Scratchpad Allocation for Concurrent Embedded Software

Vivy Suhendra, Abhik Roychoudhury, and Tulika Mitra
Department of Computer Science, National University of Singapore
{vivy, abhik, tulika}@comp.nus.edu.sg

ABSTRACT

Software-controlled scratchpad memory is increasingly employed in embedded systems as it offers better timing predictability compared to caches. Previous scratchpad allocation algorithms typically consider single process applications. But embedded applications are mostly multi-tasking with real-time constraints, where the scratchpad memory space has to be shared among interacting processes that may preempt each other. In this paper, we develop a novel dynamic scratchpad allocation technique that takes these process interferences into account to improve the performance and predictability of the memory system. We model the application as a Message Sequence Chart (MSC) to best capture the inter-process interactions. Our goal is to optimize the worst-case response time (WCRT) of the application through runtime reloading of the scratchpad memory content at appropriate execution points. We propose an iterative allocation algorithm that consists of two critical steps: (1) analyze the MSC along with the existing allocation to determine potential interference patterns, and (2) exploit this interference information to tune the scratchpad reloading points and content so as to best improve the WCRT. We evaluate our memory allocation scheme on a real-world embedded application controlling an Unmanned Aerial Vehicle (UAV).

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems

General Terms

Design, Performance

Keywords

Scratchpad memory, WCET, Message Sequence Chart

1. INTRODUCTION

Scratchpad memory is a software-managed on-chip memory that has been widely accepted as an alternative to caches in real-time embedded systems, as it offers better timing predictability compared to caches. The compiler and/or the programmer explicitly controls the allocation of instructions and data to the scratchpad memory. Thus the latency of each memory access is completely predictable. However, this predictability is achieved at the cost of compiler support for content selection and runtime management.

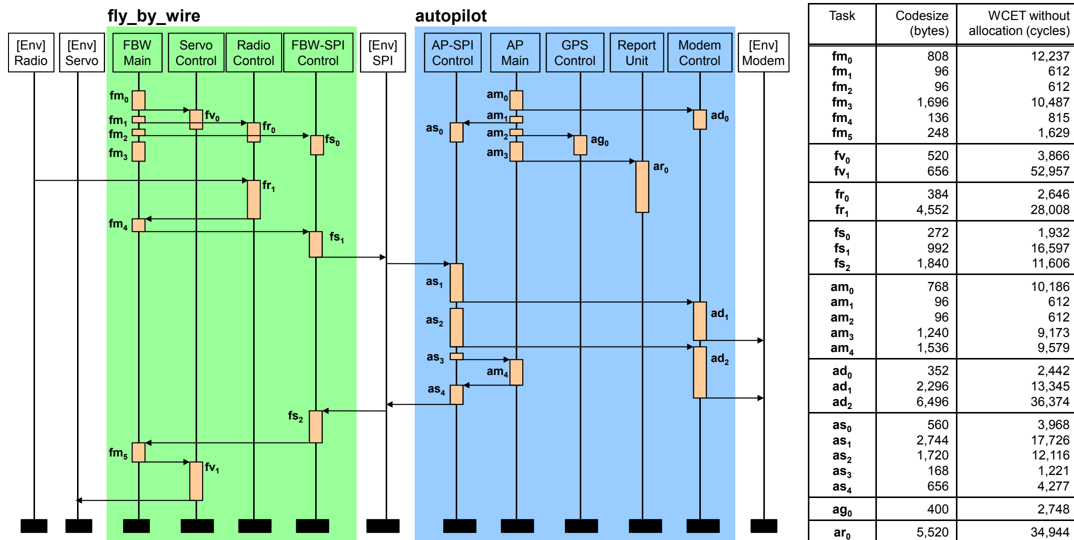
In this paper, we address the problem of *scratchpad memory allocation for concurrent embedded software (with real-time constraints) running on uniprocessor or multiprocessor platforms.* Our objective is to reduce the worst-case response time (WCRT) of the entire application. Our problem setting is representative of the current generation embedded applications (e.g., in automotive and avionics domain) that are inherently concurrent in nature and, at the same time, are expected to satisfy strict timing constraints. The combination of concurrency and real-time constraints introduces significant challenges to the allocation problem.

Given a sequential application, the problem of content selection for scratchpad memory has been studied extensively [9, 10, 12, 14]. However, these techniques are not directly applicable to concurrent applications with multiple interacting processes. Figure 1 shows a Message Sequence Chart (MSC) model [1, 3] depicting the interaction among the processes in an embedded application. We use MSC model as it provides a visual but formal mechanism to capture the inter-process interactions. Visually, an MSC consists of a number of interacting *processes* each shown as a vertical line. Time flows from top to bottom along each process. A process in turn consists of one or more *tasks* represented as blocks along the vertical line. Message communications between the processes are shown as horizontal or downward sloping arrows. Semantically, an MSC denotes a labeled partial order of tasks. This partial order is the transitive closure of (a) the total order of the tasks in each process, and (b) the ordering imposed by message communications — a message is received after it is sent.

A naive allocation strategy can be to share the scratchpad memory among all the tasks of all the processes throughout the lifetime of the application. Allocation algorithms proposed in the literature for sequential applications can be easily adapted to support this strategy. However, this strategy is clearly sub-optimal, as a task executes for only a fraction of the application's lifetime yet occupies its share of the memory space for the entire lifetime of the application. Instead, two tasks with disjoint lifetimes (e.g., tasks fm_0 and fm_4 in Figure 1) should be able to use the same memory space through time multiplexing. This is known as *dynamic scratchpad allocation* or *scratchpad overlay* where the scratchpad memory content can be replaced and reloaded at runtime.

As timing predictability is the main motivation behind the choice of scratchpad memory over caches, it should be maintained even in the presence of scratchpad overlay. This implies that in a concurrent system (e.g., as shown in Figure 1), *two tasks t_1 and t_2 should be mapped to the same memory space only if we can guarantee*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-470-6/08/10 ...$5.00.



Task	Codesize (bytes)	WCET without allocation (cycles)
fm_0	808	12,237
fm_1	96	612
fm_2	96	612
fm_3	1,696	10,487
fm_4	136	815
fm_5	248	1,629
fv_0	520	3,866
fv_1	656	52,957
fr_0	384	2,646
fr_1	4,552	28,008
fs_0	272	1,932
fs_1	992	16,597
fs_2	1,840	11,606
am_0	768	10,186
am_1	96	612
am_2	96	612
am_3	1,240	9,173
am_4	1,536	9,579
ad_0	352	2,442
ad_1	2,296	13,345
ad_2	6,496	36,374
as_0	560	3,968
as_1	2,744	17,726
as_2	1,720	12,116
as_3	168	1,221
as_4	656	4,277
ag_0	400	2,748
ar_0	5,520	34,944

Figure 1: Message Sequence Chart (MSC) model of the adapted UAV control application

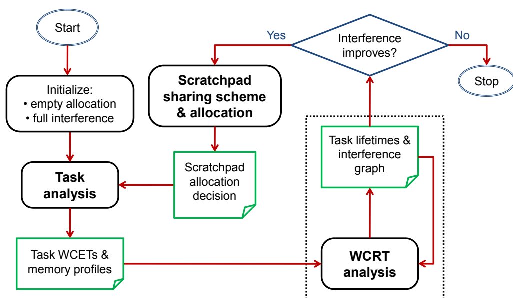


Figure 2: Workflow of WCRT-optimizing scratchpad allocation

that t_1 and t_2 have disjoint lifetimes. Otherwise, t_1 and t_2 may preempt each other, leading to scratchpad reloading delay at every preemption point. We can trivially identify certain tasks with disjoint lifetimes; for example, based on the partial order of an MSC, if task t_1 "happens before" task t_2, clearly t_1 and t_2 have disjoint lifetimes. However, there may exist many pairs of tasks that are incomparable as per MSC partial order but still have disjoint lifetimes. Moreover, as scratchpad allocation reduces execution times of the individual tasks, the lifetimes and the interference among the tasks may change. Therefore, an effective scratchpad allocation scheme attempting to minimize the WCRT of the application should consider the process interference as well as the impact of allocation on process interference.

In this paper, we propose an *iterative allocation algorithm* (Figure 2) consisting of two critical steps: (1) analyze the MSC along with existing allocation to estimate the lifetimes of tasks and hence the non-interfering tasks, and (2) exploit this interference information to tune scratchpad reloading points and content so as to best improve the WCRT. The iterative nature of our algorithm enables us to handle the mutual dependence between allocation and process interaction. As we ensure a monotonic reduction of WCRT in every iteration, our allocation algorithm is guaranteed to terminate.

Concretely, our main contribution in this paper is a novel dynamic scratchpad allocation technique that takes process interferences into account to improve the performance and predictability of the memory system for concurrent embedded software. Our case study with a complex embedded application controlling an Unmanned Aerial Vehicle (UAV) reveals that we can achieve significant performance improvement through appropriate content selection and runtime management of the scratchpad memory.

Related Work. The problem of content selection for scratchpad memory has been studied extensively for sequential applications. Most of these works [10, 14] aim to minimize average-case execution time or energy consumption through scratchpad allocation. Scratchpad content selection for minimizing the worst-case execution time (WCET) has been addressed as well [9, 12]. However, these techniques are not applicable when the scratchpad space needs to be shared among multiple interacting processes.

The work by Verma et al. [15] presents a set of scratchpad sharing strategies among the processes for energy consumption minimization. However [15] simply assumes a statically defined schedule whereas we consider priority driven preemptive scheduling. Moreover, the scratchpad sharing decisions in [15] are not based on interactions and interference among the processes, which are critical in our case to provide real-time guarantees.

Scratchpad sharing among different processing elements (PEs) in a multiprocessor system-on-a-chip has also been investigated. Here the focus is more on mapping of codes/data to the private scratchpad memories of the PEs so as to maximize the benefit from the scratchpad allocation [4]. Other techniques include exploration of scratchpad hierarchy [2, 13] and runtime customization of scratchpad sharing and allocation among the PEs [5].

Finally, this work complements the research on cache-related preempted delay (CRPD) [6, 7], which provides timing guarantee for concurrent software by analyzing interferences in cache memory due to process interactions. Our work, on the other hand, eliminates interference in memory through scratchpad allocation.

2. PROBLEM FORMULATION

The input to our problem is in the form of Message Sequence Chart (MSC) [1, 3] that captures process interactions corresponding to a concurrent embedded application. We assume a *preemptive, multi-tasking* execution model. The application is periodic in nature. The MSC represents interactions within one such invocation where all processes involved should adhere to a common period and deadline. The underlying hardware platform contains one

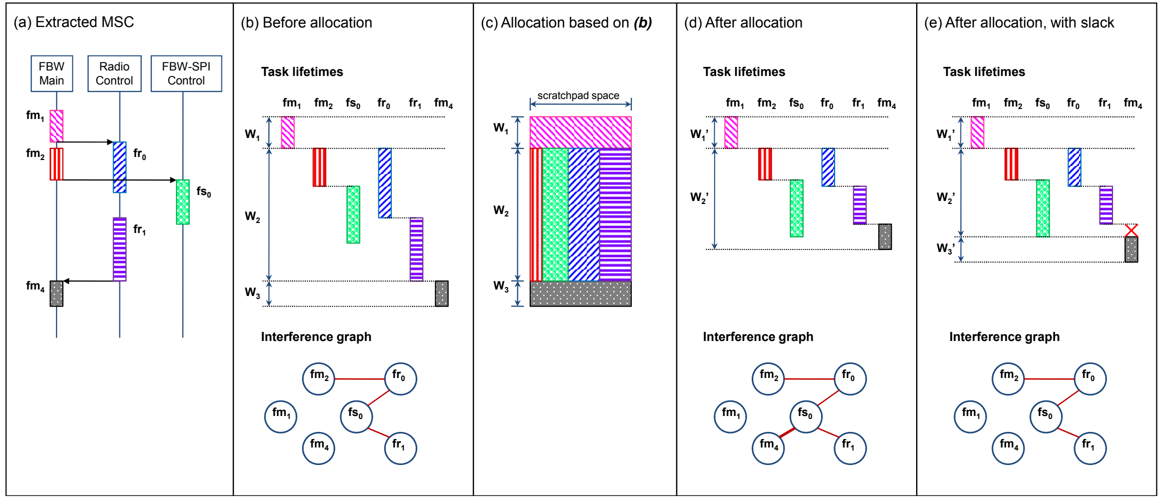


Figure 3: Sample MSC, task lifetimes and interference graphs before and after scratchpad allocation

or more processing elements (PEs), each associated with a private scratchpad memory. A process (a vertical line in the MSC) typically corresponds to a specific functionality. It is thus natural to assign all the tasks in a process to one PE. The order in which the tasks appear on the process lifeline reflects their order of execution on the PE. In this paper, we assume zero communication delay between processes. However, our analysis can be easily adapted to include non-zero communication delays.

Each process is assigned a unique static priority. The priority of a task is equal to the priority of the process it belongs to. A task t_1 in a process P may get preempted by a task t_2 from a higher priority process P'. The assignment of static priorities to processes and the mapping of processes to PEs are inputs to our framework. Note that statically assigned priorities do not guarantee a fixed execution schedule at runtime. The preemptions and execution time variations depending on input lead to varying completion times of a task. This, in turn, gives rise to different execution schedules.

We now formalize the problem definition. Let $t_1, \ldots, t_N$ denote the tasks belonging to all the processes in the application. Each task t_i $(1 \leq i \leq N)$ is associated with a period p_i, a static priority r_i (the range of r_i is $[1, R]$ with 1 being the highest priority), and mapping to a PE PE_i (the range of PE_i is $[1, Q]$ where Q is the number of PEs in the system). As mentioned before, all the tasks belonging to a process have the same priority and are mapped to the same PE. Further, let c_i denote the *uninterrupted* worst-case execution time (WCET) of the task t_i running on PE_i in isolation. The estimation of c_i does not assume any scratchpad allocation, i.e., all the accesses incur the main memory latency.

Let $\mathcal{S}$ be a particular scratchpad allocation for the application. In this work, we consider allocating program codes into the scratchpad. The method applies similarly to data allocation. $\mathcal{S}$ consists of two components: (1) the amount of scratchpad space $space(t_i)$ allocated to each task, and (2) the allocation of $space(t_i)$ among the code blocks of t_i. Note that as we allow scratchpad overlay, the same scratchpad memory space can be allocated to two or more tasks as long as they have disjoint lifetimes. Let Mem_i denote the set of all code blocks of t_i available for allocation. Given $space(t_i)$, the allocation $Alloc(t_i) \subseteq Mem_i$ is the set of most profitable code blocks from t_i to fit the capacity. Finally, WCET of t_i as a result of allocation $\mathcal{S}$ is denoted as $wcet(t_i, \mathcal{S})$.

Given an allocation $\mathcal{S}$ and the corresponding WCET of the tasks, we can estimate the *lifetime* of each task, defined as the interval between the lower bound on the start time $Start(t_i, \mathcal{S})$ and the upper bound on the finish time $Finish(t_i, \mathcal{S})$ of the task. This estimation should take into account the dependencies among the tasks (total order among the tasks within a process and ordering imposed by message communication) as well as preemptions. The WCRT of the whole application is now given by

$$WCRT = max_{1 \leq i \leq N}\ Finish(t_i, \mathcal{S}) - min_{1 \leq i \leq N}\ Start(t_i, \mathcal{S}) \quad (1)$$

Our goal is to construct the scratchpad allocation $\mathcal{S}$ that minimizes the WCRT of the application.

3. METHOD OVERVIEW

Our proposed method is an iterative scheme (Figure 2), which we will elaborate below. Figure 3(a) shows an MSC extracted from our case study for the purpose of illustration.

Task Analysis. We analyze each task to determine its WCET with a given scratchpad allocation (initially empty), along with the area and the gain of allocating each of its code blocks. The WCET will serve as input to the WCRT analysis, while the memory profile will be used to choose the scratchpad content for the task at the allocation step. We handle code allocation in this paper, for which possible choices of granularity are a basic block (which we use here) or a function. The gain of allocating a block of code, in terms of execution time saving, is the execution frequency of the block multiplied by the reduction in latency required to fetch the block from scratchpad instead of from main memory. As we are considering systems with real-time constraints, the execution frequencies correspond to the worst-case execution path, which in turn is obtained via static analysis [11]. Worst-case execution path may shift as allocation decisions change; thus task profiles should be updated following each change in the course of the iterative improvement. We adapt our previous work on WCET-centric allocation of data for a single task to scratchpad memory [12] for this purpose.

WCRT Analysis. The WCRT of a task t_i is a function of its WCET value and the delay caused by higher priority tasks whose lifetimes overlap with that of t_i. A fixed-point iteration computes this value by finding the root to the equation

$$x = g(x) = wcet(t_i, \mathcal{S}) + \sum_{t_j \in intf(t_i)} wcet(t_j, \mathcal{S}) \times \lceil x/p_j \rceil$$

$$intf(t_i) = \{ \, t_j \mid r_j < r_i \;\; \text{AND} \tag{2}$$
$$[Start(t_j, \mathcal{S}), Finish(t_j, \mathcal{S})] \cap [Start(t_i, \mathcal{S}), Finish(t_i, \mathcal{S})] \neq \emptyset \, \}$$

If we denote this WCRT value as $wcrt(t_i, \mathcal{S})$, we have for each task t_i: $Finish(t_i, \mathcal{S}) = Start(t_i, \mathcal{S}) + wcrt(t_i, \mathcal{S})$. Further, the partial ordering of tasks in the MSC imposes the constraint that a task t_i can start execution only after all its predecessors have completed execution. In other words, $Start(t_i, \mathcal{S}) \geq Finish(u, \mathcal{S})$ for all tasks u preceding t_i in the partial order of the MSC. Observing these rules, the WCRT analysis computes the lifetimes of all tasks in all processes. After the analysis, we construct the *task interference graph* for the purpose of scratchpad allocation. Figure 3(b) shows the task lifetimes computed by the WCRT analysis and the constructed interference graph given the MSC in (a). An edge between two nodes in the interference graph implies overlapping lifetimes of the two tasks represented by the nodes.

Scratchpad Sharing Scheme & Allocation. Based on the interference pattern resulting from the WCRT analysis, we can construct a scratchpad sharing scheme among tasks on the same PE. One possible scheme is illustrated in Figure 3(c), which shows the space sharing among tasks as well as the dynamic overlay over time. The schemes will be elaborated in the next section. The sharing scheme determines the space allocated to each task, and the memory profiles obtained in the first step are used to select the most beneficial scratchpad content. The selection strategy is based on our previous work that aims to minimize the task WCET, taking into account the possible shift in worst-case execution path [12]. After allocation is performed, the task analysis is re-applied to find the new WCET.

Post-Allocation Analysis. Given updated task WCETs after allocation, the WCRT analysis is performed once again to compute updated task lifetimes. There is an important constraint to be observed in the WCRT analysis when the allocation decision has been made. The new WCET values have been computed based on the current scratchpad allocation, which is in turn decided based on the task interference pattern resulting from the previous analysis. In particular, scratchpad overlays have been decided among tasks determined to be interference-free. Therefore, these values are only valid for the same interference pattern, or for patterns with less interference. To understand this, suppose the interference graph in Figure 3(b) leads to the allocation decision in (c). The reduction in WCET due to the allocation in turn reduces task response times and changes task lifetimes to the one shown in (d). However, this computation of lifetimes is incorrect, because it assumes the WCET value of fs_0 given that it can occupy the assigned scratchpad space throughout its execution. If fm_4 is allowed to start earlier, right after its predecessor fr_1 as shown in (d), it may in fact preempt fs_0, flushing the scratchpad content of fs_0 and causing additional delay for reload when fs_0 resumes. Indeed, we see that the interference graph in (d) has an added edge from fm_4 to fs_0.

To avoid this unsafe assumption, we need to maintain that tasks known not to interfere when allocation decision is made will not become interfering in the updated lifetimes. This is accomplished by introducing *slack* that forces the latter task to "wait out" the conflicting time window. The adapted WCRT analysis consults existing interference graph and adjusts $Start(fm_4, \mathcal{S})$ such that $Start(fm_4, \mathcal{S}) \geq Finish(fs_0, \mathcal{S})$. Figure 3(e) shows the adjusted schedule, which maintains the same interference graph as (a) by forcing fm_4 to start after fs_0 has completed.

With a more sophisticated sharing/allocation scheme and schedule adjustment as we will introduce next, we can sometimes remove existing task interferences without adding interference elsewhere. When this happens, we iterate over the allocation and analysis steps to enhance current decision, until no more improvement can be made (Figure 2). As task interferences are enforced to be non-increasing, the iteration is guaranteed to terminate.

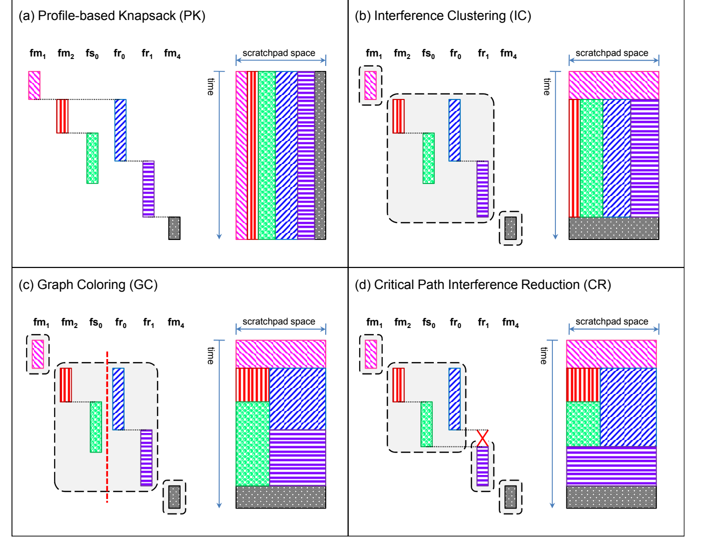


Figure 4: PK, IC, GC, and CR allocation schemes

4. ALLOCATION METHODS

This section describes the scratchpad allocation routine, which is the focus of our paper.

As only one task will be running on the PE at any given time, we can actually utilize the whole scratchpad space for the single executing task. The concern arises when a task is preempted, as flushing the scratchpad content will cause additional reloading delay when the task resumes. In that case, it may be beneficial to reserve a portion of the scratchpad for each of the tasks (*space-sharing*), thus avoiding the need to flush and reload the scratchpad memory at each preemption/resume. On the other hand, two tasks guaranteed to never interfere with each other can share the same space via overlay (*time-sharing*). In Figure 3(b), tasks fm_2 and fr_0 are space-sharing tasks, while task fm_1 in time window W_1 has a time-sharing relationship with all tasks in time window W_2.

The various schemes are illustrated in Figure 4. The left side of each picture shows task lifetimes as determined by the WCRT analysis, and the right side sketches the state of the scratchpad memory due to the different allocation schemes. For the purpose of comparing the scratchpad state, the lifetime of each task has been drawn with the same height across the different schemes (with the exception of *CR*). In reality, the heights (representing the length of task runtime) will vary due to the different allocation decisions.

Profile-based Knapsack (PK). As the baseline method, we consider a profile-based static allocation method. In this scheme, all tasks executing on the same PE will share the PE's scratchpad space throughout application lifetime. The allocation decision does not consider the possible interferences (or lack thereof) within the PE.

Partitioning and scratchpad allocation for each PE q can be simultaneously optimized via an Integer Linear Programming (ILP) formulation. The objective is to minimize the combined WCET weighted by task periods, defined as

$$\sum_{t_i : PE_i = q} wcet(t_i, \mathcal{S}) / p_i$$

$$wcet(t_i, \mathcal{S}) = \; c_i - \sum_{b \in Alloc(t_i)} freq_b \times area_b \times \Delta$$

Recall that c_i is the running time of t_i when all code blocks are fetched from the main memory, and $Alloc(t_i) \subseteq Mem_i$ is the selected set of code blocks of t_i in scratchpad allocation $\mathcal{S}$. $freq_b$ and $area_b$ are respectively the execution frequency in the worst-case path and the area occupied by block b. The term Δ defines the savings in execution time per unit area due to the scratchpad allocation. Given the scratchpad size of cap_q attached to PE q, the capacity constraint is expressed as

$$\left(\sum_{t_i:PE_i=q} \sum_{b \in Alloc(t_i)} area_b \right) \leq cap_q$$

For allocation of program code into the scratchpad, an additional constraint is needed to maintain correct control flow [10]. If two sequential basic blocks are allocated in different memory areas (i.e. one in scratchpad and one in main memory), then a jump instruction should be inserted at the end of the earlier block.

Figure 4(a) shows the partitioned scratchpad by *PK*. As the allocation decision does not depend on task interference, *PK* will only execute for one round; no iterative improvement can be made.

Interference Clustering (IC). In this second method, we use task lifetimes determined by the WCRT analysis to form *interference clusters*. Tasks whose lifetimes overlap at some point are grouped into the same cluster. They will share the scratchpad for the entire duration of the common time window, from the earliest start time to the latest finish time among all tasks in the cluster. The same partitioning/allocation routine used in *PK* is employed among all tasks in the same cluster. The left part of Figure 4(b) shows the clustering decision for the given task schedule. fm_1 as well as fm_4 have been identified as having no interference from any other task. Each of them is placed in a singleton cluster and enjoys the whole scratchpad space during its lifetime.

Graph Coloring (GC). The *IC* method is prone to produce large clusters due to transitivity. In Figure 4(b), even though fm_2 and fs_0 do not interfere with each other, their independent interferences with fr_0 end up placing them in the same cluster. Because of this, simply clustering the tasks will likely result in inefficient decisions. The third method attempts to enhance the allocation within the clusters formed by the *IC* method by making use of the task-to-task interference relations captured in the interference graph. If we apply *graph coloring* to this graph, the resulting colors will give us groups of tasks that do not interfere with each other within the cluster. Tasks assigned to the same color have disjoint lifetimes, therefore can reuse the same scratchpad space via further overlay.

Graph coloring using the minimum number of colors is known to be NP-Complete. We employ the Welsh-Powell algorithm [16], a heuristic method that assigns the first available color to a node, without restricting the number of colors to use. Given the interference graph, the algorithm can be outlined as follows.

1. Initialize all nodes to uncolored.
2. Traverse the nodes in *decreasing order of degree*, assigning color 1 to a node if it is uncolored and no adjacent node has been assigned color 1.
3. Repeat step 2 with colors 2, 3, etc. until no node is uncolored.

After we obtain the color assignment, we formulate the scratchpad partitioning/allocation with the refined constraint that a task t_i with assigned color k_i can occupy at most the space allocated for k_i, denoted by $area(k_i)$. The scratchpad space given to all K colors used for PE q add up to the total capacity cap_q, as expressed below.

$$\sum_{b \in Alloc(t_i)} area_b \leq area(k_i); \quad \sum_{k=1}^{K} area(k_i) \leq cap_q$$

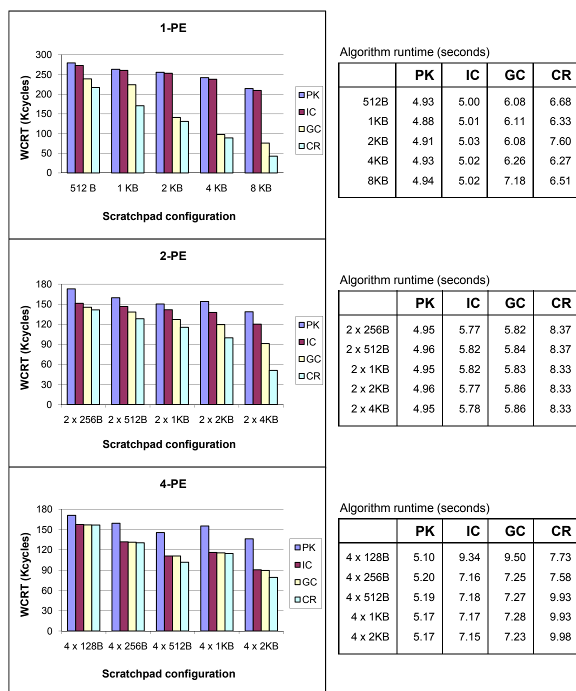
Figure 4(c) shows the further partitioning within the second cluster formed by *IC*. fm_2 and fs_0 have been assigned the same color, and allocated the same partition of the scratchpad to occupy at different time windows. The similar decision applies to fr_0 and fr_1. The partition will be reloaded with the relevant task content when execution transfers from one task to another.

```
1  repeat
2      CT := ∅;
3      foreach task t on the critical path of current schedule do
4          CT := CT ∪ { (t, u) | u ∈ intf(t) };
5          /* intf(t): the set of higher priority tasks whose lifetimes
           overlap with t's (Equation 2) */
6      if CT ≠ ∅ then
7          Find any (t_m, u_m) in CT such that
             wcet(u_m, S) ≥ wcet(u, S) for all pairs (_, u) in CT;
8          /* eliminate this interference by imposing slack */
9          Set constraint Start(t_m, S) ≥ Finish(u_m, S);
10         Run WCRT analysis to propagate lifetime shift;
   until CT = ∅;
```

Algorithm 1: The CR algorithm

Critical Path Interference Reduction (CR). While the above three schemes try to make the best out of the given interference pattern, the final method that we propose turns the focus to reducing the interference instead. This is motivated by the observation that allocation decisions are often compromised by heavy interference. When the analysis recognizes a potential preemption of one task by another, both tasks will have to do space-sharing; in addition, the lifetime window of the preempted task must make allowance for the time spent waiting for the preempting task to complete. In an extreme case, if a task t_1 is released right before a higher priority task t_2, it must wait for practically the entire execution duration of t_2. In this case, suppressing the release time of t_1 until t_2 completes can only be beneficial: t_1's waiting time is still the same, yet no preemption cost is incurred, and a better allocation decision can be made for both tasks. In general, this is a good strategy when the waiting time far outweighs the actual computation time of the task.

The method proceeds as shown in Algorithm 1. We first work on the schedule produced by the WCRT analysis to improve the interference pattern. In choosing which interference to eliminate, we naturally look at the critical path of the application, as determined by the WCRT analysis. We consider all interferences in which tasks on the critical path are preempted or have to wait for tasks with higher priority (line 4). From these, we choose the interference that occupies the longest time window, that is, one in which the higher-priority task has the longest WCET (line 7). We eliminate this interference by forcing a delayed start time for the affected task (line 9), then propagate the shift to all tasks by re-running the WCRT analysis. Certainly, new interferences are not allowed to arise in this step. From the new schedule, we again consider preemptions on the critical path, which may or may not have shifted. The elimination and re-analysis are iterated until no more interferences can be eliminated from the critical path. We then proceed to perform scratchpad partitioning/allocation as in the *GC* scheme on this improved interference graph. In Figure 4(d), the interference between fs_0 and fr_1 has been eliminated by letting fr_1 wait out the lifetime of fs_0 instead of starting immediately after the completion of its predecessor fr_0. This improvement frees fr_1 from all interference. It can now occupy the whole scratchpad memory throughout its lifetime.



Algorithm runtime (seconds)

	PK	IC	GC	CR
512B	4.93	5.00	6.08	6.68
1KB	4.88	5.01	6.11	6.33
2KB	4.91	5.03	6.08	7.60
4KB	4.93	5.02	6.26	6.27
8KB	4.94	5.02	7.18	6.51

Algorithm runtime (seconds)

	PK	IC	GC	CR
2 x 256B	4.95	5.77	5.82	8.37
2 x 512B	4.96	5.82	5.84	8.37
2 x 1KB	4.95	5.82	5.83	8.33
2 x 2KB	4.96	5.77	5.86	8.33
2 x 4KB	4.95	5.78	5.86	8.33

Algorithm runtime (seconds)

	PK	IC	GC	CR
4 x 128B	5.10	9.34	9.50	7.73
4 x 256B	5.20	7.16	7.25	7.58
4 x 512B	5.19	7.18	7.27	9.93
4 x 1KB	5.17	7.17	7.28	9.93
4 x 2KB	5.17	7.15	7.23	9.98

Figure 5: WCRT of the benchmark application after allocation by PK, IC, GC, and CR, along with algorithm runtime

5. EXPERIMENTS AND DISCUSSION

We use as case study the UAV control application from PapaBench [8], adapted into a distributed implementation. The controller consists of two main functional units (`fly_by_wire`, `autopilot`) and operates in one of two modes: manual and automated. Figure 1 shows the active processes in one of the scenarios under the manual mode. The original implementation as shown uses 2 PEs with a total of 5KB scratchpad memory space. For the purpose of observation, we vary the number of PEs from 1 to 4. In the 4-PE case, processes `FBW Main` and `Radio Control` are assigned to the 1st PE, `Servo Control` and `FBW-SPI Control` to the 2nd PE, `AP Main` and `AP-SPI Control` to the 3rd PE, and the remaining processes to the 4th PE. Total scratchpad size (for instructions) is varied from 512B to 8KB, distributed evenly among the PEs. The table in Figure 1 gives the codesizes and uninterrupted runtimes of each task, obtained via WCET analysis of the program code assuming all instructions are fetched from the main memory. We assume uniform execution time of 1 cycle per instruction, and that it takes 1 cycle to fetch a 16-byte block from the scratchpad, 100 cycles from the main memory.

Figure 5 shows the final WCRT of the application for various scratchpad configuration, after applying the four discussed schemes. The three charts correspond to the cases where the tasks are distributed on 1, 2, and 4 PEs. Obviously, with more PEs, less interference is observed among the tasks. On the other hand, it also means less scratchpad space per PE for the same total scratchpad size, which limits the maximum space utilizable by a task.

When only 1 PE is utilized, most tasks are interfering with each other. We can see a drastic WCRT improvement from 1-PE to 2-PE setting for all schemes, which confirms the observation that task interferences significantly influence application response time. With 1 PE, *IC* does not give significant improvement over the baseline *PK*, as the transitive interference places most tasks into the same space-sharing cluster. With more PEs, *IC* is able to perform better than *PK*. *GC* performs no worse than *IC* in all cases, as it has a more refined view of interference relation among individual tasks. *CR* in turn is never worse than *GC*. The respective improvements are particularly pronounced in the 1-PE setting where interference is heavy. With 4 PEs employed, task interference is reduced, and *GC* has only slight advantage over *IC*, as does *CR*. Comparing the same scheme for the same total scratchpad size over increasing number of PEs, the smaller scratchpad size per PE limits the area utilizable by each task, and gives less runtime improvement in some cases.

The proposed scheme *CR* gives the best WCRT improvement over all other schemes. This justifies the strategy of eliminating critical interferences via slack enforcement, whenever any additional delay that is incurred can be overshadowed by the gain through a better scratchpad sharing and allocation scheme. Finally, comparison of algorithm runtimes in the tables of Figure 5 shows that all schemes are reasonably efficient and no scalability issue is evident.

6. CONCLUDING REMARKS

In this paper, we have done a detailed study of scratchpad allocation schemes for concurrent embedded software running on single or multiple processing elements. The novelty of our work stems from taking into account both concurrency and real-time constraints in our scratchpad allocation. Our allocation schemes consider (i) communication or interaction among the threads or processes of the application, as well as (ii) interference among the threads or processes due to preemptive scheduling in the processing elements. As the interactions and interference among the processes can greatly affect the worst-case response time (WCRT) of a concurrent application, our scratchpad allocation methods achieve substantial reduction in WCRT as evidenced by our experiments.

7. ACKNOWLEDGMENTS

This work is partially supported by NUS research projects R252-000-292-112 and R252-000-321-112.

8. REFERENCES

[1] R. Alur and M. Yannakakis. Model checking message sequence charts. In *CONCUR*, 1999.

[2] I. Issenin, E. Brockmeyer, B. Durinck, and N. Dutt. Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In *DAC*, 2006.

[3] ITU-T. 120: Message sequence chart (MSC). *ITU-T, Geneva*, 1996.

[4] M. Kandemir. Data locality enhancement for CMPs. In *ICCAD*, 2007.

[5] M. Kandemir, O. Ozturk, and M. Karakoy. Dynamic on-chip memory management for chip multiprocessors. In *CASES*, 2004.

[6] C.-G. Lee, J. Hahn, Y.-Min Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE transactions on computers*, 47(6):700–713, 1998.

[7] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS*, 2003.

[8] F. Nemer, H. Cassé, P. Sainrat, J-P. Bahsoun, and M. De Michiel. PapaBench: A free real-time benchmark. In *WCET*, 2006.

[9] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *ECRTS*, 2006.

[10] S. Steinke, L. Wehmeyer, B. S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE*, 2002.

[11] V. Suhendra, T. Mitra, and A. Roychoudhury. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC*, 2006.

[12] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *RTSS*, 2005.

[13] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for MPSoC architectures. In *CASES*, 2006.

[14] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES*, 2003.

[15] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *3rd Workshop on Embedded Systems for Real-Time Multimedia*, 2005.

[16] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–87, 1967.