668 | 1 | 104 |
| 4 | 90867 | 9000 | 3 | 416 |
| 5 | 32544 | 280 | 2 | 28 |
| 6 | 6116 | 140 | 1 | 60 |
| 7 | 41124 | 28 | 1 | 8 |
| 8 | 217675 | 230 | 6 | 216 |

Table 3: Task parameters

| Time slot | Size (ms) | Baseline: slot utilization ($U_{share}$) | Heuristic: slot utilization ($U^h_{wc}$) |
|---|---|---|---|
| 1 | 2.3 | 103.8 | 70.4 |
| 2 | 1.2 | 45.9 | 35.8 |
| 3 | 3.2 | 48.0 | 42.2 |
| 4 | 1.9 | 54.8 | 48.9 |
| 5 | 4.4 | 99.5 | 80.5 |
| 6 | 3.67 | 100.2 | 77.4 |

Table 4: Task worst-case utilization

though some tasks may use a large amount of memory, the range of cache sizes at which its $exec^C$ function differential is significant may be smaller. This is because, in most cases, the worst case execution time path accesses only a small part of all the memory allocation. For example, memory size of task 4 is about 9000KB but its $exec^C$ is subject to big variations only at cache sizes smaller than 512KB. In other words, it is possible to assign to a task an amount of cache smaller than its memory allocation without increasing its execution time much. This suggests that cache partitioning can still be useful in practice even with big memory-footprint applications.

To calculate $CRT$ function, we use the method discussed in Section 6.1. In addition, the task's parameters, i.e. $A, \theta, k^0$, are given by fitting $Missrate(k)$ (Equation 5) into the measured trace. The correctness of this model is verified by the fact that $CRT$ is always smaller than 30% of $exec^C$ (see Section 1 for more details). In this case study, the algorithm outputs a partitioned configuration where all tasks use a private cache. Column 5 of Table 3 reports the resulting size of each cache partition. As expected, in many cases the partition size is much smaller than task's memory size.

Table 4 reports time slot's utilization $U_{share}$ for the baseline configuration (that uses only a shared cache) and the improved time slot's utilization (by using the proposed heuristic) on columns 3 and 4, respectively. The utilization of a time slot is the percentage of slot duration used by the task(s) assigned to that slot; tasks running in a time slot are not schedulable if the slot utilization is greater than $100\%$. In this case study, slots 1 and 6 are not schedulable under the baseline configuration but they are under the

| $A$ (KB) | $\theta$ | $k^0$ (KB) | number of memory references |
|---|---|---|---|
| $[1, 10]$ | $[1.5, 3]$ | $[A_1, 1024]$ | $[10^3, 10^6]$ |

Table 5: Simulation parameters

partitioned configuration. The baseline utilization $U_{share}$, the partitioned one $U^h_{wc}$, and the utilization bound $U_b$ are $79.7\%$, $64.2\%$, and $61.3\%$, respectively. The utilization gain is $15.4\%$ while $U^h_{wc}$ is only $2.9\%$ greater than $U_b$.

### 6.3. Evaluation with Simulated Systems

The same system parameters shown in Table 1 are used for simulations. Tasks' parameters are randomly chosen with uniform distribution within the intervals presented in Table 5. The range of $A$ and $\theta$ is selected based on values given by [14] which also fit well with those used in the case study. All experiments use the range of $k^0$ shown in Table 5 except where noted otherwise. $A_1$ is calculated using Equation 6. Each task has utilization smaller than $100\%$ when it runs with the baseline configuration. The following simulations describe how different factors (i.e., the total cache size, the number of tasks, and the size of time slots) affect the performance of the proposed algorithm. The performance measures are the average utilization gain defined as $U_{share} - U^h_{wc}$ and the average value of $U_b - U^h_{wc}$. For comparison purposes, we also estimate the worst case utilization of proportional cache partitioned systems ($U_{prop}$). In such a system, a task is assigned a private cache partition of size proportional to its memory footprint, i.e. $k_i = \frac{k_i^0}{\sum_1^N k_j^0} K^{size}$. The average value of $U_{prop} - U^h_{wc}$ is reported. Simulated system parameters are as follows except where stated otherwise: $N = 10$, $K^{size} = 2MB$, and time slot range from $1$ to $3ms$. All results are the average over the outputs of 30 different task sets.

**Effect of the total cache size:** in this experiment, we measured systems having cache sizes of 512KB, 1MB, and 2MB. The performance measures plotted in Figure 2(a) show that the heuristic performs very well especially when the total cache size is large, i.e. 2MB: the gain in utilization is about $15\%$ and the difference between the heuristic utilization and the bound is less than $2\%$.

**Effect of the number of tasks:** in this experiment, the number of simulated tasks is 10, 20, 30, and 40. Results are plotted in Figure 2(b). Notice that two phenomena occur when the number of tasks increases: 1) the gain in utilization $U_{share} - U^h_{wc}$ is smaller since more tasks are forced to use the shared partition, 2) the poor performance of proportional cache partitioning is more prominent since there are more tasks running on smaller private cache partitions.

**Effect of the size of time slots:** in this experiment, tasks' time slot sizes are randomly chosen with uniform distribution within the following three ranges: $1 - 3ms$, $1 - 5ms$, $1 - 10ms$. Intuitively, when the time slot size increases, the

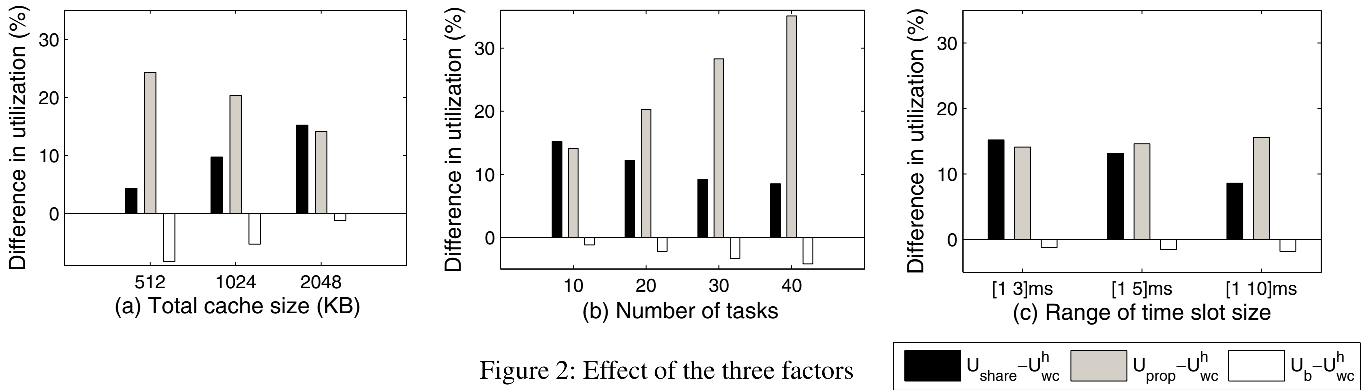


Figure 2: Effect of the three factors

effect of cache-related preemption delay is reduced since tasks experience less preemptions during each execution. Consequently, the utilization gain due to cache partitioning is lower. This behavior is shown in Figure 2(c): as expected, the utilization gain $U_{share} - U^h_{wc}$ is reduced to $8\%$ at the largest range whereas the gap between $U^h_{wc}$ and $U_b$ is almost constant. Notice that time slot variation has no effect on $U_{prop} - U^h_{wc}$.

## 7. Conclusion

This research has shown that cache partitioning can be used to improve system schedulability. This is because in real-time systems a schedulability test has to always assume worst-case execution times and worst-case task interference. Cache partitioning helps to reduce the interference thus being able to improve system schedulability. As future work, we would like to investigate more on the cache partitioning problem by taking into account other factors that can affect system schedulability and system safety such as L1 cache and I/O traffic.

## References


[1] S. Dropsho. Comparing caching techniques for multitasking real-time systems. Technical Report UM-CS-1997-065, Amherst, MA, USA, 1997.

[2] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization. WWC-4. IEEE International Workshop*, 2001.

[3] D. B. Kirk and J. K. Strosnider. Smart (strategic memory allocation for real-time) cache design using the mips r3000. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, 1990.

[4] C. G. Lee, K. L., J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *Software Engineering*, 27(9):805–826, 2001.

[5] J. Liedtke, H. Hartig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, 1997.

[6] J. W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[7] F. Mueller. Compiler support for software-based cache partitioning. In *Proceedings of the ACM Workshop on Languages, Compilers, and Tools for Real-Time Systems*, 1995.

[8] S. Oikawa and R. Rajkumar. Linux/rk: A portable resource kernel in linux. In *Proceedings of the 19th IEEE Real-Time Systems Sumposium*, 1998.

[9] R. Pellizzoni and M. Caccamo. Toward the predictable integration of real-time cots based systems. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, 2007.

[10] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. Practical solutions for QoS-based resource allocation. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.

[11] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, 2005.

[12] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[13] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Muller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem - overview of methods and survey of tools. Technical report, 2007.

[14] J. L. Wolf, H. S. Stone, and D. Thiébaut. Synthetic traces for trace-driven simulation of cache memories. *IEEE Trans. Comput.*, 41(4):388–410, 1992.

[15] A. Wolfe. Software-based cache partitioning for real-time applications. *J. Comput. Softw. Eng.*, 2(3):315–327, 1994.

[16] M. Zbigniew and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, December 2004.