Scratchpad Allocation for Concurrent Embedded Software

VIVY SUHENDRA
Institute for Infocomm Research, Singapore
ABHIK ROYCHOUDHURY and TULIKA MITRA
National University of Singapore, Singapore

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 work, 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 present various alternative scratchpad allocation heuristics and evaluate their effectiveness in reducing the WCRT. The scheme is also extended to work on Message Sequence Graph models. We evaluate our memory allocation scheme on two real-world embedded applications controlling an Unmanned Aerial Vehicle (UAV) and an in-orbit monitoring instrument, respectively.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Memory Management*

General Terms: Design, Performance

Additional Key Words and Phrases: Scratchpad memory, Compiler controlled memories, Multi-core architectures, Worst-case response time, Message Sequence Chart, UML Sequence Diagram.

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 [Banakar et al. 2002]. The scratchpad memory is mapped into the address space of the processor, and is accessed whenever the address of a memory access falls within a pre-defined range (Figure 1). The compiler and/or the programmer

Authors' addresses: Vivy Suhendra, Institute for Infocomm Research, 1 Fusionopolis Way, Singapore 138632; e-mail: vsuhendra@i2r.a-star.edu.sg; Abhik Roychoudhury and Tulika Mitra, Department of Computer Science, National University of Singapore, 13 Computing Drive, Singapore 117417; e-mail: {abhik,tulika}@comp.nus.edu.sg

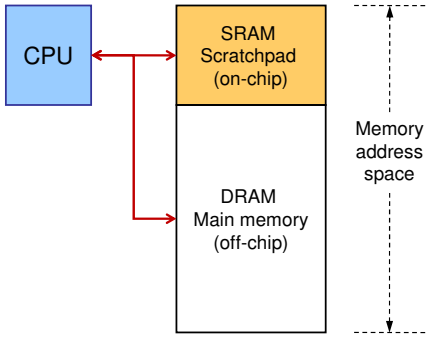


Fig. 1. Scratchpad memory

explicitly controls the allocation of instructions and data to the scratchpad memory. This operating principle makes the latency of each memory access, and thus program execution time, 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 memory allocation problem.

Given a sequential application, the problem of content selection for scratchpad memory has been studied extensively [Puaut 2006; Suhendra et al. 2005; Udayakumaran and Barua 2003]. However, these techniques are not directly applicable to concurrent applications with multiple interacting processes. Let us illustrate the issues involved with an example. Figure 2 shows a Message Sequence Chart (MSC) model [Alur and Yannakakis 1999; ITU-T 1996] 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 (1) the total order of the tasks in each process, and (2) the ordering imposed by message communications — a message is received after it is sent.

Consider the MSC example in Figure 3, which has been extracted from the earlier application. A naive scratchpad allocation strategy can be to share the scratchpad memory among all the tasks of all the processes throughout the lifetime of the application. This is illustrated in Figure 4a, where the distribution of scratchpad space over tasks is depicted in the horizontal direction, while the content reloading over time is depicted in order from top to bottom. The scratchpad is loaded with the designated memory contents once when

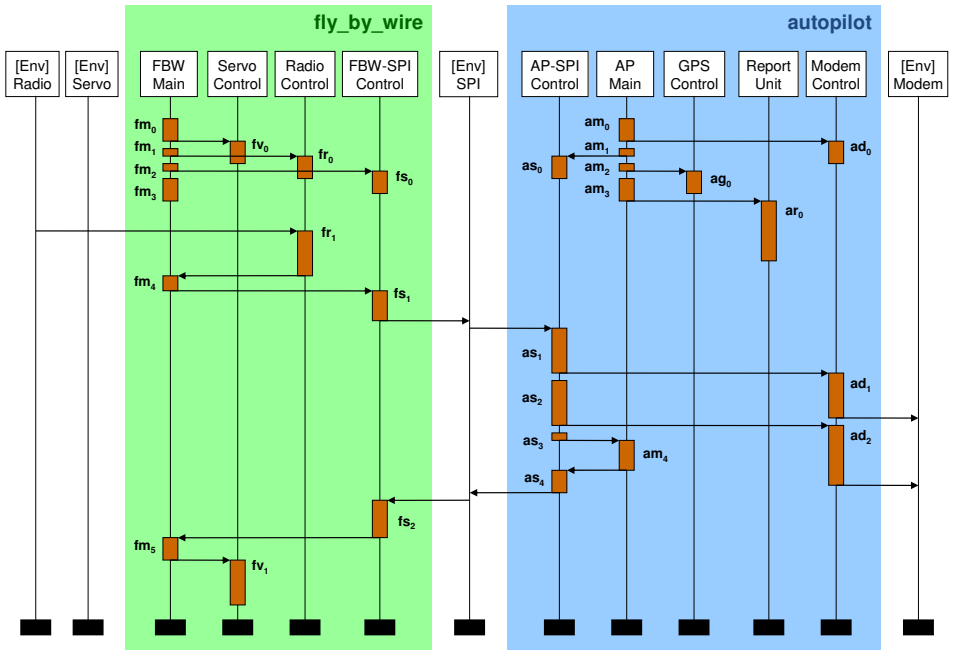


Fig. 2. Message Sequence Chart (MSC) model of the adapted UAV control application

the tasks start execution. 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 scratchpad space for the entire lifetime of the application. Instead, two tasks with disjoint lifetimes (e.g., tasks fm_1 and fm_2) should be able to use the same scratchpad space through time multiplexing. This is known as *dynamic scratchpad allocation* or *scratchpad overlay*, where the scratchpad content can be replaced and reloaded at runtime.

At the other extreme of this approach, we can also let each task occupy the whole scratchpad while it is executing (Figure 4b). When a task is preempted, its corresponding memory content in the scratchpad is replaced by that of the preempting task. Certainly, the loading and reloading of the scratchpad memory following each preemption and resuming of task execution also add to the total latency experienced by the system, and they have to be bounded to provide timing guarantee. In a system with a large number of tasks vying for CPU time, the chain of preemptions can get arbitrarily long and difficult to analyze.

The key to our proposed technique is finding a balance between these two extremes. Clearly, we would like to employ scratchpad overlay as much as possible for optimal gain in application response time. However, 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 2), *two tasks should be mapped to the same memory space only if we can guarantee that they have disjoint lifetimes*. Otherwise, the task with higher priority may preempt the other, leading to scratchpad reloading delay when the preempted task resumes.

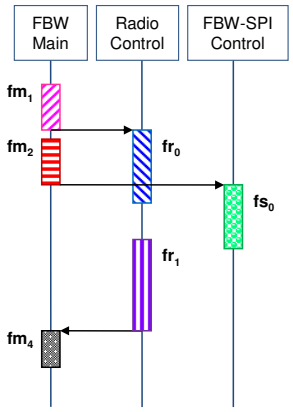


Fig. 3. A sample MSC extracted from the UAV control application case study

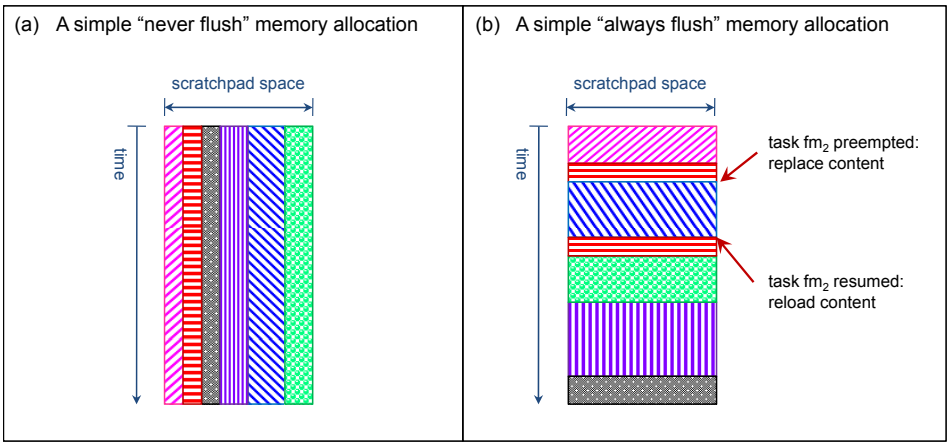


Fig. 4. Naive scratchpad allocation schemes for the model in Figure 3

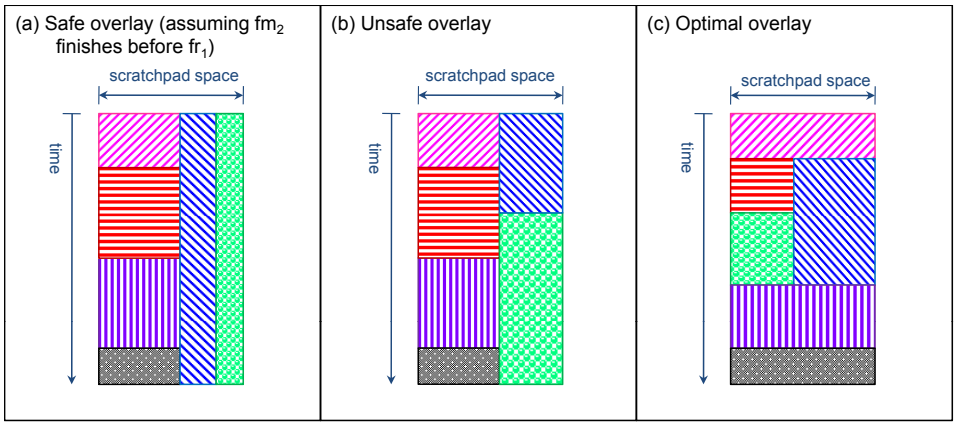


Fig. 5. Choices of scratchpad overlay schemes for the model in Figure 3: (a) safe, (b) unsafe, and (c) optimal

We can trivially identify certain tasks with disjoint lifetimes based on the partial order of an MSC; for example, as task fm_1 "happens before" task fm_2, clearly fm_1 and fm_2 have disjoint lifetimes (Figure 3). However, there may exist many pairs of tasks that are incomparable as per MSC partial order but still have disjoint lifetimes. For instance, after examining the execution schedule of tasks, the timing analysis may be able to determine that the execution time of fr_0, given any input, is always long enough to ensure that the succeeding task fr_1 can never be started before fm_2 finishes executing. In this case, we will be able to employ a better scratchpad overlay scheme as illustrated in Figure 5a, where fm_1, fm_2, fm_4 and fr_1 are mapped to the same scratchpad space which they can utilize during their respective lifetimes without disrupting one another. On the other hand, we should not arrive at a decision such as the one in Figure 5b, which lets fr_0 and fs_0 share the same scratchpad space without any guarantee that fs_0 will not preempt fr_0 in the actual execution. This situation will lead to unexpected reloading delays that will invalidate the WCRT estimation.

A scratchpad allocation scheme that achieves optimal WCRT reduction for this particular example may look like the layout in Figure 5c, which requires a deeper analysis of process interaction to arrive at. Moreover, as scratchpad allocation reduces execution times of the individual tasks, the lifetimes of the tasks and thus their interaction pattern may change. Therefore, an effective scratchpad allocation scheme attempting to minimize the WCRT of the application should consider process interferences as well as the impact of allocation on process interferences.

In this paper, we propose an *iterative scratchpad allocation algorithm* 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 scratchpad allocation and process interaction. In addition, we ensure monotonic reduction of WCRT in every iteration, so that our allocation algorithm is guaranteed to terminate.

Concretely, the main contribution of 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 studies with two complex embedded control applications for an Unmanned Aerial Vehicle (UAV) and an in-orbit monitoring instrument reveal that we can achieve significant performance improvement through appropriate content selection and runtime management of the scratchpad memory.

Following this section, we discuss the current state of research for scratchpad memory allocation and its application in the multiprocessing environment. Section 3 then establishes basic concepts and presents the problem formulation. We present the big picture of our allocation framework in Section 4, followed by the detailed techniques in Section 5. Section 6 evaluates the performance of the proposed scheme on our first case study. In Section 7, we extend our framework to handle Message Sequence Graph models and report the result when evaluated on the same case study. Next, we prove the scalability of our method by applying it on the second case study of a larger scale in Section 8. Finally, Section 9 concludes the paper and discusses extensions to the presented work.

2. RELATED WORK

2.1 Scratchpad Memory Allocation and Content Selection

The problem of content selection for scratchpad memory has been studied extensively for sequential applications. The memory objects considered for allocation into the scratchpad can be *program data* [Avissar et al. 2002; Deverge and Puaut 2007; Dominguez et al. 2005; Panda et al. 2000; Udayakumaran and Barua 2003], *program code* [Angiolini et al. 2004; Egger et al. 2006; Janapsatya et al. 2006; Ravindran et al. 2005; Verma et al. 2004a], or a combination of both [Verma et al. 2004b; Wehmeyer et al. 2004]. The different memory access behaviors between these two types give rise to different concerns. Allocating program code requires additional care to maintain program flow [Steinke, Wehmeyer et al. 2002], while allocating program data generally calls for specific considerations depending on the type of the data (global, stack, or heap) and the different nature of their access. The allocation schemes are often coupled with supporting techniques such as data partitioning [Falk and Verma 2004], loop and data transformations [Kandemir et al. 2004] or memory-aware compilation [Marwedel et al. 2004] to make the access pattern more amenable for allocation.

Existing scratchpad allocation schemes in the literature can be majorly classified into *compile-time* and *runtime* techniques (Figure 6), differing in the point of time when the allocation decision is made. *Compile-time* scratchpad allocation techniques perform offline analysis of the application program and select beneficial memory content to be placed in the scratchpad. This approach incurs no computation overhead during the execution of the application itself. The methods in this category can be further classified into *static allocation* and *dynamic overlay*.

Static allocation loads selected memory blocks into the scratchpad during system initialization, and does not change the content until the completion of the application. The techniques for scratchpad content selection include dynamic programming [Angiolini et al. 2004] and 0-1 ILP [Wehmeyer et al. 2004]. Panda et al. [2000] view the allocation problem as a partitioning of data into the different levels of the memory hierarchy. They present a clustering-based partitioning algorithm that takes into account the lifetimes and potential access conflicts among the data, as well as possibilities of context switching in a multitasking environment. The static allocation scheme is reasonably efficient to implement, even though its effectiveness may be limited to applications with relatively small memory requirement compared to available memory space.

Scratchpad allocation with dynamic overlay, on the other hand, may reload the scratchpad with new contents when the execution reaches designated program points. This approach requires a way to reason about the contents of the scratchpad memory over time. Udayakumaran [2003] introduces the concept of timestamps to mark the program points of interest. A cost model determines the cost of memory transfers at those points, and a greedy compile-time heuristic selects transfers that maximize the overall runtime benefit. Verma et al. [2004b] performs liveness analysis to determine the live range of each memory object to be placed in the scratchpad. This information is then used to construct an ILP formulation with the objective of maximizing energy savings. Steinke et al. [Steinke, Grunwald et al. 2002] also formulates the problem as an ILP optimization by modeling the cost of copying memory objects at selected program points. Kandemir et al. [2001] focus on data reuse factor and uses Presburger formula to determine the maximal set of loop iterations that reuse the elements residing in the scratchpad, in order to minimize data transfer

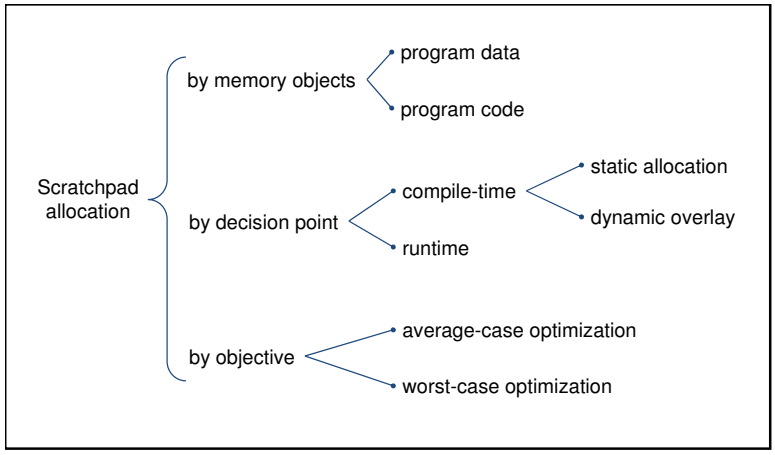


Fig. 6. Classification of scratchpad allocation techniques

between on-chip and off-chip memory. For these methods, even though the reloading of scratchpad contents is executed at runtime, the entire analysis to select memory blocks and reloading points is performed at compile time, thus incurring no runtime delay for computation of the gain functions. Note that the dynamic overlay in the above context is restricted to individual tasks, and is different from the inter-task overlay that we address in this work.

In contrast to the compile-time allocation described above, *runtime scratchpad allocation* techniques decide on the scratchpad memory content when the application is running. There can be several reasons to opt for this approach. One reason is the situation when it is not possible to perform selection at compile time, because the size of the scratchpad or the program memory requirement is unknown at compile time. Such a situation may arise when the embedded program is not burned into the system at the time of manufacture, but is rather downloaded during deployment via the network or portable media [Nguyen et al. 2005]. Another reason is the situation when the memory requirement of the specific application varies widely across its input. In this case, a beneficial allocation decision can be better determined after analysing the execution trace or history [Egger et al. 2008; Ravindran et al. 2005]. Egger et al. [2008], in particular, make use of the page fault exception mechanism of the Memory Management Unit (MMU) to track page accesses and copy frequently executed code sections into the scratchpad.

Runtime allocation methods inevitably add the cost of performing content selection to the application runtime, even if it may be offset by the gain due to allocation. Nevertheless, most methods are able to alleviate this overhead by pre-computing part of the analysis that does not depend on runtime information. Nguyen et al. [2005] first identify potentially beneficial memory blocks at compile time when the scratchpad size is still unknown; then perform the actual selection once the program loader discovers the scratchpad size at startup. In a similar approach, Dominguez et al. [2005] allow allocation of heap objects whose sizes are unknown at compile time. The method performs compile-time analysis to determine potential allocation sites, then reserves fixed-size portions in the scratchpad to be occupied by subsets of these objects once they are created, selected at runtime via a cost-model driven heuristic.

Most of the works discussed above aim to minimize the *average-case* execution time or energy consumption through scratchpad allocation. The timing predictability of scratchpad memory has made it especially suited for use in real-time systems, where the concern is instead the *worst-case* execution time (WCET) of the application. Consequently, researches have also considered scratchpad allocation to optimize the worst-case performance (Figure 6). Our previous work [Suhendra et al. 2005] presented optimal and heuristic WCET-centric static allocation methods for program data, while Deverge and Puaut [2007] considered dynamic allocation for global and stack data. The main concern in WCET-centric allocation is that the worst-case execution path of the program may change as scratchpad allocation changes. These methods account for this change in WCET path by performing iterative analysis along with incremental fine-tuning of the scratchpad allocation.

Following the classification in Figure 6, this paper addresses *static allocation* for *program code* that optimizes the *worst-case performance*. While worst-case performance is a central concern, the scheme presented here can be extended beyond the two other axes to handle data allocation as well as to further apply intra-task dynamic overlay.

2.2 Scratchpad Allocation for Multiprocessing Systems

As we move on to consider applications running on multiprocessor systems, concurrency and sharing among the multiple tasks or processing elements (PEs) become important factors. Early static allocation methods [Panda et al. 2000] simply partition the scratchpad to the tasks according to gain functions extended from allocation strategy for single-process applications. As noted in the motivating example, this simple approach suffers from underutilization of scratchpad space.

Verma et al. [2005] present a set of scratchpad sharing strategies among the processes for energy consumption minimization. Processes may take up disjoint space in the scratchpad so that no restoration is required upon context switch, or they may share the whole scratchpad whose content will be refreshed when a process is activated. A hybrid between the two is also considered. Their work assumes a statically defined schedule, where all processes have equal priority and are scheduled in a round-robin manner. In contrast, we consider priority-driven preemptive scheduling, which provides better flexibility for real-time and reactive systems as they move from one mode of execution to another during the mission, ensuring that critical functions are accomplished in time. Moreover, as the method in [Verma et al. 2005] assumes a non-preemptive system, it is not applicable to systems with heavy process interactions, where it is critical to account for interferences among tasks in order to provide real-time guarantees as in the domain we consider.

Another class of scratchpad allocation methods focuses on the *mapping of codes/data* to the scratchpad memories so as to maximize the benefit from the allocation. Avissar et al. [2002] formulate the distribution of program data among multiple memory banks in a heterogeneous system, taking into account the individual space limitation, to achieve best latency reduction. The problem is modeled via Integer Linear Programming which can be solved optimally, subject to the accuracy of the memory access profile used. Their method handles context-switching environments by simply partitioning the available space across the processors and considering all the memory requirement simultaneously. On the other hand, Kandemir et al. [2002] address the problem of mapping shared arrays to private scratchpad memories of the PEs in order to reduce off-chip memory accesses. In this setting, the private scratchpad of a PE can be accessed by other PEs albeit with slightly longer latency. Their method examines the access patterns to the shared array as issued by

the different processors, and schedules the accesses to optimize the collective reusability of the portions of the array brought into any of the scratchpad space. The more recent work of the group [Kandemir 2007] introduces the concept of inter-processor reuse distance to capture the access patterns, and optimizes the temporal locality of shared data via code restructuring to modify shared data access patterns such that accesses to the same data from multiple processors are closer in time.

Others propose runtime *customization of scratchpad sharing* among tasks or PEs to adapt to the changing memory requirement. Kandemir et al. [2004] present a strategy to dynamically reconfigure scratchpad sharing and allocation among PEs to adapt to runtime variations in data storage demand and interprocessor sharing patterns. Ozturk et al. [2006] first perform automated compiler analysis to capture memory requirements and reuse characteristics of the tasks at loop granularity, then use this information to dynamically partition the available SPM space across tasks at runtime.

In a broader perspective, researches have also looked at *exploration of scratchpad design space* in conjunction with other multiprocessing aspects. We have previously recognized the dependency among scratchpad allocation with how tasks are scheduled on the multiple processing elements, and formulated an integrated approach to optimize the gain from scratchpad allocation for all tasks [Suhendra, Raghavan, and Mitra 2006]. Issenin et al. [2006] present a multiprocessor data reuse analysis that explores a range of customized memory hierarchy organizations with different size and energy profiles.

Finally, this work complements the research on *cache-related preempted delay (CRPD) analysis* [Lee et al. 1998; Negi et al. 2003; Tomiyama and Dutt 2000; Staschulat and Ernst 2004]. CPRD analysis provides timing guarantee for concurrent software by analyzing interferences in cache memory due to process interactions. An important point of this analysis is identifying memory blocks of a process that are at risk of getting replaced by a preempting process, and the additional miss latencies. Our work, on the other hand, eliminates interference in memory through scratchpad allocation.

3. PROBLEM FORMULATION

Let us now formally present our problem formulation.

3.1 Application Model

The input to our problem is in the form of Message Sequence Chart (MSC) [Alur and Yannakakis 1999; ITU-T 1996] that captures process interactions corresponding to a concurrent embedded application. An MSC is equivalent to the labeled partial sequence order of a task graph model. It provides an explicit, intuitive view of the task distribution on multiple processes within the application, which helps in the design of our interaction-based allocation schemes.

We assume a *preemptive, multi-tasking* execution model. The application is periodic in nature. The MSC represents interactions within one such invocation. In other words, a complete execution of the MSC accomplishes one activation of the application 'mission', with the 'work' distributed among the processes involved. Therefore, all tasks in all involved processes within the same MSC adhere to a common period and deadline, which is the period of the application. The concept of task periodicity can thus be abstracted from our framework, which works on a single MSC. In general, an application may have several functionalities that execute with different periods. Such an application needs to be represented as a set of MSCs in order to capture all possible process interactions. The

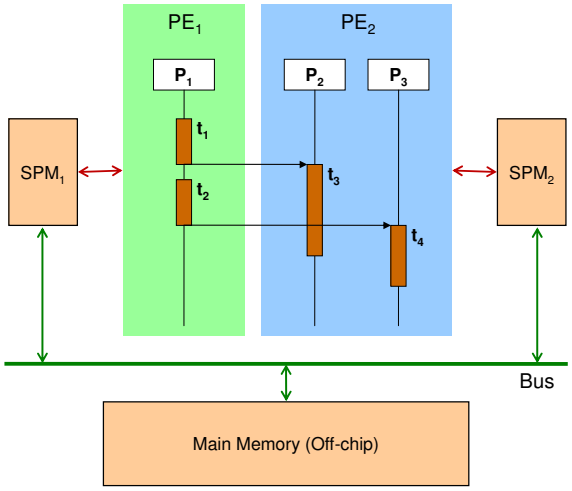


Fig. 7. A simple MSC running on two processing elements PE_1 and PE_2 with scratchpad memories SPM_1 and SPM_2, respectively

MSCs can then be aggregated via unfolding to the hyper-period and replicating the tasks as necessary, before being handled by our method.

The underlying hardware platform contains one or more processing elements (PEs), each associated with a private scratchpad memory. We do not consider any other memory. Figure 7 shows a simple example that illustrates this setting. The scratchpad memories are connected via bus to a main memory. The scratchpad may be reloaded at runtime with content from the main memory through this connection, incurring a constant delay per fetch (bus contention is abstracted). Inter-task communication takes place via message exchange in separate buffers, which are not modeled in this paper beyond the introduced delay. The scratchpad memory is reserved for the memory requirement of tasks alone, which in this paper refers to program code.

A vertical line in the MSC represents the lifeline of a process, that is, the time period during which the process is alive. A process may consist of more than one tasks. A process typically corresponds to a specific functionality, and 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 in the MSC reflects their order of execution on the PE. In Figure 7, tasks t_1 and t_2 belong to the same process P_1 scheduled on PE_1, and t_2 executes after t_1 completes execution on the same PE. Dependencies across processes are represented as horizontal arrows from the end of the *predecessor* task to the start of the *successor* task. In our example, the communication delay between processes are zero. Including non-zero communication delay in the analysis is straightforward: the start of the successor task is simply pushed back by the amount of the delay.

Each process is assigned a unique static priority. The priority of a task is equal to the priority of the process it belongs to. If more than one processes are assigned to the same PE, then a task executing on that PE may get preempted by a task from another process with higher priority if their lifetimes overlap. The assignment of static priorities to processes and the mapping of processes to PEs are inputs to our framework. Note that static

priority assignment alone does 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. In Figure 7, supposing process P_3 has higher priority than P_2 on PE_2, then task t_3 will be preempted when task t_4 becomes ready, *if* the execution times of the tasks in that particular invocation are such that t_3 has not completed execution when task t_2 completes on PE_1. We see that this situation is determined not only by the tasks involved in the preemption, but other tasks in the system with dependency relationship with these tasks as well.

The analysis and discussion in the rest of this paper will be at the task level instead of the process level, as we make allocation decision for each task individually. Formally, 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:

(1) a static priority, $pr(t_i)$, in the range $[1, R]$ with 1 being the highest priority, and
(2) mapping to a PE, $PE(t_i)$, in the range $[1, Q]$, where Q is the number of PEs in the system.

As mentioned earlier, all the tasks belonging to a process have the same priority and are mapped to the same PE. This policy arises from the fact that the concept of a process represents a specific functional unit in the actual system. For example, the radio controller unit in our referenced UAV application is represented as the 'Radio Control' process in the MSC model (Figure 2), and all tasks of the 'Radio Control' process executes on the radio controller unit in reality. As we will see in the experiments (Section 7), this convention may lead to less-than-ideal processor utilization in certain cases, but is nevertheless realistic.

3.2 Response Time

We use the term *task lifetime* to mean the interval from the time a task is started and the time it completes execution, specified in absolute time values. The absolute time that a task can be started depends on the completion times of its predecessors, according to the dependency specified in the MSC model. The length of a task's lifetime is its *response time*, which consists of the time it takes to perform computation (without interruption) and the delay it experiences due to preemptions by higher priority tasks.

In general, the computation time of a task may vary due to (1) the variation in input data that triggers different execution paths within the program, and (2) the variation in memory access latencies (whether the accessed code/data object is in scratchpad or main memory). The uninterrupted computation time required by each individual task can be determined via static analysis [Li et al. 2007] of the program corresponding to a task, and represented as a range between its best-case execution time (BCET) and worst-case execution time (WCET) [Mitra and Roychoudhury 2007]. However, this estimation is intertwined with the second component, memory access latencies, as we will elaborate when we discuss scratchpad allocation in the next subsection. The timing analysis needs to account for both components in an integrated manner [Suhendra, Mitra, and Roychoudhury 2006].

Obviously, the longer the execution time of a task, the longer its response time. However, the impact of a task's response time on *other tasks' response times (and thus overall application response time)* is not straightforward. In the example shown in Figure 7, a longer t_2 execution time will cause t_4 to start later and not preempt t_3 on the same PE (supposing t_4 has higher priority than t_3). In this scenario, the response time of t_3 becomes shorter, and this possibly leads to an earlier completion time of the overall application. As our ultimate

aim is to optimize for the *worst-case response time (WCRT) of the whole application*, we need to take into account the interplay among these components.

3.3 Scratchpad Allocation

The term *scratchpad allocation* in our context consists of two components: (1) the distribution of scratchpad space among the application tasks, and (2) the selection of memory blocks of each task to fit into the allocated space. When a task can access part of its memory requirement from the scratchpad memory instead of the significantly slower main memory, its execution time can reduce dramatically. Depending on the portion of memory address space allocated to the scratchpad memory, the time taken by a single execution path within the task itself may vary greatly. Therefore, in the presence of scratchpad memory, the execution path and execution time of a task in the best or worst case should be determined by taking into account the allocation decision.

In this work, we consider allocating program codes of the tasks into the scratchpad. The granularity of allocation can be basic blocks or entire functions, or other less common options such as partial basic blocks. Hereafter we shall use the general term of 'code blocks' to refer to the unit of allocation. The methodology presented here uses the granularity of basic blocks, as will be apparent in Section 5. The presented method can be directly applied to data allocation so long as the accesses can be determined statically. In this case, instead of code blocks of the task, the candidates for allocation are data variables (scalars and arrays) accessed by the task. Data allocation does not affect program control flow as code allocation does, as we shall see in later sections. Instead, the major concern in data allocation is regarding dynamically allocated data, whose sizes are not known at compile time. The full treatment of data allocation is left as future work.

To make better use of the limited scratchpad space, we allow scratchpad overlay, that is, the same scratchpad memory space can be allocated to two or more tasks as long as they have disjoint lifetimes. Certainly, this condition is always true for tasks belonging to the same process, as they execute one after another. However, analysis may also reveal tasks on different processes that have disjoint lifetimes and may share scratchpad space. In our formulation, we do not make explicit distinction of these cases, and only focus on task-level non-interference property as a requirement for scratchpad overlay. As apparent from Figure 7, each PE accesses its own scratchpad, and the space will be shared among tasks executing on the corresponding PE only; we do not consider accesses to remote scratchpad which will be effective only if the PEs have access to fast interconnection network [Kandemir et al. 2002].

Formally, let $\mathcal{S}$ be a particular scratchpad allocation for the application. As described earlier, $\mathcal{S}$ consists of two components:

(1) $space(t_i)$, the amount of scratchpad space allocated to each task $t_i, 1 \leq i \leq N$, and
(2) the allocation of $space(t_i)$ among the code blocks of t_i.

By virtue of scratchpad overlay, the sum of the scratchpad space allocated to all tasks, $\sum_i^N space(t_i)$, is not necessarily less than or equal to the total available scratchpad space. This will be clear in the description that follows.

Let $Mem(t_i)$ denote the set of all code blocks of t_i available for allocation. Given $space(t_i)$ assigned in $\mathcal{S}$, the allocation $Alloc(t_i, \mathcal{S}) \subseteq Mem(t_i)$ is the set of most profitable code blocks from t_i to fit the capacity. The BCET and WCET of t_i as a result of allocation $\mathcal{S}$ are denoted as $bcet(t_i, \mathcal{S})$ and $wcet(t_i, \mathcal{S})$ respectively. Given an allocation $\mathcal{S}$

and the corresponding BCET, WCET of the tasks, we can estimate the lifetime of each task t_i, defined as the interval between the lower bound on its start time, $EarliestSt(t_i, \mathcal{S})$, and the upper bound on its completion time, $LatestFin(t_i, \mathcal{S})$. This estimation should take into account the dependencies among the tasks specified in the model (total order among the tasks within a process, and the ordering imposed by message communication) as well as preemptions.

The WCRT of the whole application is now given by

$$WCRT = max_{1 \leq i \leq N}\ LatestFin(t_i, \mathcal{S}) - min_{1 \leq i \leq N}\ EarliestSt(t_i, \mathcal{S}) \qquad (1)$$

that is, the duration from the earliest start time of any task in the application until the latest completion time of any task in the application.

Our goal is to construct the scratchpad allocation $\mathcal{S}$ that utilizes inter-task overlay to minimize the WCRT of the application (Equation 1). We define a *scratchpad overlay set* for a certain time interval $[s, e]$ as a set of tasks that are allotted the same space in the scratchpad during the time interval $[s, e]$. Such a set, denoted $G_{[s,e]}$, is constructed to have the following properties.

(1) All tasks in the set start and complete execution within the time interval $[s, e]$.

$$\forall\ t \in G_{[s,e]} \quad s \leq EarliestSt(t, \mathcal{S}) < LatestFin(t, \mathcal{S}) \leq e$$

(2) Any pair of tasks in the set have disjoint lifetimes. That is, for any pair of tasks (t, u) chosen from the set, either u completes before t starts, or t completes before u starts. The tasks are not necessarily related via dependency; the situation may also arise if the times of dispatch are well separated so that one task invariably completes before the other is ready.

$$\forall\ t, u \in G_{[s,e]} \quad EarliestSt(t, \mathcal{S}) > LatestFin(u, \mathcal{S}) \\ \vee\ EarliestSt(u, \mathcal{S}) > LatestFin(t, \mathcal{S})$$

(3) The set, as a whole, shall be allotted a portion of size $space(G_{[s,e]})$ in the scratchpad throughout the interval $[s, e]$. $space(G_{[s,e]})$ is not necessarily non-zero. All tasks in the set may occupy this same portion for allocation. It is important to note that the space should be reserved throughout the interval to preserve predictability.

$$\forall\ t \in G_{[s,e]} \quad space(t) \leq space(G_{[s,e]})$$

The illustration in Figure 8 is based on the same example presented in Figure 3 earlier. If we consider the time interval $[a, d]$, a valid overlay set is $\{fm_1, fm_2, fr_1, fm_4\}$.

More than one scratchpad overlay sets can exist for the same time interval. To construct a complete allocation layout for a certain time interval $[s, e]$, we construct the sets such that *every* task that executes within $[s, e]$ (satisfying property (1)) belongs to *exactly one* overlay set for that interval. The overlay set may have the task as a single member if the task's lifetime overlaps with all other tasks (failing property (2)). When $[s, e]$ covers the entire application lifetime, we have the complete allocation solution for the application.

Collectively, all overlay sets for the same time interval $[s, e]$ will contain all tasks that are active within $[s, e]$. We use the term *configuration* to refer to the collective overlay sets. The implication of this concept is that the scratchpad space will be partitioned exclusively among the sets in the same configuration, while tasks within each set will occupy the same partition over time. That is, the sum of $space(G_{[s,e]})$ over all overlay sets in a configuration

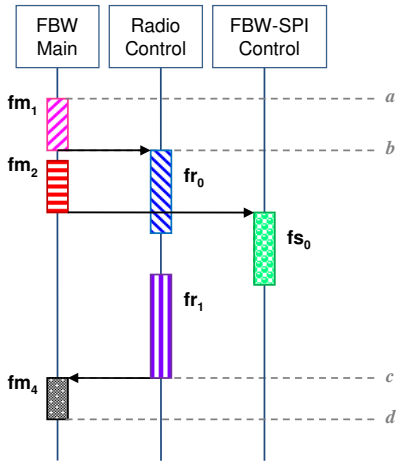


Fig. 8. MSC extracted from the UAV control application case study, with time intervals

is less than or equal to the scratchpad capacity. In general, there can be more than one ways to group tasks into overlay sets, and thus more than one choices of configuration that satisfy the above properties. Our goal then translates to finding the optimal configuration in terms of WCRT reduction.

In Figure 8, the time interval $[a, d]$ covers the entire execution of the MSC. For this interval, we can construct a configuration consisting of three overlay sets

$$\{\ \{fm_1, fm_2, fr_1, fm_4\},\ \{fr_0\},\ \{fs_0\}\ \}$$

A different possible configuration is

$$\{\ \{fm_1, fm_2, fs_0\},\ \{fr_0, fr_1\}\ \}$$

Both choices give a complete solution to scratchpad allocation for the MSC, and the evaluated gain of these choices will determine the actual layout chosen.

In the above, we have considered the entire MSC duration at once. In Figure 8, we can observe 'clean breaks' of execution at point b and point c. In real applications, these may correspond to the points when the system moves from one stage to another, for instance. Naturally, these are ideal points to reload the entire scratchpad, as all current contents (belonging to completed tasks) have just become obsolete. In our context, this translates to sub-dividing the time interval. The configuration constructed for each interval will have less tasks that are competing for space, and therefore potentially greater gain can be achieved. In the illustration, if we consider the time interval $[a, b]$ alone, the trivial choice of configuration is the singleton $\{\{fm_1\}\}$. This implies that task fm_1 can utilize the entire scratchpad space throughout its execution, since no other task is active at the same time. Similarly, $\{\{fm_4\}\}$ is the obvious choice for time interval $[c, d]$. We can then search for an optimizing configuration for $[b, c]$ to complete the allocation solution.

The rest of this paper shall elaborate the schemes that seek to define such optimizing time intervals and scratchpad overlay configurations.

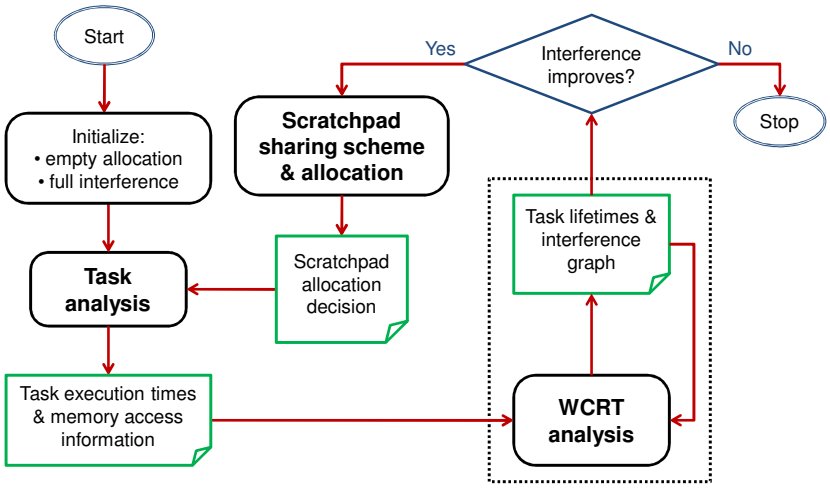


Fig. 9. Workflow of the proposed WCRT-optimizing scratchpad allocation

4. METHOD OVERVIEW

Our proposed method for scratchpad allocation is an iterative scheme (Figure 9). Analysis is performed on each task to determine the bounds on its execution time, given the initially empty scratchpad allocation. The memory access information of the task is also produced as a by-product of this analysis, to be used in the later steps. The WCRT analysis then takes in the execution time values and computes the lifetimes of all tasks. An inter-task *interference graph* is then constructed. Figure 10a shows task lifetimes computed by the WCRT analysis for our MSC example in Figure 3, along with the constructed interference graph. An edge between two nodes in the graph implies overlapping lifetimes of the two tasks represented by the nodes.

Based on the analysis result, we can decide on a suitable scratchpad sharing scheme and select actual scratchpad contents for each task, making use of the memory access information from the earlier task analysis. One possible scheme is illustrated in Figure 10b, which shows the space sharing among tasks as well as the dynamic overlay over time. With the change in allocation, the execution time of each task is recomputed, and the WCRT analysis is performed once again to update task lifetimes (see Figure 9). We ensure at this point that inter-task interference does not worsen. The reason and technique for this will be elaborated in the discussion that follows. If task interference has been reduced because of scratchpad allocation, we end up with more tasks that are disjoint in their lifetimes (Figure 10c). We shall see in Section 5 how we may direct the solution to actively seek reduced task interference as shown. These tasks can now enjoy more scratchpad space through dynamic overlay. If this is the case, we proceed to the next iteration, in which the scratchpad sharing scheme is re-evaluated and the allocation is enhanced (Figure 10d); otherwise, the iteration terminates.

We elaborate each of these steps in the following.

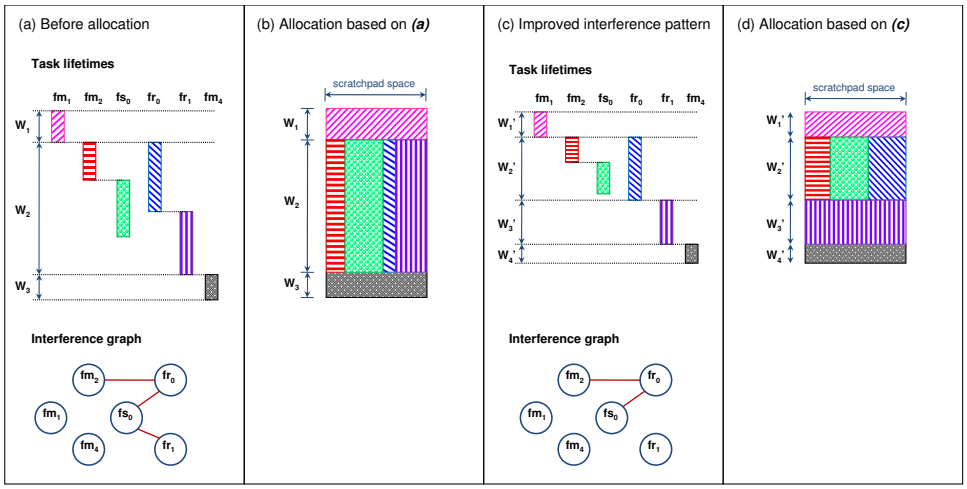


Fig. 10. Task lifetimes before and after allocation, and the corresponding interference graphs

4.1 Task Analysis

The Task Analysis step determines both the best-case execution time (BCET) and worst-case execution time (WCET) of each task, given a certain scratchpad allocation. These two values bound the range of uninterrupted execution time required to complete the task given all possible inputs.

The timing analysis of the task for a given scratchpad allocation proceeds as follows. We extract the control flow graph (CFG) of each task. Each node in the CFG is a basic block, a sequence of program instructions that forms the unit of the path-based analysis. The time required to execute each of these basic blocks can be determined via micro-architectural modeling that accounts for the execution cycles of each instruction, along with features such as pipelining and branch prediction policy. In this work, we assume a simple non-pipelined architecture with perfect branch prediction. With this assumption, the only factor affecting execution time is the memory latency, which is the focus of this paper. Nevertheless, the micro-architectural modeling is an independent routine whose effect is restricted to the basic block execution time determination, and can be extended to more complex execution platforms using a state-of-the-art WCET analysis tool such as Chronos [Li et al. 2007].

To the basic block execution time obtained above, we add the additional delay for fetching these instructions from memory, whose value depends on whether the allocation decision places this particular basic block in the scratchpad or in the main memory. Note that this work uses the allocation granularity of basic blocks, which coincides with the computation unit in a path-based timing analysis. A different choice of allocation granularity can be accomodated during the timing analysis by propagating the allocation decision to each basic block (for granularity larger than a basic block) or breaking down the computation for the basic block (for granularity smaller than a basic block).

A static path-based timing analysis method we previously developed [Suhendra, Mitra, and Roychoudhury 2006] then traverses the CFG to find the path with the longest (respec-

tively, shortest) execution time. The program may contain loops, which make the CFG cyclic. We require that the bounds on the iteration counts are specified as input to the analysis, so that the execution time can be bounded. To tighten the estimation, our timing analysis method performs simple infeasible path detection during traversal of the CFG. This is achieved by ruling out pairwise branch outcomes that are statically known to be impossible given the constraint implied by the execution sequence up to that point. This detection reduces the possibility of reporting an infeasible execution path as the worst-case (respectively, best-case) execution path, which may result in large overestimation and mislead the scratchpad allocation decision.

After determining the best-case and worst-case execution paths, we may then extract the execution frequencies of basic blocks along these paths. This gives us the current *gain* of allocating each of the blocks. The *cost* of allocating the block is the area it occupies in the memory space. These two values form the *memory access information* of the task, which will serve as input to the scratchpad content selection in the next iteration of the our technique. In this particular setting, we choose to use the memory access information corresponding to the worst-case execution path. This information will be used in refining the scratchpad allocation, which will in turn affect the execution time of each basic block, and ultimately affect the best- and worst-case execution paths. Following each such change, the memory access information is also updated. We see here that the allocation step and the task analysis form a feedback loop, which justifies the need for an iterative solution.

4.2 Worst-Case Response Time (WCRT) Analysis

As established in the earlier discussion, the response time of a single task has two components: its uninterrupted execution time, and the delay due to preemptions by higher priority tasks. The preemption delay is itself a function of the execution time of the task, as the lifetime of a task affects the way it interacts with other tasks.

The WCRT Analysis step takes the task execution times estimated in the Task Analysis step and the dependencies specified in the MSC model as input. Based on these, it determines the lifetime of each task t_i, which ranges from the time when t_i may start execution until the time when t_i may complete execution, represented by the four values $EarliestSt(t_i, \mathcal{S})$, $LatestSt(t_i, \mathcal{S})$, $EarliestFin(t_i, \mathcal{S})$, and $LatestFin(t_i, \mathcal{S})$. The WCRT of the application given scratchpad allocation $\mathcal{S}$, as formulated in Equation 1, is determined via a fixed-point computation that updates these values for each task t_i as more interaction information becomes available throughout the iteration. The change in each iteration is brought on by the variation in the execution time of the individual tasks as the allocation is refined, as well as possible preemption scenarios that arise from the different task lifetimes.

The WCRT analysis method is modified from Yen and Wolf's method [1998] and proceeds as follows. We first examine the delay component resulting from preemptions among tasks assigned to the same PE. A higher priority task can only preempt t_i on the same PE if it is possible to start execution *during* the execution of t_i, that is, after t_i starts and before t_i finishes. Recall that we denote the priority of task t_i as $pr(t_i)$, with a smaller value translating to a higher priority. Certainly, the priority order makes sense only among tasks competing on the same PE. The following equation defines the set of such tasks, denoted as $intf(t_i)$.

$$intf(t_i) = \{\ t_j \mid pr(t_j) < pr(t_i)\ \wedge \quad (2)$$
$$EarliestSt(t_i, \mathcal{S}) < EarliestSt(t_j, \mathcal{S}) < LatestSt(t_j, \mathcal{S}) < LatestFin(t_i, \mathcal{S})\ \}$$

In the very first round of computation, the start and end times of each task are not yet known, and it is assumed that all higher priority tasks on the same PE can preempt t_i unless they are related by dependency. That is, before the start and end times of tasks are completely determined, the last condition in Equation 2 fails only for dependencies inferred from the MSC model. This information will be refined in the subsequent rounds as we shall see later in this section.

Given the possible preemptions as inferred above, the WCRT of a single task t_i can then be computed as

$$wcrt(t_i, \mathcal{S}) \ = \ wcet(t_i, \mathcal{S}) + \sum_{t_j \in intf(t_i)} wcet(t_j, \mathcal{S}) \quad (3)$$

The term $wcet(t_i, \mathcal{S})$ refers to the WCET value of t_i given allocation $\mathcal{S}$ determined in the Task Analysis step. The second term in the above equation gives the total preemption delay endured by t_i in the worst scenario. As all tasks in the model adhere to the same period (see Section 3.1), each task may preempt another task at most once, and the maximum duration of the delay due to a preempting task $t_j \in intf(t_i)$ is the WCET of t_j. The best-case response time (BCRT) of t_i can be computed similarly.

The start and completion time of task t_i are related to $wcrt(t_i, \mathcal{S})$ and $bcrt(t_i, \mathcal{S})$ as follows.

$$EarliestFin(t_i, \mathcal{S}) = EarliestSt(t_i, \mathcal{S}) + bcrt(t_i, \mathcal{S})$$

$$LatestFin(t_i, \mathcal{S}) = LatestSt(t_i, \mathcal{S}) + wcrt(t_i, \mathcal{S})$$

Further, the partial ordering of tasks in the MSC imposes the constraint that task t_i can start execution only after all its predecessors have completed execution, that is

$$EarliestSt(t_i, \mathcal{S}) \geq EarliestFin(t_j, \mathcal{S})$$

$$LatestSt(t_i, \mathcal{S}) \geq LatestFin(t_j, \mathcal{S})$$

for all tasks t_j preceding t_i in the partial order of the MSC.

Observing these constraints, the WCRT analysis computes the lifetimes of all tasks in the application. As mentioned earlier, the first round of this computation uses task WCRT values that have been determined with the assumption that preemptions will occur between any pair of tasks with no dependency. Once the start and finish times of tasks are determined in a certain iteration, it may become apparent that the lifetimes of certain tasks are well separated, and thus they cannot preempt one another. Given this refined information, the estimation of each task lifetime becomes tighter in the subsequent iteration. The fixed-point computation terminates when there is no further refinement. The application WCRT is then determined based on Equation 1.

From the analysis result, tasks with overlapping lifetimes are said to be interfering, with the higher-priority task possibly preempting the lower-priority task. This interference pattern is captured in a *task interference graph* (Figure 10a) for the purpose of scratchpad allocation in the next stage.

4.3 Scratchpad Sharing Scheme and Allocation

Given the current interference pattern captured in the interference graph, we decide on a scratchpad sharing scheme among the tasks that incurs no unpredictable reloading delay when tasks resume after preemption. In this paper, we consider four scratchpad sharing schemes with varying sophistication. The simplest scheme (*Profile-based Knapsack*) performs scratchpad space distribution and allocation without any regard for the interference pattern. The second scheme (*Interference Clustering*) groups tasks based on their lifetime overlap, thus isolating the interference within mutually exclusive time windows and enabling time-multiplexing among the groups. The third scheme (*Graph Coloring*) improves this by mapping the allocation problem to a graph coloring problem. Our final proposal (*Critical Path Interference Reduction*) eliminates interferences that compromise tasks on the critical path of the application by inserting strategically placed slacks, in order to improve the situation before applying the allocation scheme. These techniques will be discussed in full details in the next section.

Figure 10b and 10d visualize possible allocation schemes based on task lifetimes in Figure 10a and 10c, respectively. They clearly show how task interference pattern influences the allocation decision. An important feature of the allocation is that each task occupies the space assigned to it for the whole duration of its execution, without being preempted by any other task. This ensures that reloading of the scratchpad occurs exactly once for each task activation, and the incurred delay can be tightly bounded.

In each scheme, aside from the scratchpad sharing, the memory content to be allocated in the scratchpad for each task is also selected for optimal WCRT. After the allocation is decided, each task goes through the Task Analysis step once again to determine the updated BCET, WCET, and worst-case memory access information.

4.4 Post-Allocation WCRT Analysis

Given updated task execution times 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 and BCET 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 issue, suppose the interference graph in Figure 11a leads to the allocation decision in Figure 11b. The reduction in WCET due to the allocation in turn reduces task response times and changes task lifetimes to the one shown in Figure 11c. However, this computation of lifetimes is incorrect, because it has assumed the BCET and WCET values 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 Figure 11c, 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 deduced by the WCRT analysis in Figure 11c 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 a certain allocation decision is made *will not become interfering* in the updated lifetimes. This is accomplished by introducing a *slack* that forces the later task to "wait out"

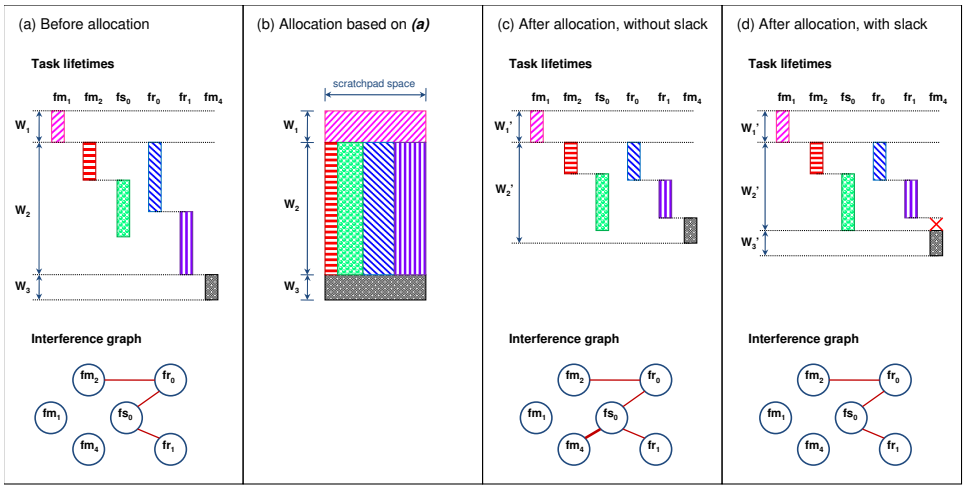


Fig. 11. Motivation for requiring non-increasing task interference after allocation

the conflicting time window. The adapted WCRT analysis consults existing interference graph and adjusts the start time of fm_4 such that

$$EarliestSt(fm_4, \mathcal{S}) \geq LatestFin(fs_0, \mathcal{S})$$

The start times of tasks that are dependent on fm_4 are adjusted accordingly. Figure 11d shows the adjusted schedule, which maintains the same interference graph as Figure 11a by forcing fm_4 to start after fs_0 has completed, thus inserting a slack between the fm_4 and its immediate predecessor fr_1.

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 (for example, in a situation depicted in Figure 10). When this happens, we iterate over the allocation and analysis steps to enhance current decision, until no more improvement can be made (Figure 9). Through the above step, we enforce that task interferences are monotonically non-increasing from one iteration to the next. Therefore, the gain from scratchpad allocation, which is decided based on the interference pattern, will either improve or equal to the gain from the previous iteration. Given that the scratchpad capacity is finite, the improvement will reach a point where no further refinement is enabled. The iterative allocation scheme is thus guaranteed to terminate.

5. ALLOCATION METHODS

This section describes the scratchpad allocation routine, which is the main focus of our paper. The emphasis in our scratchpad allocation strategy is to achieve a balance between scratchpad overlay (*time-sharing*) and scratchpad partitioning (*space-sharing*). We want to maximize the utilization of the available scratchpad space via overlay for tasks which are guaranteed to never interfere with each other, while maintaining timing predictability by reserving partitions among tasks which may be preempted during its execution.

Figure 12 illustrates four allocation schemes considered in this paper. The left side of each picture shows task lifetimes as determined by the WCRT analysis, and the right side

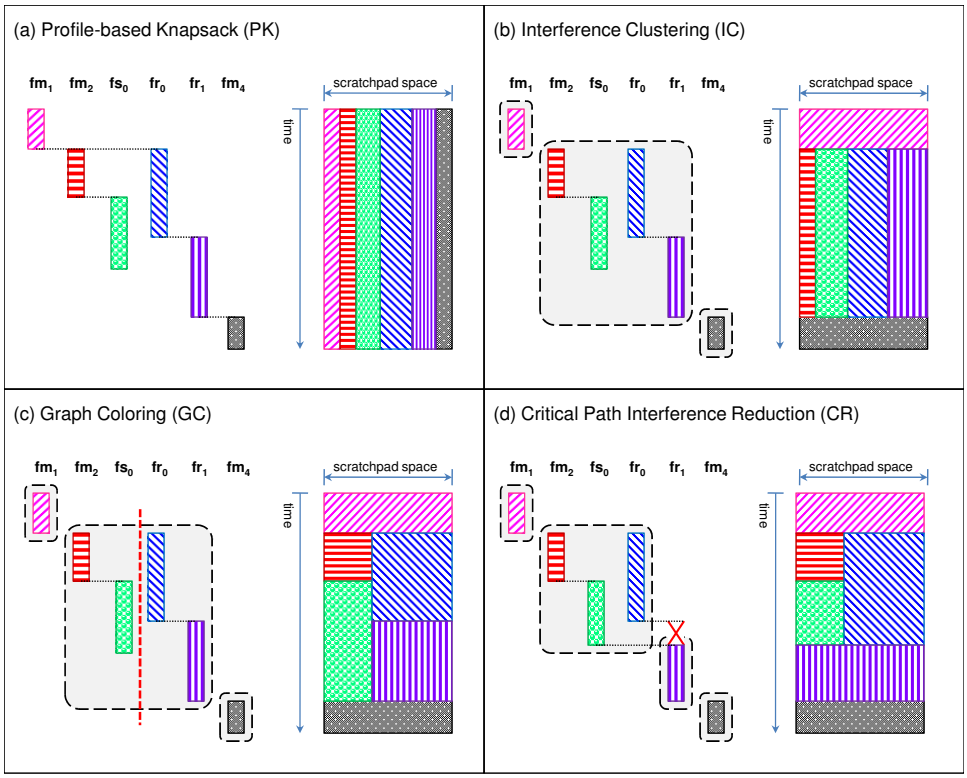


Fig. 12. Four scratchpad allocation schemes with varying degrees of sophistication

sketches the state of the scratchpad memory due to the different allocation schemes. We shall now elaborate on the techniques utilized in each scheme.

5.1 Profile-based Knapsack (PK)

As the baseline method, we consider a straightforward static allocation strategy where all tasks executing on the same PE will share the PE's scratchpad space throughout application lifetime. It does not take into account the possible interferences among tasks running on the PE. The main focus here is the scratchpad content selection routine, which uses the information on sizes (cost) and access frequencies (gain) of code blocks along the worst-case execution path of the tasks. In other words, the allocation decision is based on the 'worst-case execution profile' obtained via static analysis of the task. We thus refer to this scheme as *Profile-based Knapsack (PK)*, illustrated in Figure 12a.

As each PE makes use of its own scratchpad space, we can perform allocation for each PE independently. Partitioning and static scratchpad allocation for a PE q can be simultaneously optimized via a 0-1 Integer Linear Programming (ILP) formulation.

Objective Function. An ILP formulation for an allocation that minimizes energy consumption has been presented in [Steinke, Wehmeyer et al. 2002]. However, our objective here is to minimize the WCRT of the whole application. This presents a significantly differ-

ent challenge, as there exists no closed-form linear representation for the definition of the application WCRT given a particular allocation decision (Equation 1). In fact, the WCRT evaluation requires a fixed-point computation as we have seen in the previous section. Hence, the objective function to find the optimal allocation $\mathcal{S}$ is formulated to *approximate* this definition, as follows.

$$\sum_{t_i:PE(t_i)=q} (F(t_i)+d) \times wcet(t_i, \mathcal{S})$$

Recall that $wcet(t_i, \mathcal{S})$ denotes the WCET of task t_i given scratchpad allocation $\mathcal{S}$. This variable is weighted by the value $F(t_i)+d$, which measures the *potential contribution of the task to the application WCRT*.

Task Criticality. The first term, $F(t_i)$, represents the contribution of t_i to the *current* application WCRT, which includes the delay t_i possibly introduces when it preempts any other task. The value of $F(t_i)$ is estimated based on the current WCRT path of the application (critical path) $\mathcal{P}_0$ using the following rules:

$$F(t_i) = \begin{cases} \frac{lifetime(t_i,\mathcal{S}_0)}{\mathcal{W}_0} & \text{if } t_i \in \mathcal{P}_0; \\ \frac{wcrt(t_i,\mathcal{S}_0)}{\mathcal{W}_0} & \text{if } t_i \text{ may preempt a task in } \mathcal{P}_0; \\ 0 & \text{otherwise.} \end{cases}$$

The term $\mathcal{S}_0$ denotes the current scratchpad allocation (empty for the first allocation attempt), while the term $\mathcal{W}_0$ denotes the current application WCRT.

If t_i is part of the current WCRT path, then t_i's contribution to the current application WCRT is the ratio of the length of its lifetime to the overall WCRT $\mathcal{W}_0$. Here, the task lifetime $lifetime(t_i, \mathcal{S}_0)$ includes the whole duration when t_i is active, from the earliest start time to the latest completion time:

$$lifetime(t_i, \mathcal{S}_0) = LatestFin(t_i, \mathcal{S}_0) - EarliestSt(t_i, \mathcal{S}_0)$$

If t_i is not on the critical path itself but may preempt a task on that path, then t_i may introduce a preemption delay up to the maximum length of its own response time, $wcrt(t_i, \mathcal{S}_0)$. Recall that the WCRT of an individual task t_i is related to its lifetime as follows.

$$wcrt(t_i, \mathcal{S}_0) = LatestFin(t_i, \mathcal{S}_0) - LatestSt(t_i, \mathcal{S}_0)$$

Critical Path Shift. The second term of the objective function, d, is a measure of the *density* of the critical path of the application, defined as the ratio of the number of tasks in the critical path to the total number of tasks in the application, N:

$$d = \frac{|\mathcal{P}_0|}{N}$$

The value of d ($0 < d \leq 1$) is evaluated following each round of WCRT analysis. In our context here, it may be taken to indicate the reverse likelihood of the critical path shifting. If few, long tasks dominate the runtime of the application, then it is likely that these tasks will remain in the critical path. In this case, d evaluates to a small value, and the allocation objective will put more weight on $F(t_i)$. Hence, tasks that concretely participate in the current WCRT path will be prioritized over tasks that are unlikely to contribute to the WCRT. If, on the contrary, the critical path is formed by many tasks, then d will have a large value, which serves as a base weight for tasks that do not participate in the current

critical path. This induces a more balanced runtime reduction across tasks that have similar possibilities of forming the application WCRT path.

Task Execution Time. We now examine the variable component of the objective function. The WCET of t_i in the presence of scratchpad allocation $\mathcal{S}$ is defined as

$$wcet(t_i, \mathcal{S}) = c(t_i) - saving(t_i, \mathcal{S}) + load(t_i, \mathcal{S}) + trans(t_i, \mathcal{S})$$

$c(t_i)$ is the uninterrupted worst-case running time of t_i without any allocation (i.e., all code blocks are fetched from the main memory), which is evaluated once for each task during the initial Task Analysis step. From this value, we discount the time savings due to allocation, $saving(t_i, \mathcal{S})$, but account for the time needed to load the allocated blocks into the scratchpad once at the start of the task, $load(t_i, \mathcal{S})$, as well as additional instructions needed for transition between scratchpad memory and main memory, $trans(t_i, \mathcal{S})$. We elaborate these terms below.

Time Saving. Let wd denote the unit of memory transfer of the system, that is, the amount of memory (in bytes) that can be transferred in a single fetch. Further, let lat_S denote the latency of a single memory fetch from the scratchpad, and let lat_M denote the latency of a single fetch from the main memory. Naturally $lat_M > lat_S$, and the difference between the two gives the time saving for each access to a memory unit of size wd allocated in the scratchpad. The total time saving for t_i due to allocation $\mathcal{S}$ is defined as

$$saving(t_i, \mathcal{S}) = \sum_{b \in Alloc(t_i, \mathcal{S})} freq_b \times \left\lceil \frac{area_b}{wd} \right\rceil \times (lat_M - lat_S)$$

$area_b$ is the memory space (in bytes) occupied by a memory block b, and $freq_b$ is the execution frequency of b in the worst-case execution path; both information are obtained during the Task Analysis step. As defined in the problem formulation, $Alloc(t_i, \mathcal{S})$ refers to the selected set of code blocks of t_i in scratchpad allocation $\mathcal{S}$. In the ILP formulation, this term is expanded using binary decision variables X_b for each block $b \in Mem(t_i)$, whose value is 1 if $b \in Alloc(t_i, \mathcal{S})$, or 0 otherwise. Recall that $Mem(t_i)$ is the set of all code blocks of t_i that are considered for allocation (see Section 3.3). The above definition translates to

$$saving(t_i, \mathcal{S}) = \sum_{b=0}^{|Mem(t_i)|} X_b \times freq_b \times \left\lceil \frac{area_b}{wd} \right\rceil \times (lat_M - lat_S)$$

Loading Time. The time taken to load the scratchpad at the start of the task is directly proportional to the total size of the allocated blocks, fetched once from the main memory:

$$load(t_i, \mathcal{S}) = \sum_{b \in Alloc(t_i, \mathcal{S})} \left\lceil \frac{area_b}{wd} \right\rceil \times lat_M$$

which is expanded in the same manner as $saving(t_i, \mathcal{S})$ into

$$load(t_i, \mathcal{S}) = \sum_{b=0}^{|Mem(t_i)|} X_b \times \left\lceil \frac{area_b}{wd} \right\rceil \times lat_M$$

Transition Cost. In this work, we handle code allocation with the granularity of a basic block. This choice requires that an adjustment is made to maintain correct control flow

[Steinke, Wehmeyer et al. 2002]. Scratchpad allocation essentially divides the program address space between the scratchpad and the main memory. The basic block allocation granularity makes it possible for two sequential basic blocks in the control flow graph to be stored in non-sequential order in memory (one in scratchpad and the other in main memory). If the basic block ends with an unconditional jump, the compiler will supply the correct destination address to the jump instruction after allocation is decided. Otherwise, an additional jump instruction has to be inserted at the end of the earlier block to maintain the correct control flow. Note that this applies to a branch instruction as well, because in the case that the branch condition is not satisfied, the control will simply fall through to the next basic block.

Let Y_b be a binary variable whose value is 1 if b is allocated in the scratchpad ($X_b = 1$) but the block following it is not ($X_{b+1} = 0$) and 0 otherwise. When $Y_b = 1$, it means that an additional jump instruction should be inserted at the end of block b in the scratchpad to jump to the start of block $(b + 1)$ in the main memory. Similarly, let Z_b be the binary variable whose value is 1 if $X_b = 0$ and $X_{b+1} = 1$, to represent the insertion of a jump instruction at the end of block b in the main memory to jump to the start of block $(b+1)$ in the scratchpad. Otherwise, Z_b has value 0. The definition of Y_b and Z_b can be linearized in terms of X_b and X_{b+1} as follows:

$$Y_b <= X_b \ ; \ Y_b <= 1 - X_{b+1} \ ; \ Y_b >= X_b - X_{b+1}$$

$$Z_b <= X_{b+1} \ ; \ Z_b <= 1 - X_b \ ; \ Z_b >= X_{b+1} - X_b$$

Let the size of the jump instruction be jz bytes, and the time to execute the instruction be jt. The total time needed to fetch and execute these additional jump instructions is

$$trans(t_i, \mathcal{S}) = \sum_{b \in Mem_N(t_i)} freq_b \times (Y_b \times jcost_S + Z_b \times jcost_M)$$

$$jcost_S = jt + \left\lceil \frac{jz}{wd} \right\rceil \times lat_S \ \ ; \ \ jcost_M = jt + \left\lceil \frac{jz}{wd} \right\rceil \times lat_M$$

where $Mem_N(t_i) \subseteq Mem(t_i)$ represents the basic blocks of t_i that do not end with an unconditional jump.

Capacity Constraint. Finally, all blocks selected for allocation should fit into the scratchpad space of the host PE. Any additional jump instruction inserted into blocks allocated in the scratchpad should also be accounted for. Given the scratchpad size of cap_q attached to PE q, the capacity constraint is expressed as

$$\sum_{b=0}^{|Mem(t_i)|} (X_b \times area_b + Y_b \times jz) \ \leq \ space(t_i) \tag{4}$$

for each task t_i, and

$$\sum_{t_i : PE(t_i) = q} space(t_i) \ \leq \ cap_q \tag{5}$$

The ILP formulation is solved for X_b, Y_b and Z_b. The solution values for X_b indicate the actual selection of memory blocks for optimal allocation given current worst-case execution profile.

5.2 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 interval, from the earliest start time to the latest finish time among all tasks in the cluster.

```
1  Clusters := ∅
2  foreach task t in the application do
3      if ∃ C ∈ Clusters s.t.
         [C.Begin, C.End] ∩ [EarliestSt(t, S), LatestFin(t, S)] ≠ ∅ then
4          /* update C to include t */
5          C.Tasks := C.Tasks ∪ {t};
6          changed := FALSE;
7          if EarliestSt(t, S) < C.Begin then
8              C.Begin := EarliestSt(t, S); changed := TRUE;
9          if LatestFin(t, S) > C.End then
10             C.End := LatestFin(t, S); changed := TRUE;
11         /* check if inclusion of t affects the overall clustering */
12         if changed then
13             foreach cluster C' ∈ Clusters \ {C} do
14                 if [C.Begin, C.End] ∩ [C'.Begin, C'.End] ≠ ∅ then
15                     /* merge the two clusters */
16                     C.Tasks := C.Tasks ∪ C'.Tasks;
17                     C.Begin := min (C.Begin, C'.Begin);
18                     C.End := max (C.End, C'.End);
19                     Clusters := Clusters \ {C'};
20     else
21         /* create a new cluster C' for t */
22         C'.Tasks := {t};
23         C'.Begin := EarliestSt(t, S);
24         C'.End := LatestFin(t, S);
25         Clusters := Clusters ∪ {C'};
```

Algorithm 1: The *Interference Clustering (IC)* algorithm

The clustering is performed as detailed in Algorithm 1. We start with an empty cluster set (line 1). For each task, we try to find an existing cluster whose duration overlaps the lifetime of t (line 3). If there is no such cluster, we start a new cluster with t as the only member (lines 22–25). Otherwise, we add t into the existing cluster C and update C's duration (lines 5–10). If the duration of the cluster changes because of the new inclusion, we check against all other existing clusters whether any of them now overlaps with C, in which case they will be merged (lines 16–19).

After the clustering is decided, the same routine in *PK* that simultaneously optimizes partitioning and allocation is employed *within each cluster*. The ILP capacity constraint in Equation 5 is therefore modified to

$$\sum_{t_i \in C} space(t_i) \ \leq \ cap_q$$

for each cluster C of tasks executing on PE q. Two distinct clusters on q are guaranteed to have disjoint execution time intervals, thus the allocated memory blocks of tasks in a later cluster can completely replace the scratchpad content belonging to the previously executing cluster when the corresponding execution interval is entered.

The left part of Figure 12b shows the clustering decision for the given task schedule. Three clusters have been formed, and 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.

5.3 Graph Coloring (GC)

The *Interference Clustering (IC)* method is prone to produce large clusters due to transitivity. In Figure 12b, 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 considering task-to-task interference relations captured in the interference graph. If we apply *graph coloring* to this graph, the resulting assignment of 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, thus 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 [Welsh and Powell 1967], 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.

The above algorithm is illustrated in Figure 13, featuring the tasks that have been assigned to the same cluster by *IC* in our running example. The numbering in Figure 13a shows the order of traversal based on the degree of the nodes. The first iteration (Figure 13b) assigns the first color to all uncolored nodes with no neighbor of the same color, that is, fs_0 followed by fm_2. The next iteration (Figure 13c) assigns the second color to fr_0 and fr_1, and completes the coloring with a total of two colors used. This result dictates that the scratchpad space will be partitioned into two: one portion to be occupied by fm_2 and fs_0 during their respective lifetimes, and another portion to be occupied by fr_0 and fr_1 during their respective lifetimes.

Even though the graph coloring algorithm is a heuristic, in practice we observe that it does not pose serious sub-optimality to the solution, due to the fact that coloring options for actual task interference graphs are typically limited by the dependencies.

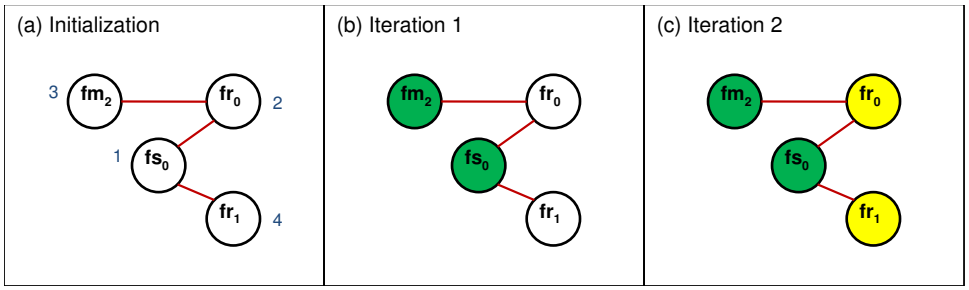


Fig. 13. Welsh-Powell algorithm for graph coloring, applied to the set of tasks clustered together in Figure 12(b)

After we obtain the color assignment, we formulate the scratchpad partitioning/allocation with the refined constraint (replacing Equation 4) that each task t_i with assigned color $color(t_i)$ can occupy at most the space allocated for $color(t_i)$, denoted by $space(color(t_i))$.

$$\sum_{b=0}^{|Mem(t_i)|} (X_b \times area_b + Y_b \times jz) \leq space(color(t_i))$$

In place of Equation 5, we now have the constraint that the scratchpad space given to all M_q colors used for PE q add up to its total scratchpad capacity cap_q:

$$\sum_{m=1}^{M_q} space(color(t_i)) \leq cap_q$$

Figure 12c shows the further partitioning within the second cluster previously formed by the *IC* scheme. fm_2 and fs_0 have been assigned the same color, and are allocated the same partition of the scratchpad to occupy at different time intervals. 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.

5.4 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 interval of the preempted task t must make allowance for the time spent waiting for the preempting task u to complete. If t is on the critical path of the application, it may be beneficial for the application WCRT to let t complete first before allowing u to start. By removing the interference between t and u, we will be able to apply scratchpad overlay on the both of them as well. The gain from allocation will potentially reduce the execution time of both, leading to further reduction in WCRT. However, the interference elimination requires delaying the start of task u and hence pushing back its entire lifetime, albeit with potentially reduced length after the updated scratchpad allocation. This delay may be propagated to other tasks with direct or indirect correlations with u. Therefore, we need to ensure that the decision will not adversely lead to an undesirable increase in the final WCRT.

```
1  repeat
2    /* Identify task interferences on current WCRT path P_0 */
3    CrIf := { (t,u) | t ∈ P_0 ∧ u ∈ intf(t) };
4    /* intf(t): set of higher priority tasks who may preempt t (Equation 2) */
5    /* Evaluate effect of eliminating each interference */
6    foreach interference pair (t,u) ∈ CrIf do
7      /* Cost: propagated slack */
8      slack := LatestFin(t,S) − LatestFin(u,S);
9      maxEnd := W_0;   /* current WCRT */
10     foreach task v that starts after u in current schedule do
11       if v depends on u then
12         maxEnd := max (maxEnd, LatestFin(v,S) + slack);
13       else if v ∉ intf(u) ∧ EarliestSt(v,S) < LatestFin(u,S) + slack
           then
14         vSlack := (LatestFin(u,S) + slack) − EarliestSt(v,S);
15         maxEnd := max (maxEnd, LatestFin(v,S) + vSlack);
16     /* Gain: removed preemption and potential gain via overlay */
17     preemptLen := LatestFin(u,S) − EarliestSt(u,S);
18     /* bestFit(x,y): set of most-accessed unallocated blocks of x to fit space(y) */
19     tGain := Σ_{b∈bestFit(t,u)} freq_b × area_b × (lat_M − lat_S);
20     uGain := Σ_{b∈bestFit(u,t)} freq_b × area_b × (lat_M − lat_S);
21     /* Projected WCRT if this interference is eliminated */
22     estW(t,u) := maxEnd − preemptLen − tGain − uGain;
23     if estW(t,u) > W_0 then
24       Remove (t,u) from CrIf;
25   /* Choose most beneficial interference to eliminate */
26   if CrIf ≠ ∅ then
27     Find (t_m,u_m) ∈ CrIf s.t. estW(t_m,u_m) ≤ estW(t,u) ∀(t,u) ∈ CrIf;
28     Set constraint EarliestSt(u_m,S) ≥ LatestFin(t_m,S);
29     Run WCRT analysis to propagate lifetime shift;
   until CrIf = ∅ ;
```

Algorithm 2: The *Critical Path Interference Reduction (CR)* algorithm

Our final proposed method proceeds as shown in Algorithm 2. We first work on the schedule produced by the WCRT analysis to improve the interference pattern. We consider all interferences in which tasks on the critical path are preempted or have to wait for tasks with higher priority (line 3). Each candidate is evaluated to find out the effect on overall WCRT if it is chosen to be eliminated. Interference among a task t and its preempting task u is eliminated by forcing u to wait until the completion of t, as illustrated in Figure 14b. The amount of slack introduced, $slack$, is thus the remaining execution time of t after the original preemption, $LatestFin(t,\mathcal{S}) - LatestFin(u,\mathcal{S})$ (line 8).

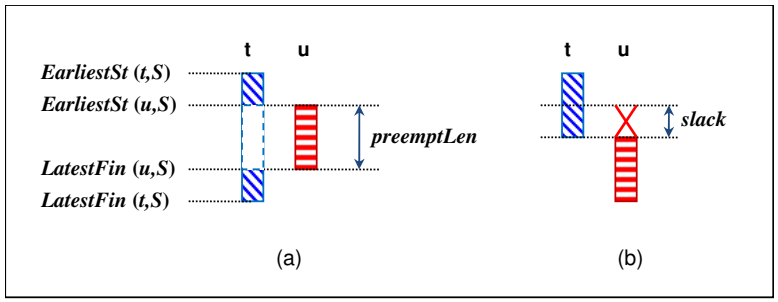


Fig. 14. Mechanism of slack insertion for interference elimination

We then examine all tasks that execute after u in the current schedule to see if the propagated slack pushes their lifetime beyond the current WCRT (line 9). The slack may affect tasks executing after u in two ways. First, if a task v is dependent on u (either as a direct successor or in the transitive closure of u's succeeding tasks), its start and completion time will also be pushed back by up to the same amount of slack (line 12). Second, if a task v is not dependent on u but is not interfering with u in the current schedule, then our iterative scratchpad algorithm will require us to maintain the non-interference among them. As the execution of u has been pushed back, it now completes only at $LatestFin(u, \mathcal{S}) + slack$. We check if this new lifetime interval intrudes into the lifetime of v. If necessary, the start and completion time of v will also be pushed back so that its lifetime remains not overlapping with u's (lines 13–15).

We move on to examine the potential gain from eliminating the candidate interference. The first source of gain is the removed preemption delay along the critical path (Figure 14a) which amounts to the length of the preempting task u's lifetime (line 17). The second source of gain is the potential time saving due to allocation, as we can now allow t and u to occupy more scratchpad space via overlay. This value is estimated by assuming t can place a selection of its unallocated code blocks into the scratchpad space currently occupied by u, and vice versa (lines 18–20). The selected blocks for this estimation are those with highest access frequency along the task's current WCET path, among all blocks that are currently not allocated in the scratchpad. Note that for the first iteration of this algorithm, the current scratchpad allocation refers to an empty allocation. This selection strategy is chosen for efficiency, and differs from the actual ILP-based content selection (as described in Section 5.1) in that it uses sub-optimal first-fit 'packing' of these most-accessed blocks, and it does not account for the transfer code needed between the scratchpad and main memory.

We can now estimate the application WCRT after the elimination of this candidate interference. It is the latest completion time over all tasks affected by the inserted slack, discounting the original preemption delay and the projected gain from allocation (line 22). If this estimated WCRT is larger than the current WCRT, then we conclude that it is not beneficial to remove this interference, and remove this candidate from consideration (lines 23–24). Note that this estimated WCRT is a heuristic that may not capture all actual impacts of eliminating the interference (for example, the delay on v at line 14 may trigger another series of delays following v). An accurate estimation will require invoking a full-fledged WCRT analysis for each candidate, on top of actually performing the contemplated changes, because updating allocation will potentially lead to new task lifetimes and

new task interaction patterns. The actual impact will be accurately determined after the considered elimination is selected and carried out later in this algorithm.

After evaluating the gain function for all candidate interferences along the critical path, we select one of them to remove. A sensible choice is the candidate with the best projected WCRT after removal (line 27). We eliminate this interference by forcing a delayed start for the preempting task (line 28), then propagate the shift to all tasks by re-running the WCRT analysis (line 29). 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 current critical path. We then proceed to perform scratchpad partitioning/allocation routine as used by the *Graph Coloring (GC)* scheme on this improved interference graph.

In Figure 12d, 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.

It is worth noting that this scheme is indeed a greedy heuristic, as it chooses the interference to eliminate based on best projected gain for the current worst-case path known at the point of decision. There remains a possibility that a decision with worse result in one iteration may lead to better improvement in future iterations, which may not be enabled in scenarios resulting from current better results. This makes the search for global optimum exponential.

6. EXPERIMENTS

6.1 Setup

We use two real-life embedded applications to evaluate the scracthpad allocation schemes that we have presented. Our first case study is the Unmanned Aerial Vehicle (UAV) control application from PapaBench [Nemer et al. 2006], a derivation from the real-time embedded UAV control software Paparazzi. We adapt the C source code of this application into a distributed implementation. The task programs are compiled for the SimpleScalar PISA architecture platform [Austin et al. 2002], and our analysis works on the resulting binaries.

The controller consists of two main functional units, `fly_by_wire` and `autopilot`, inter-connected by SPI serial link. The `fly_by_wire` unit is responsible for managing radio-command orders and servo-commands, while `autopilot` runs the navigation and stabilization tasks of the aircraft. The two components can operate in one of two modes. In the *manual* mode, `fly_by_wire` obtains navigation instructions via the radio, passes them through `autopilot` computation, then communicates the necessary actions to the servos. In the *automated* mode, `autopilot` reads information from the GPS unit to compute necessary adjustments, which are then passed to `fly_by_wire` to be actuated by the servos.

Figure 2, at the beginning of this paper, shows the active processes in one of the scenarios under the manual mode. This is the MSC we use in the experiments. The original implementation as shown uses 2 PEs with a total of 5KB scratchpad memory space. For the purpose of observation, we perform experiments for 1-PE, 2-PE, and 4-PE settings. We consider a homogeneous system where all the PEs are identical. The 1-PE case has all tasks assigned to the single PE. The 2-PE case follows the same division as the original implementation (`fly_by_wire`, `autopilot`) shown in Figure 2. The task scheduling in

PE_i	t_i	Codesize (bytes)	c_i (cycles)
1	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
1	fv_0	520	3,866
	fv_1	656	52,957
2	fr_0	384	2,646
	fr_1	4,552	28,008
2	fs_0	272	1,932
	fs_1	992	16,597
	fs_2	1,840	11,606
3	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
3	ad_0	352	2,442
	ad_1	2,296	13,345
	ad_2	6,496	36,374
4	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
4	ag_0	400	2,748
4	ar_0	5,520	34,944

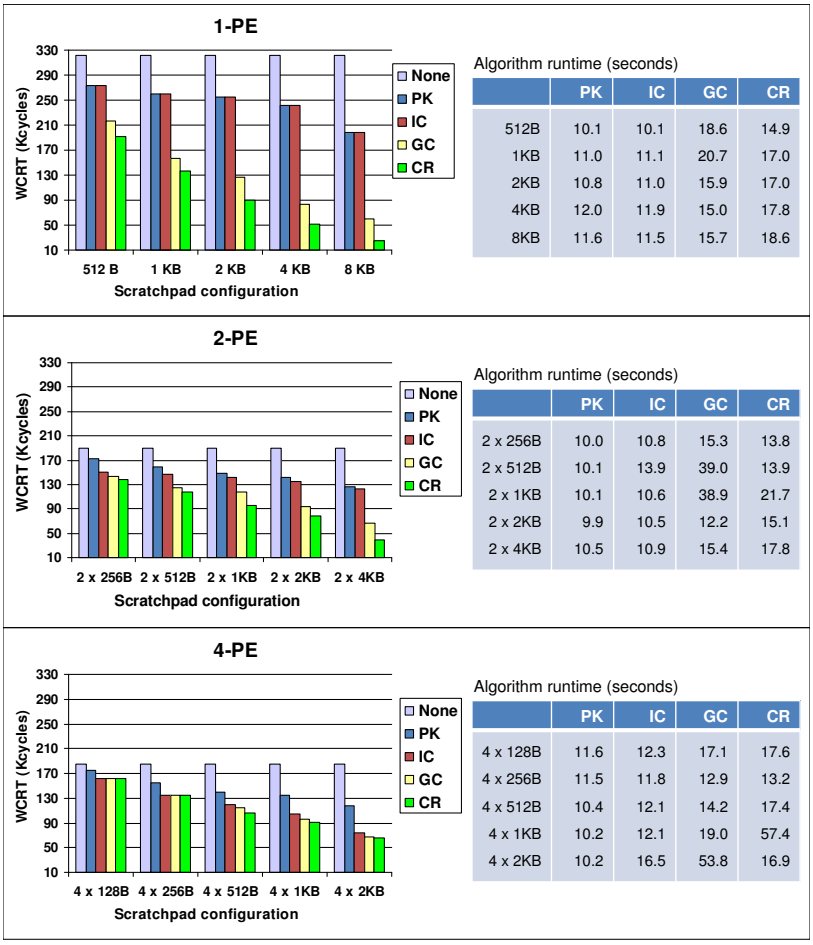
Fig. 15. Codesize and WCET of tasks in the PapaBench application

the 4-PE case is shown in Figure 15 along with the codesizes and uninterrupted worst-case runtimes of each task on the given architecture platform. The uninterrupted runtime values are obtained via WCET analysis of the program code assuming all instructions are fetched from the main memory. As the PEs are homogeneous in this setting, these values remain the same irrespective of PE assignment. We assume uniform execution time of 1 cycle per instruction, and compute the execution time of tasks as elaborated in Section 4.1. We assume fetch width of 16 bytes. It takes 1 cycle for a single fetch from the scratchpad memory. The off-chip memory latency is set at 100 cycles per fetch, given that real systems may have off-chip latencies ranging from 50 to over 100 cycles.

Total scratchpad size (for instructions) is varied from 512B to 8KB, distributed evenly among the PEs. The range is chosen to provide reasonable contention for memory space given the codesizes of the tasks. The ILP formulation for scratchpad content selection is solved using the tool CPLEX from ILOG [CPLEX 2002]. The solution determines scratchpad allocation for each scheme, which then becomes an input to the WCRT analysis (Section 4.2). The WCRT values obtained by the analysis for the different schemes are then compared.

6.2 Results and Discussion

Figure 16 shows the WCRT (in kilocycles) of the application for various scratchpad configurations. The first column shows the application WCRT without any allocation, that is, all memory blocks are fetched from the main memory. The next four columns show the final WCRT after applying the four discussed schemes respectively. 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. We see from the figure that the application WCRT without allocation reduces by a small margin as we go from 2 to 4 PEs, which implies a limit in the extent of paralellization due to task dependencies. This means that task interferences



	PK	IC	GC	CR
512B	10.1	10.1	18.6	14.9
1KB	11.0	11.1	20.7	17.0
2KB	10.8	11.0	15.9	17.0
4KB	12.0	11.9	15.0	17.8
8KB	11.6	11.5	15.7	18.6

	PK	IC	GC	CR
2 x 256B	10.0	10.8	15.3	13.8
2 x 512B	10.1	13.9	39.0	13.9
2 x 1KB	10.1	10.6	38.9	21.7
2 x 2KB	9.9	10.5	12.2	15.1
2 x 4KB	10.5	10.9	15.4	17.8

	PK	IC	GC	CR
4 x 128B	11.6	12.3	17.1	17.6
4 x 256B	11.5	11.8	12.9	13.2
4 x 512B	10.4	12.1	14.2	17.4
4 x 1KB	10.2	12.1	19.0	57.4
4 x 2KB	10.2	16.5	53.8	16.9

Fig. 16. WCRT of the benchmark application, before and after allocation by *Profile-based Knapsack (PK)*, *Interference Clustering (IC)*, *Graph Coloring (GC)*, and *Critical Path Interference Reduction (CR)*, along with algorithm runtime

do not change significantly between these two settings; however, the utilizable scratchpad space is smaller for each task, leading to decreased gain. Consequently, we see that the WCRT reductions achieved by the allocation schemes are generally worse.

When only 1 PE is utilized, most tasks are interfering with each other. The *Interference Clustering (IC)* method does not improve over the baseline *Profile-based Knapsack (PK)*, as the transitive interference places most tasks into the same space-sharing cluster. The *Graph Coloring (GC)* method performs much better than *IC* here, as it has a more refined view of interference relation among individual tasks. The *Critical Path Interference Reduction (CR)* method is, in turn, able to further improve the performance of *GC*.

When we move to the 2-PE setting, we see a drastic drop in application WCRT obtained by all schemes, as task interferences have reduced considerably. This confirms our observation that task interferences significantly influence application response time. With more

PEs employed, *IC* is able to perform better than *PK*, though the improvement is still not much, when compared to the reduction achieved by *GC* and *CR*. Again, *CR* emerges with the most performance gain in this setting.

As task interferences reduce even more in the 4-PE setting, the improvement by *GC* and *CR* also lessens, as the lack of refined task interaction knowledge is less damaging. With the reduced dominance of task interferences on performance here, we observe that relative performance gain follows more or less a normal curve along the scratchpad size axis – when the scratchpad space is very small, it gives little flexibility to all schemes. When the scratchpad space is large relative to task memory requirements, it allows reasonable performance gain even without overlay. The advantage of sophisticated account of task interaction is nevertheless pronounced for common cases in between these extremes.

From these results, we can conclude that the proposed scheme *CR* gives the best WCRT improvement over all other schemes. Averaged over all settings, the resulting application WCRT achieved by *CR* allocation is 42.2% lower than that achieved by the baseline *PK*. Individually, the reduction ranges from 7.9% to 87.6%. 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.

In all cases, when we compare the application WCRT without any allocation to the application WCRT after applying any of the allocation schemes (Figure 16), we see that scratchpad allocation indeed improves application WCRT tremendously. The simplest scheme, *PK*, is able to achieve 20.6% improvement. The gains from *IC* and *GC* are 24.9% and 45.3% respectively; and the best improvement by *CR* achieves 52.3% improvement. In particular, with a moderately-sized scratchpad, the gain from allocation is already larger than the gain from simply distributing the application on more PEs without allocation (compare for example *GC* on 1 PE with 1 KB scratchpad, with no-allocation on 4 PEs).

Finally, the tables of Figure 16 show the comparison of algorithm runtimes when run on a 3.0 Ghz Pentium 4 CPU with 1MB cache and 2GB memory. Our iterative scheme takes between 2–11 iterations to arrive at the final solution, where no further refinement is seen. As discussed in Section 4, the termination of the scheme is guaranteed. We observe a wide variation of runtime in certain cases, for example the runtime of *GC* in the 2-PE case. The dominant component of the algorithm runtime is the ILP solution time, which highly depends on the complexity of the problem. The variation in runtime arises from the variation in the complexity of the formulation due to different configurations and scratchpad sharing decisions. Nevertheless, the runtimes of all schemes as shown here are reasonably efficient, ranging from 10 seconds to a little less than a minute.

7. EXTENSION TO MESSAGE SEQUENCE GRAPH

Message Sequence Graph (MSG) is a finite state automaton where each state is described by an MSC. Multiple outgoing edges from a node in the MSG represent a non-deterministic choice, so that exactly one of the destination charts will be executed in succession. Figure 17 shows an example of an MSG, modeling the PapaBench application. The top left box shows the MSG, and the labeled boxes display the MSCs corresponding to the nodes. The MSC model used in the experiment presented earlier (Figure 2) corresponds to a single execution path through this MSG, that is, it shows only one of the possible scenarios represented by this MSG. While an MSC describes a single scenario in the system execu-

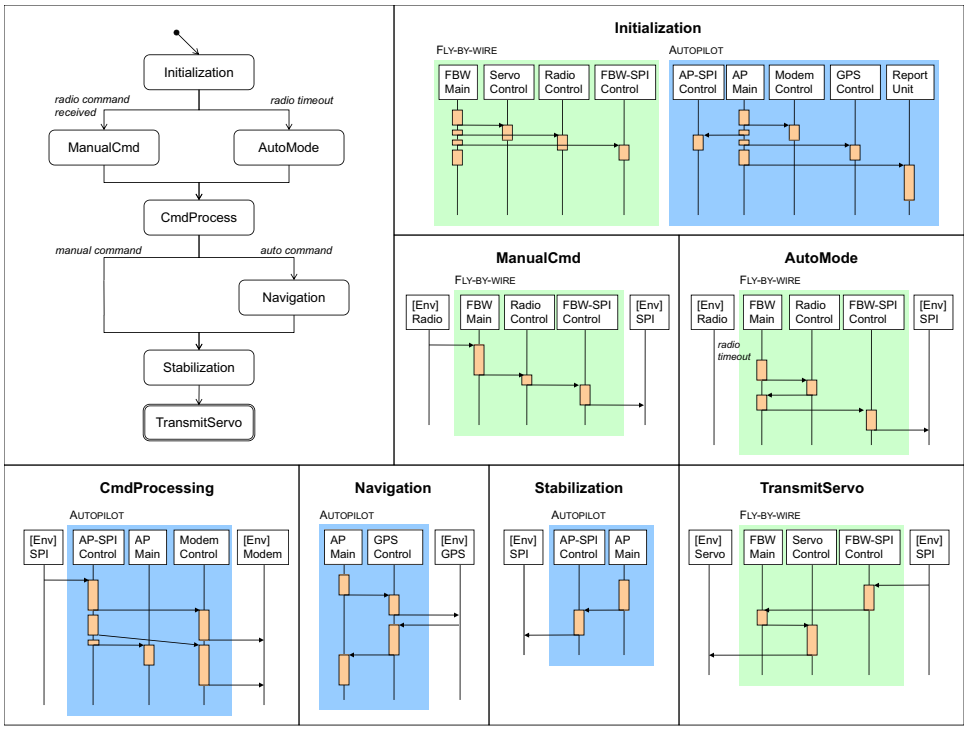


Fig. 17. Message Sequence Graph of the PapaBench application

tion, an MSG describes the control flow between these scenarios, allowing us to form a complete specification of the application. Therefore, to optimize the WCRT of the entire application, the analysis as well as the scratchpad allocation technique need to be extended to take into account the conditional control flow specified by the MSG model.

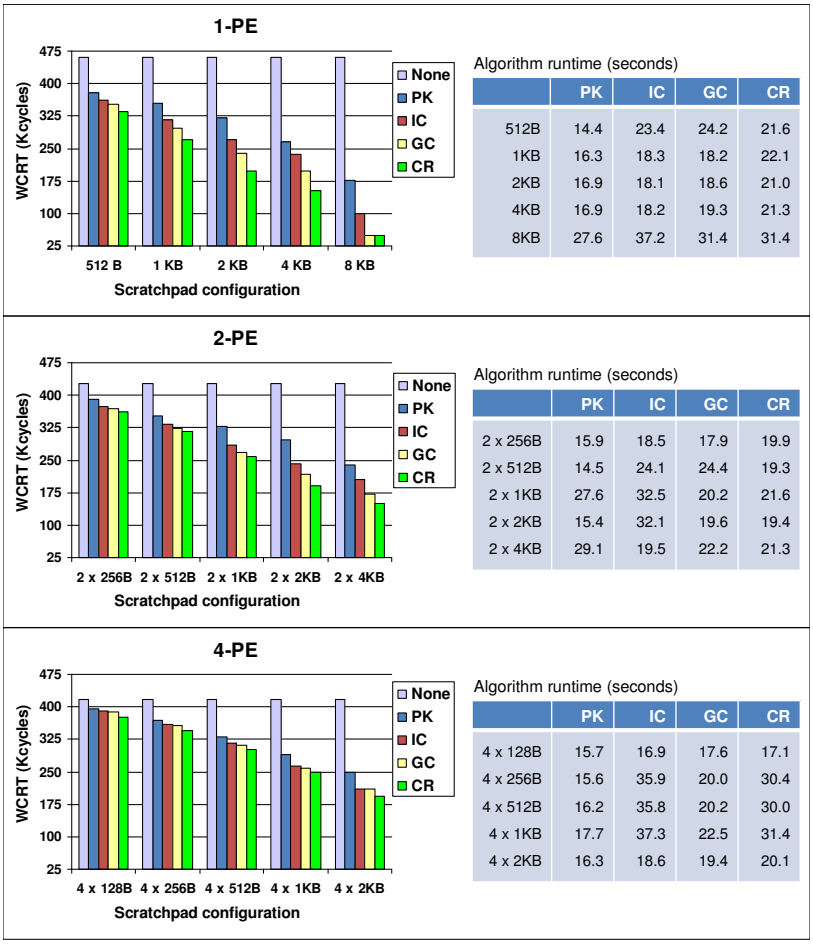
An execution of the modeled application traces a path in the MSG from an initial state to a terminal state (marked with double lining), and can be viewed as a concatenation of the MSCs along that path [Harel and Thiagarajan 2003]. Here, we consider the *synchronous* concatenation of the MSG, where all the tasks in an MSC (that is, an MSG node) must complete before any task in a subsequent MSC can execute. Each MSC can thus be viewed independently. The extended scratchpad allocation technique proceeds as follows.

(1) Perform scratchpad allocation on each MSC to obtain the WCRT-optimizing allocation along with the post-allocation WCRT value for each MSG node.

(2) Traverse the MSG to find the longest path in terms of total post-allocation WCRT.

(3) Compose the scratchpad allocation of the MSG nodes according to their execution order along the WCRT path.

The MSG may contain loops between the initial state and the terminal state, in which case we require that the maximum number of times the cyclic edges can be traversed is known, so that the longest path is well-defined. The WCRT of the application is then the sum of the WCRT of the MSCs along the longest path, and the worst-case-optimizing



Algorithm runtime (seconds)

	PK	IC	GC	CR
512B	14.4	23.4	24.2	21.6
1KB	16.3	18.3	18.2	22.1
2KB	16.9	18.1	18.6	21.0
4KB	16.9	18.2	19.3	21.3
8KB	27.6	37.2	31.4	31.4

Algorithm runtime (seconds)

	PK	IC	GC	CR
2 x 256B	15.9	18.5	17.9	19.9
2 x 512B	14.5	24.1	24.4	19.3
2 x 1KB	27.6	32.5	20.2	21.6
2 x 2KB	15.4	32.1	19.6	19.4
2 x 4KB	29.1	19.5	22.2	21.3

Algorithm runtime (seconds)

	PK	IC	GC	CR
4 x 128B	15.7	16.9	17.6	17.1
4 x 256B	15.6	35.9	20.0	30.4
4 x 512B	16.2	35.8	20.2	30.0
4 x 1KB	17.7	37.3	22.5	31.4
4 x 2KB	16.3	18.6	19.4	20.1

Fig. 18. WCRT of the complete PapaBench application, before and after allocation by *Profile-based Knapsack (PK)*, *Interference Clustering (IC)*, *Graph Coloring (GC)*, and *Critical Path Interference Reduction (CR)*, along with algorithm runtime

scratchpad allocation is the composition of the allocation decisions for the MSCs according to their execution sequence on that path. This composition is basically a chart-level scratchpad overlay among the independent MSCs, justified by definition of the synchronous concatenation.

We run the extended scratchpad allocation on the PapaBench MSG (Figure 17). The assignment of processes to PEs as well as latency settings are the same as in our experiments on the single MSC, presented in the previous section. Figure 18 shows the resulting WCRT after allocation by the four schemes, compared with the WCRT without allocation. Here, task interactions are limited within each MSG node. Again, we observe that while there is a little reduction in WCRT without allocation from the 2-PE to the 4-PE case, most of the allocation results in the 4-PE settings do not gain as much as they do in the 2-PE settings. This is due to the fact that the PEs are not necessarily fully utilized in each node,

as processes have been mapped to PEs according to functionality without regard for load balancing (recall the scheduling policy described in Section 3.1). For example, in the 2-PE case where one PE takes up `fly_by_wire` tasks and the other executes `autopilot` tasks, then only one of them is active in MSG nodes other than `Initialization`. As we have observed earlier, for the same total scratchpad space, utilizing more PEs narrows down the available space on each PE, and tasks may contrarily take longer time to complete.

The results in terms of post-allocation WCRT of the complete application still confirm our hypothesis that allocation techniques with more sophisticated view of task interactions obtain better results. In particular, our proposed scheme *CR* still gives the best WCRT improvement. In a similar trend as in the single MSC case, the improvements are more pronounced in configurations with less number of PEs and hence heavier task interferences. All four schemes achieve large improvement in WCRT compared to the case without any allocation, with the gain ranging from 25.9% for *PK* up to 41.3% for *CR*.

As charts are examined individually, the method does not experience noticable scalability problem. The tables in Figure 18 shows the time required by the schemes to produce the allocation for this application, which consists of 7 individual MSCs with varying degree of complexity. When we compute the individual increase in algorithm runtime compared to the corresponding runtime for the single MSC model presented in Figure 16 (that is, comparing the time taken by the same scheme for the same scratchpad configuration), we see only about 50% increase across all schemes and settings.

8. METHOD SCALABILITY

To evaluate the scalability of our proposed technique, we run it on a more complex application adapted from DEBIE-I DPU Software [European Space Agency 2008], an in-situ space debris monitoring instrument developed by Space Systems Finland Ltd. We model the software as an MSG, shown in Figure 19. The main functions involved have been broken down into several concurrent processes.

The DEBIE instrument utilizes up to four Sensor Units to detect particle impacts on the spacecraft. As the system starts up, it performs initializations and runs tests of the main functionalities. The system then enters the Standby state. When the command to start the acquisition of impact data is received via the Telecommand handler, the system goes into the Acquisition state and turns on at least one of the Sensor Units. In this mode, each particle impact will trigger a series of measurement, and the data are classified and logged for further transmission to the ground station. Data acquisition will continue until the stopping command is received, after which the system returns to the Standby state. In either mode, the Health Monitoring process periodically monitors the health of the instrument and runs housekeeping checks. If any error is detected, the system will reboot.

The codesize and WCET of each task as well as its mapping to 4 PEs are listed in Figure 20. When running the experiments for 2 PEs, the tasks in PE_1 and PE_2 are assigned to the first PE, and the rest of the tasks are assigned to the second PE. As before, the WCET values have been computed assuming all accesses are directed to the main memory, but we assume an off-chip latency of 10 cycles instead of 100 cycles (as in the previous setting). This adjustment has the effect of scaling down the difference between the shortest-running task and the longest-running task in this application, and reduces the skew in evaluation. As apparent from Figure 20, major tasks in this application have runtimes that are several orders of magnitude larger than the previous PapaBench benchmark, even with this

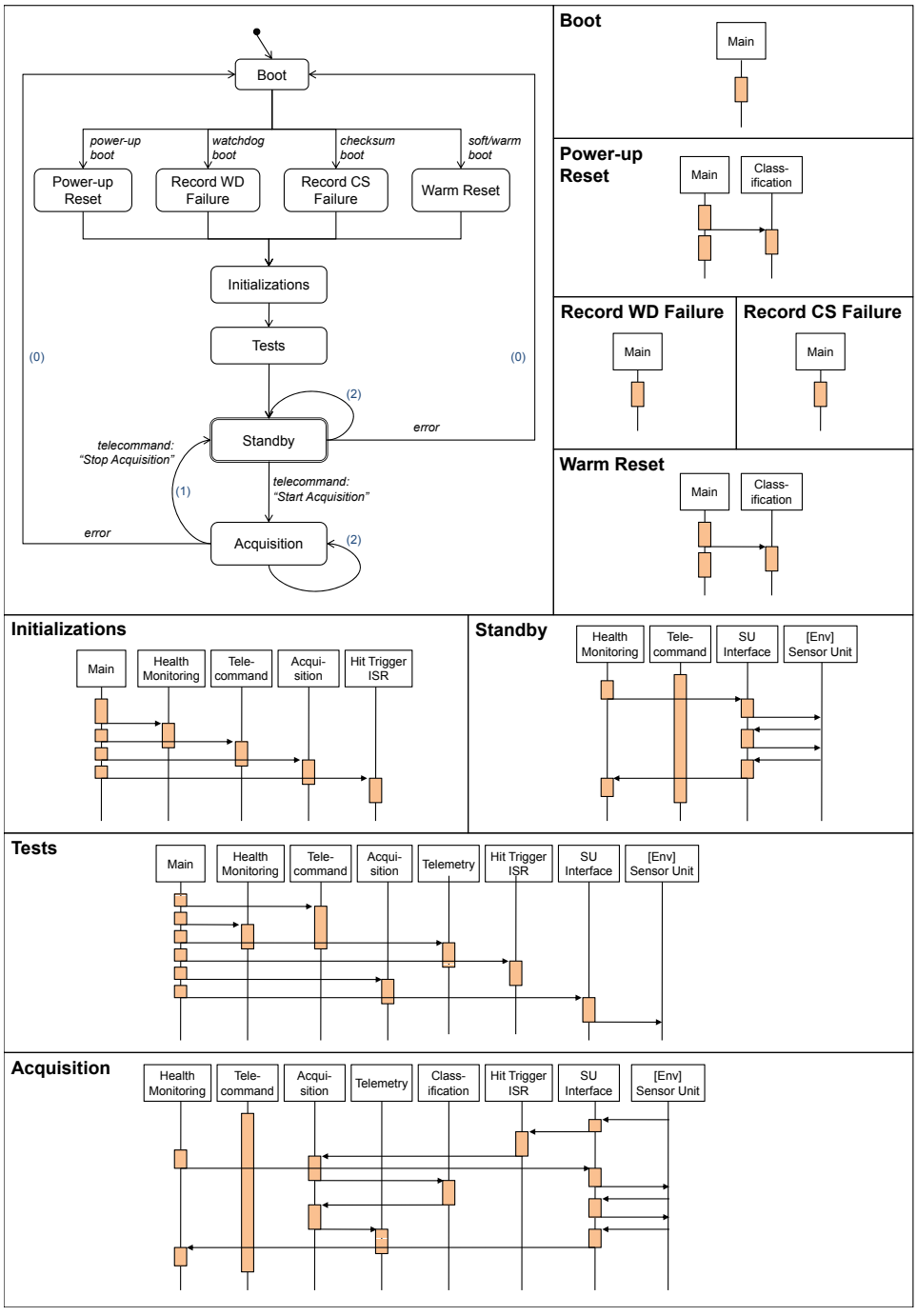


Fig. 19. Message Sequence Graph of the DEBIE application

PE_i	t_i	Codesize (bytes)	c_i (cycles)
1	**mn-boot**	3,200	4,325
	mn-pw$_1$	9,456	14,589
	mn-pw$_2$	3,472	6,784
	mn-wd	3,400	6,745
	mn-cs	3,400	6,745
	mn-wr$_1$	3,408	10,757
	mn-wr$_2$	5,952	6,594
	mn-ini$_1$	320	260
	mn-ini$_2$	376	271
	mn-ini$_3$	376	271
	mn-ini$_4$	376	271
	mn-tst$_1$	240	180
	mn-tst$_2$	240	180
	mn-tst$_3$	240	180
	mn-tst$_4$	240	180
	mn-tst$_5$	240	180
	mn-tst$_6$	240	180
1	**tm-tst**	56,960	839,607
	tm-acq	3,768	5,560
2	**ht-ini**	616	507
	ht-tst	10,776	105,224,302
	ht-acq	8,016	1,008,155
2	**su-tst**	50,176	15,606,989
	su-sby$_1$	6,512	103,375,726
	su-sby$_2$	4,392	51,684,932
	su-sby$_3$	1,320	91,593
	su-acq$_0$	2,536	712
	su-acq$_1$	6,512	103,375,726
	su-acq$_2$	4,392	51,684,932
	su-acq$_3$	1,320	91,593

PE_i	t_i	Codesize (bytes)	c_i (cycles)
3	**cl-pw**	1,648	1,266
	cl-wr	1,648	1,266
	cl-acq	3,064	14,637
3	**tc-ini**	4,408	3,341
	tc-tst	45,368	38,009,988
	tc-sby	23,288	117,268
	tc-acq	23,288	117,268
4	**aq-ini**	200	165
	aq-tst	44,128	126,996,985
	aq-acq$_1$	3,136	2,400
	aq-acq$_2$	3,024	2,688
4	**hm-ini**	5,224	155,055,938
	hm-tst	44,176	743,373,112
	hm-sby$_1$	16,992	413,497,626
	hm-sby$_2$	448	343
	hm-acq$_1$	16,992	413,497,626
	hm-acq$_2$	448	343

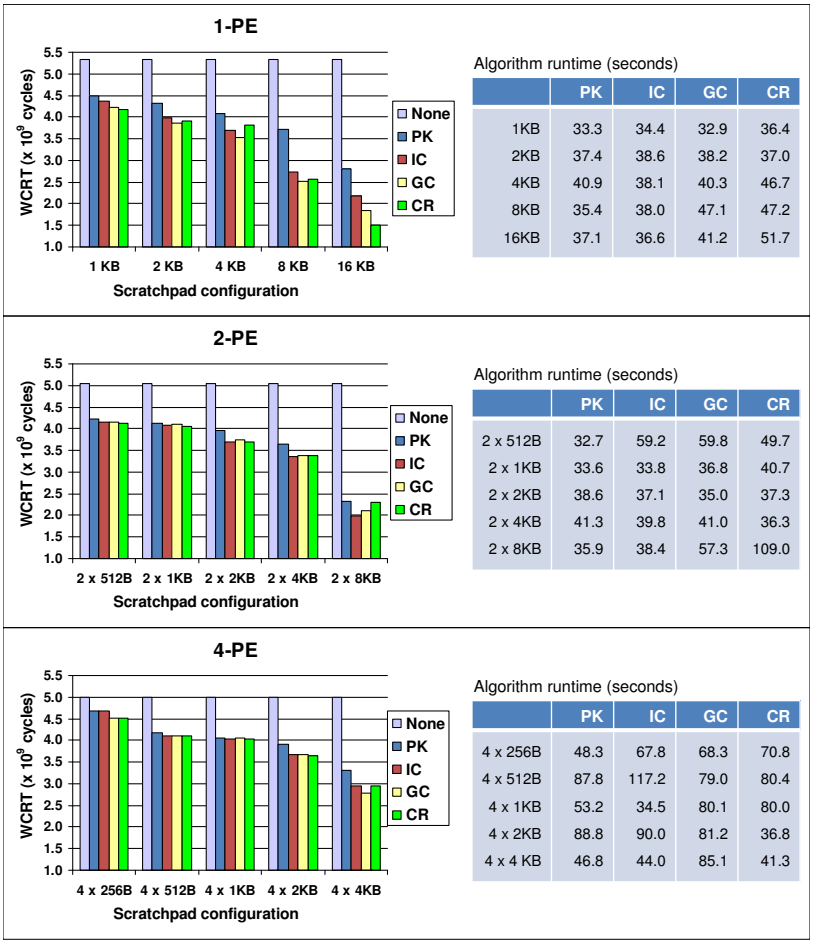
Fig. 20. Codesize and WCET of tasks in the DEBIE application

adjustment. The scratchpad latency remains at 1 cycle. Following the increase in average codesize of the tasks, the total scratchpad size is now varied from 1 KB to 16 KB.

The DEBIE application model as shown in Figure 19 is a cyclic MSG, and we have indicated the bounds used for our experiments in the brackets labeling the cyclic edges. Each specified bound is an absolute count of the number of times the edge is taken in a complete scenario according to the MSG.

The optimization results in terms of post-application WCRT for the MSG along with algorithm runtimes for all four schemes are shown in Figure 21, alongside the WCRT without allocation. Overall, the allocation schemes improve performance by 24–31% compared to the case without allocation.

As we compare the four allocation schemes, we observe two interesting phenomena. First, we see that *GC* may sometimes perform worse than *IC* (for example, in the 2-PE, 2 x 2KB scratchpad configuration). In these cases, the performance flop is caused by the decision to allocate less space to a color which is used by only one task, which then becomes the critical task in the particular chart. Recall that the allocation of scratchpad space and the content selection are simultaneously performed to optimize for the approximated definition of total WCRT. As coloring imposes a 'grouping' among tasks, the evaluation of the gain function may be biased towards a group with more tasks, moreover when it is difficult to predict which tasks may turn out critical after all task interactions are considered.



Algorithm runtime (seconds)

	PK	IC	GC	CR
1KB	33.3	34.4	32.9	36.4
2KB	37.4	38.6	38.2	37.0
4KB	40.9	38.1	40.3	46.7
8KB	35.4	38.0	47.1	47.2
16KB	37.1	36.6	41.2	51.7

Algorithm runtime (seconds)

	PK	IC	GC	CR
2 x 512B	32.7	59.2	59.8	49.7
2 x 1KB	33.6	33.8	36.8	40.7
2 x 2KB	38.6	37.1	35.0	37.3
2 x 4KB	41.3	39.8	41.0	36.3
2 x 8KB	35.9	38.4	57.3	109.0

Algorithm runtime (seconds)

	PK	IC	GC	CR
4 x 256B	48.3	67.8	68.3	70.8
4 x 512B	87.8	117.2	79.0	80.4
4 x 1KB	53.2	34.5	80.1	80.0
4 x 2KB	88.8	90.0	81.2	36.8
4 x 4 KB	46.8	44.0	85.1	41.3

Fig. 21. WCRT of the DEBIE application, before and after allocation by *Profile-based Knapsack (PK)*, *Interference Clustering (IC)*, *Graph Coloring (GC)*, and *Critical Path Interference Reduction (CR)*, along with algorithm runtime

Secondly, there are also cases where *CR* results in worse performance than *GC* or *IC* (for example, in the 1-PE, 4KB scratchpad configuration). Investigation reveals that this deterioration happens over the iterative improvements for certain charts in the application (recall that allocation is done independently on each chart). The decision to eliminate selected interferences does yield better reduction in the first few iterations of the allocation scheme. However, as it continues, it hits a point when the refined allocation/content selection does not improve the WCRT, and thus the iteration ends. Meanwhile, the *GC* or *IC* method, while achieving longer WCRT at first, continues to refine the allocation decision over the iteration and finally reaches a better overall WCRT. The choice to eliminate certain interferences certainly caters to the application WCRT as best as it can predict based on the critical path at that moment, yet the non-interference constraint that is carried forward to the next iterations restricts the 'movement' of tasks in the application. In these cases,

the shifting of tasks after the iterative improvement ultimately gives a chance to *GC* and *IC* to obtain more beneficial allocation even with less overlay.

As far as scalability is concerned, we see that all schemes can complete their computation below 2 minutes, even though the charts to be optimized are considerably more complex than the previous PapaBench benchmark. The larger codesizes imply increased complexity in the ILP formulation for scratchpad content selection, as the application has more task code blocks to consider for optimal allocation. We observe that the sizes of the ILP formulation for the DEBIE benchmark is roughly 25 times the sizes of the ILP formulation for the PapaBench benchmark. The longer runtimes also give larger windows for more interferences to occur. Despite these, the methods are able to provide results in reasonable time. We can therefore conclude that there is no evident scalability problem in all four schemes.

9. EXTENSION AND DISCUSSION

In this work, 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 (1) communication or interaction among the threads or processes of the application, as well as (2) interference among the threads or processes due to preemptive scheduling in the processing elements. The Message Sequence Chart (MSC) model is chosen as it shows the process interaction explicitly. 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 on two real-world embedded case studies. The achieved gain ranges from 20% by the simplest scheme to 52% by the most sophisticated scheme.

We have also presented an extension of our scheme to the Message Sequence Graph (MSG) model, where charts labeling the nodes in the graph are concatenated synchronously to form a complete execution scenario. Alternatively, the charts can be concatenated *asynchronously*. In this case, a task in an MSC may start as soon as all tasks of the same process in the preceding MSC have completed. The result of an asynchronous concatenation forms an MSC, while it is not necessarily so in the synchronous case.

Our scratchpad allocation method can be extended for the case of asynchronous concatenation as follows. By enumerating all paths through the MSG, we obtain all possible execution scenarios of the application. If the MSG contains loops, then the bounds on the edge counts should also be supplied to enable the enumeration of all possible paths. Each path is formed by asynchronously concatenating the MSCs, thus resulting in an MSC. Our method then performs WCRT analysis along with scratchpad allocation for each resulting MSC as before. The path/MSC with the maximum WCRT value then determines the WCRT of the application, and the allocation decision corresponding to that MSC is the worst-case-optimizing scratchpad allocation for the application. The complexity of the asynchronous concatenation method lies in the enumeration of the paths. The WCRT analysis and scratchpad allocation operate on each result of concatenation as a stand-alone MSC. As such, in our evaluation of this setting, the trend in results is similar to the single-MSC case, that is, the application WCRT improves as more refined view of task interactions is employed in the allocation schemes.

As a concluding note, this paper has focused on inter-task dynamic allocation, where a task loads its contents into the scratchpad once at the start of execution. Our proposed scheme manages the scratchpad layout in such a way that the space assigned to the task will not be subject to interference by others as long as it has not terminated. It is clearly possible to further let the task manage this reserved space for better utilization, at the cost of increased complexity. A natural extension is to allow intra-task dynamic scratchpad allocation – on top of the presented inter-task overlay scheme – where a task can replace the content it has in the scratchpad at its own discretion. The decision scheme for overlay content and designated points of reload within the task can be accounted in the WCET analysis of the task, thus preserving timing predictability.

ACKNOWLEDGMENTS

This work was partially supported by NUS research project "Platform-aware Timing Analysis of Behavioral System Models" (R252-000-321-112).

REFERENCES

ALUR, R. AND YANNAKAKIS, M. 1999. Model checking message sequence charts. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR)* (August 1999), J. C. BAETEN and S. MAUW, Eds. Springer-Verlag, London, UK, 114–129.

ANGIOLINI, F., MENICHELLI, F., FERRERO, A., BENINI, L., AND OLIVIERI, M. 2004. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. ACM, New York, NY, USA, 259–267.

AUSTIN, T., LARSON, E., AND ERNST, D. 2002. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer 35,* 2, 59–67.

AVISSAR, O., BARUA, R., AND STEWART, D. 2002. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems (TECS) 1,* 1, 6–26.

BANAKAR, R., STEINKE, S., LEE, B.-S., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES)*. ACM, New York, NY, USA, 73–78.

CPLEX. 2002. The ILOG CPLEX Optimizer v7.5. Commercial software, `http://www.ilog.com`.

DEVERGE, J.-F. AND PUAUT, I. 2007. WCET-directed dynamic scratchpad memory allocation of data. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE Computer Society, Washington, DC, USA, 179–190.

DOMINGUEZ, A., UDAYAKUMARAN, S., AND BARUA, R. 2005. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing 1,* 4, 521–540.

EGGER, B., KIM, C., JANG, C., NAM, Y., LEE, J., AND MIN, S. L. 2006. A dynamic code placement technique for scratchpad memory using postpass optimization. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. ACM, New York, NY, USA, 223–233.

EGGER, B., LEE, J., AND SHIN, H. 2008. Dynamic scratchpad memory management for code in portable systems with an MMU. *ACM Transactions on Embedded Computing Systems (TECS) 7,* 2, 1–38.

EUROPEAN SPACE AGENCY. 2008. DEBIE – First standard space debris monitoring instrument. Available at: `http://gate.etamax.de/edid/publicaccess/debie1.php`.

FALK, H. AND VERMA, M. 2004. Combined data partitioning and loop nest splitting for energy consumption minimization. In *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, H. SCHEPERS, Ed. Springer, Berlin/Heidelberg, 137–151.

HAREL, D. AND THIAGARAJAN, P. S. 2003. Message sequence charts. *UML for Real: Design of Embedded Real-time Systems*, 77–105.

ISSENIN, I., BROCKMEYER, E., DURINCK, B., AND DUTT, N. 2006. Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In *Proceedings of the 43rd Design Automation Conference (DAC)*. ACM, New York, NY, USA, 49–52.

ITU-T. 1996. Recommendation Z.120: Message Sequence Chart (MSC). *ITU-T, Geneva*.

JANAPSATYA, A., IGNJATOVIC, A., AND PARAMESWARAN, S. 2006. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proceedings of the 2006 Conference on Asia South Pacific Design Automation (ASP-DAC)*. IEEE Press, Piscataway, NJ, USA, 612–617.

KANDEMIR, M. 2007. Data locality enhancement for CMPs. In *Proceedings of the 2007 International Conference on Computer-Aided Design (ICCAD)*. IEEE Press, Piscataway, NJ, USA, 155–159.

KANDEMIR, M., KADAYIF, I., AND SEZER, U. 2001. Exploiting scratch-pad memory using presburger formulas. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS)*. ACM, New York, NY, USA, 7–12.

KANDEMIR, M., OZTURK, O., AND KARAKOY, M. 2004. Dynamic on-chip memory management for chip multiprocessors. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. ACM, New York, NY, USA, 14–23.

KANDEMIR, M., RAMANUJAM, J., AND CHOUDHARY, A. 2002. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *Proceedings of the 39th Design Automation Conference (DAC)*. ACM, New York, NY, USA, 219–224.

KANDEMIR, M., RAMANUJAM, J., IRWIN, M. J., VIJAYKRISHNAN, N., KADAYIF, I., AND PARIKH, A. 2004. A compiler based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 23,* 2 (February), 243–260.

LEE, C.-G., HAHN, J., SEO, Y.-M., MIN, S. L., HA, R., HONG, S., PARK, C. Y., LEE, M., AND KIM, C. S. 1998. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers 47,* 6, 700–713.

LI, X., LIANG, Y., MITRA, T., AND ROYCHOUDHURY, A. 2007. Chronos: A timing analyzer for embedded software. *Science of Computer Programming 69,* 1-3, 56–67. `http://www.comp.nus.edu.sg/~rpembed/chronos/`.

MARWEDEL, P., WEHMEYER, L., VERMA, M., STEINKE, S., AND HELMIG, U. 2004. Fast, predictable and low energy memory references through architecture-aware compilation. In *Proceedings of the 2004 Conference on Asia South Pacific Design Automation (ASP-DAC)*. IEEE Press, Piscataway, NJ, USA, 4–11.

MITRA, T. AND ROYCHOUDHURY, A. 2007. Worst case execution time and energy analysis. In *The Compiler Design Handbook: Optimizations and Machine Code Generation, 2nd Ed.*, Y. SRIKANT and P. SHANKAR, Eds. CRC Press, Boca Raton, FL, USA, Chapter 1.

NEGI, H. S., MITRA, T., AND ROYCHOUDHURY, A. 2003. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, New York, NY, USA, 201–206.

NEMER, F., CASS, H., SAINRAT, P., BAHSOUN, J.-P., AND MICHIEL, M. D. 2006. PapaBench: A free real-time benchmark. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, F. MUELLER, Ed. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. Available at: `http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id\_rubrique=97`.

NGUYEN, N., DOMINGUEZ, A., AND BARUA, R. 2005. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. ACM, New York, NY, USA, 115–125.

OZTURK, O., KANDEMIR, M., AND KOLCU, I. 2006. Shared scratch-pad memory space management. In *Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED)*. IEEE Computer Society, Washington, DC, USA, 576–584.

PANDA, P. R., DUTT, N. D., AND NICOLAU, A. 2000. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 5,* 3, 682–704.

PUAUT, I. 2006. WCET-centric software-controlled instruction caches for hard real-time systems. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE Computer Society, Washington, DC, USA, 217–226.

RAVINDRAN, R. A., NAGARKAR, P. D., DASIKA, G. S., MARSMAN, E. D., SENGER, R. M., MAHLKE, S. A., AND BROWN, R. B. 2005. Compiler managed dynamic instruction placement in a low-power code cache. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, Washington, DC, USA, 179–190.

STASCHULAT, J. AND ERNST, R. 2004. Multiple process execution in cache related preemption delay analysis. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT)*. ACM, New York, NY, USA, 278–286.

STEINKE, S., GRUNWALD, N., WEHMEYER, L., BANAKAR, R., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS)*. ACM, New York, NY, USA, 213–218.

STEINKE, S., WEHMEYER, L., LEE, B. S., AND MARWEDEL, P. 2002. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the 2002 Design, Automation and Test in Europe (DATE)*. IEEE Computer Society, Washington, DC, USA, 409.

SUHENDRA, V., MITRA, T., AND ROYCHOUDHURY, A. 2006. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the 43rd Design Automation Conference (DAC)*. ACM, New York, NY, USA, 358–363.

SUHENDRA, V., MITRA, T., ROYCHOUDHURY, A., AND CHEN, T. 2005. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, Washington, DC, USA, 223–232.

SUHENDRA, V., RAGHAVAN, C., AND MITRA, T. 2006. Integrated scratchpad memory optimization and task scheduling for MPSoC architectures. In *Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. ACM, New York, NY, USA, 37–42.

TOMIYAMA, H. AND DUTT, N. D. 2000. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*. ACM, New York, NY, USA, 67–71.

UDAYAKUMARAN, S. AND BARUA, R. 2003. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. ACM, New York, NY, USA, 276–286.

VERMA, M., PETZOLD, K., WEHMEYER, L., FALK, H., AND MARWEDEL, P. 2005. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *Proceedings of the 3rd Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, M. MIRANDA and S. HA, Eds. IEEE Computer Society, Washington, DC, USA, 115–120.

VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004a. Cache-aware scratchpad allocation algorithm. In *Proceedings of the 2004 Design, Automation and Test in Europe (DATE)*. IEEE Computer Society, Washington, DC, USA, 21264.

VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004b. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, New York, NY, USA, 104–109.

WEHMEYER, L., HELMIG, U., AND MARWEDEL, P. 2004. Compiler-optimized usage of partitioned memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI)*. ACM, New York, NY, USA, 114–120.

WELSH, D. J. A. AND POWELL, M. B. 1967. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal 10,* 1, 85–87.

YEN, T.-Y. AND WOLF, W. 1998. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 9,* 11, 1125–1136.