

Static Bus Schedule aware Scratchpad Allocation in Multiprocessors

Sudipta Chattopadhyay

National University of Singapore
sudiptac@comp.nus.edu.sg

Abhik Roychoudhury

National University of Singapore
abhik@comp.nus.edu.sg

Abstract

Compiler controlled memories or scratchpad memories offer more predictable program execution times than cache memories. Scratchpad memories are often employed in multi-processor system-on-chip (MPSoC) platforms which seek to meet the performance needs of embedded applications while limiting power consumption and timing unpredictability. Scratchpad allocation schemes optimize performance while ensuring predictable execution times (as compared to caches).

In this work, we develop a compile-time scratchpad allocation framework for multi-processor platforms, where the processors (virtually) share on-chip scratchpad space and external memory is accessed through a shared bus. Our allocation method considers the waiting time for bus access while deciding which memory blocks to load into the shared scratchpad memory space. Incorporating the bus schedule into our scratchpad allocation method leads to a global optimization of an application, as compared to employing local scratchpad allocation schemes in individual processors which locally optimize the per-processor execution time. We evaluate the efficacy, sensitivity and efficiency of our memory allocation scheme on two real-world embedded applications - an application controlling an Unmanned Aerial Vehicle (UAV), and a (fragment of) an in-orbit spacecraft software.

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

General Terms Design, Performance

Keywords MPSoC, WCET, Scratchpad, Shared bus

1. Introduction

Scratchpad memory (SPM) is a fast on-chip memory where the content of the scratchpad is controlled by the compiler and/or managed explicitly by the user. Therefore, the cost of each memory access is predictable in presence of SPM. Due to this predictability, scratchpads have been widely adopted for real-time embedded software design instead of *caches* where the memory management is entirely transparent to the user/compiler. However, explicit memory management by user is cumbersome and error-prone. Thus ex-

tensive compiler support is required for the content selection into scratchpad memories.

In this paper, we study content selection in shared scratchpad memories for multi-processors system on chip (MPSoC) running concurrent embedded softwares. Our goal is to reduce the overall *worst case response time* (WCRT) of the application, represented as a set of task graphs. MPSoCs usually contain an on-chip scratchpad memory attached *locally* to each processing element (PE). However, a particular PE can also access other PEs' SPMs *remotely*. On the other hand, the external (off-chip) memory is accessed through a shared bus among all the available processors in the chip.

Clearly, a processing element incurs a variable amount of delay to access the shared bus due to the bus contention introduced by other PEs. Since the requests serviced from on-chip SPMs do not access the off-chip shared bus, shared bus traffic depends on the content selection into the SPMs. On the other hand, content selection into an SPM depends on the *latency* incurred by a main memory access which in turn depends on the *waiting time* to access the shared bus. The inter-dependency between bus contention and scratchpad allocation motivates us to develop a new SPM allocation technique. Our SPM allocation method incorporates the bus schedule and hence results in a global performance optimization of the application. For the shared bus, we assume a static bus schedule using a Time Division Multiple Access (TDMA) scheme. Processors are statically assigned bus slots and the bus slots are allocated among the PEs in a round robin fashion. An integrated SPM allocation framework that considers the timing effects of shared bus in multi-processor platforms is the main contribution of the paper.

To develop such an integrated SPM allocation framework for multi-processors, we face many technical challenges. Since the SPM space is shared among multiple PEs, it is important to use the shared scratchpad space as much as possible for all the *critical tasks* (*i.e.* all tasks lying in the critical path of the application) which are responsible for higher WCRT. On the other hand, if there are two processing elements PE_1, PE_2 and we fill up the shared SPM by randomly placing items from the critical tasks of PE_1 , it may drastically limit the WCRT improvement of the application if the tasks running on PE_2 are also critical. Our global optimization scheme creates a unified view of all the items accessed in different processors and iteratively allocates the item(s) suffering from highest latencies to access the off-chip memory. We also employ an optimization where variables from different independent tasks may share the same SPM space through *overlay* due to their disjoint lifetimes. This leads us to more utilization of available shared SPM space.

Our allocation technique is iterative and we have used a *cycle accurate WCRT analyzer* to evaluate our approach. Our case study with real-life embedded applications such as an Unmanned Aerial Vehicle (UAV) controller and an in-orbit spacecraft software reveals that we can obtain significant WCRT reductions by

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCIES'11, April 11–14, 2011, Chicago, Illinois, USA.
Copyright © 2011 ACM 978-1-4503-0555-6/11/04...\$10.00

appropriate content selection and *overlay* in SPM. We have also compared our approach with existing scratchpad allocation scheme which locally optimizes the per-processor execution time without being aware of variable bus delays. We have found that our approach can further improve the WCRT upto 70% compared to local scratchpad allocation schemes.

2. System and application model

In this paper, our focus is on a multi-processor architecture, as shown in Figure 1. The architecture contains multiple processing elements (PEs) on a chip. Each PE owns a private scratchpad memory. With respect to a specific PE, the SPMs of other PEs are referred to as remote SPMs. A PE has dedicated access to its private SPM with minimum latency. A PE can also access a remote SPM through the crossbar connecting the processors. Access to a remote SPM is relatively slower than accessing private SPM but much faster than accessing the off-chip memory. In this work, we assume that the latency to access remote SPM is bounded by a small constant (since the on-chip links generally operate on high bandwidth, this is a reasonable assumption). This kind of architecture essentially creates a *virtually shared scratchpad memory space* (VS-SPM) among all the PEs [1]. If some item is not available in VS-SPM, a processor can *bypass* the VS-SPM and fetch the memory block from slow external memory. A *bypassing* VS-SPM space creates opportunities to avoid memory spill and reloading delay as compared to its *non-bypassing* counterpart. Consequently, it leads to a fully predictable memory access behavior of the underlying application. All traffic to/from the off-chip memory has to go through a shared TDMA bus which is accessed in a round-robin fashion among all the available PEs. All on-chip SPMs are non-coherent. This helps the architecture to be free of all coherence logic required otherwise. Since the SPMs are non-coherent, there is always *at most one copy* of a particular variable in VS-SPM.

We focus here only on *scalar* and *array* variables in data memory. We assume fully separated buses and memories for both code and data. Therefore, we ignore bus traffic arising from instruction memory accesses.

We model an application as a set of task graphs where each task is mapped to exactly one PE. Each task graph is a directed acyclic graph which contains a number of tasks. Let us assume $\{T_1, \dots, T_N\}$ be the set of N tasks corresponding to all the task graphs. A directed edge between two tasks T_x and T_y in a task graph signifies that task T_y cannot start execution before T_x finishes execution. We assume a *multi-tasking* execution model and we use a *fixed-priority preemptive* scheduling. Our goal is to derive a *compile-time* allocation of *data variables* into VS-SPM and off-chip memory to reduce the application's overall *worst case response time* (WCRT).

3. Related work

The problem of content selection in scratchpad for sequential applications has been well studied. The memory objects considered for allocation into the scratchpad can be *program data* (e.g., [2, 3]) or *program code* (e.g., [4]). Allocating program code requires additional care to maintain program flow, while allocating program data generally calls for specific considerations depending on the type of the data (global, stack, or heap).

Many of the works on scratchpad allocation for sequential applications (such as [5–7]) aim to minimize the average case execution time and energy consumption. Scratchpad allocation for reducing worst case execution time of sequential programs has been addressed in (e.g., [8, 9]) and most recently in [10]. However, concurrent applications require to consider task dependency and multi-processor architectures require the modeling of shared resources.

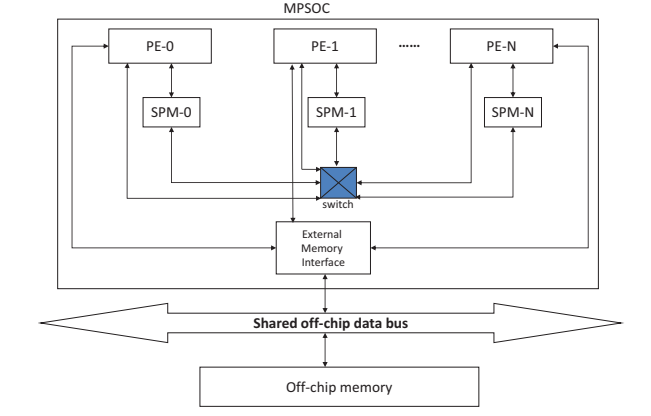


Figure 1. System Architecture

Therefore, none of the above techniques can directly be applied to concurrent applications running on multi-processors.

Scratchpad sharing among multiple processes has been first proposed in [11]. However, it does not focus on a multi-processor architecture. Moreover, the application model in [11] does not consider process interactions, whereas our framework can handle applications specified in the form of a task graph.

Scratchpad sharing among different processing elements in multiprocessor system-on-a-chip has also been explored by researchers. SPM allocation framework for average case performance improvement has been presented, among others, in [1] and [12]. A different approach proposed in [13] uses dynamic configuration of shared SPM space for reducing energy consumption. However, the above mentioned techniques do not consider variable bus delays and they are not useful for improving the worst case performance of an application.

The work proposed in [14] is the closest to ours. It uses a static SPM allocation scheme to reduce the WCRT of concurrent applications. However, there are two key differences between our work and [14]. First, [14] ignores the waiting time to access the shared bus. Secondly, the architecture explored in [14] only has a private SPM for each processing element and the private SPM is shared among different tasks by partitioning or overlay. Current commercial processors such as Cell allow for the SPM space to be (virtually) shared among all the available processing elements. Our work aims to optimize the WCRT of an application by accurate content selection and overlay in this shared SPM space, by accounting for the variable bus delays.

4. Overview of our SPM allocation framework

Figure 3 gives a high level description of our SPM allocation framework. A *bus aware and cycle accurate* WCET analyzer computes WCETs of individual tasks together with the *external memory access profile* of each variable along the worst case execution path (WCEP). WCRT analyzer uses a fixed priority preemptive scheduling and computes the WCRT of overall application from individual WCETs of all tasks. As a by-product, the WCRT analyzer also produces the *lifetime* of each variable (the time interval between which a particular variable might be accessed) and critical path of the application. The SPM allocator computes a set of allocation decisions depending on the memory access profile of each variable and the critical path of the application. An allocation decision could be either to allocate some variable in shared SPM or to revoke a previous allocation decision (i.e., to reclaim the space from a previous allocation decision and deallocate the corresponding variable(s) from SPM). Since a set of allocation decisions might change

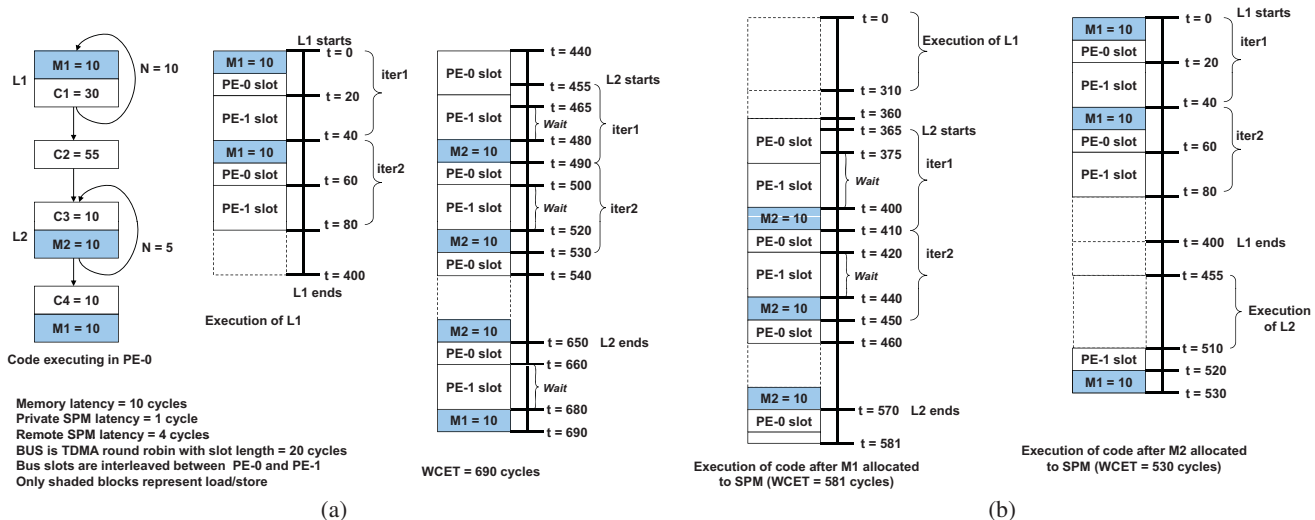


Figure 2. (a) A sample code and its execution without SPM allocation (b) Execution of the code by two possible SPM allocations

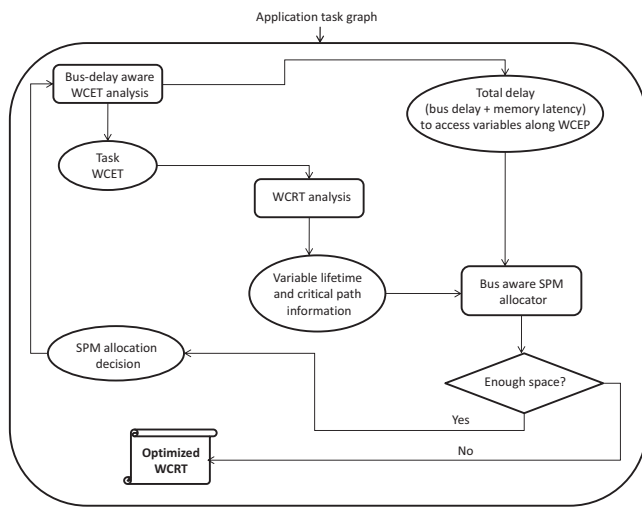


Figure 3. Overview of SPM allocation framework

the memory access statistics and the critical path, the critical path is re-computed to produce a further set of allocation decisions.

It is important to note from Figure 3 that the only information flow from the bus aware WCET analysis to our SPM allocator is in the form of an external memory access profile along WCEP. The nature of shared bus is entirely hidden to the SPM allocator. Therefore, our SPM allocator is independent of the nature of shared bus used by the underlying architecture. A bus-delay aware SPM allocator is the primary focus of this work — we shall give the motivation behind this now and discuss it further in Section 6.

An example Figure 2(a) shows a sample code and its execution at PE-0 in presence of shared bus. “C” blocks in the control flow graph (CFG) represent computations without external memory access. The number inside each block corresponds to the fixed cost of the computation. Only shaded blocks (marked with “M”) in the CFG represent external memory accesses and hence might suffer from variable bus delays. We assume an external memory latency of 10 cycles and TDMA bus slot length is 20 cycles. Let us first examine the execution patterns of two loops (L1 and L2) when there is no scratchpad. Last two parts in Figure 2(a) demonstrate the execution behaviors of L1 and L2. We observe that references to M2

frequently suffer additional bus delays to access the off-chip memory. On the other hand, references to M1 hardly suffer any additional bus delay due to a perfect alignment with corresponding bus slots most of the time. Consequently, final WCET of the example program turns out to be 690 cycles.

We now consider an architecture with scratchpad memory (SPM). Let us assume that private SPM latency is 1 cycle and remote SPM latency is 4 cycles. For simplicity, we assume that we can allocate either M1 or M2 in the SPM but not both due to space constraints. A *bus-unaware* greedy SPM allocation (e.g. in [8]) scheme allocates variables to SPM by traversing them in decreasing order of their access frequencies. Since the access frequency of M1 (11) is higher than that of M2 (5), a greedy bus-unaware SPM allocator will pick M1 as the potential candidate to be allocated in the SPM. The modified execution flow is shown in the first part of Figure 2(b). Even though loop L1 can now be completed in fewer cycles, references to M2 still suffer high bus delays. This leads to an optimized WCET of 581 cycles.

Now assume that we allocate M2 instead of M1 in the SPM (second part of Figure 2(b)). Since M1 accesses are aligned to the beginning of bus slots, they will not encounter any additional bus delay as before. On the other hand, since M2 now has been allocated to SPM, its references no more encounter any bus delay. This leads to a better optimized WCET of 530 cycles.

A *bus-unaware* SPM allocation algorithm does not take into account the bus delay encountered for memory accesses. In Figure 2, accesses of M1 inside loop L1 do not encounter any bus delay. However, each access of M2 suffers additional bus delay. Careful examination through Figure 2(a) reveals that references to M2 contribute more towards the program’s WCET than the references to M1. Therefore, compared to M1, M2 is a better candidate for SPM allocation in this example.

We shall illustrate the work-flow of our iterative SPM allocation framework by using Figure 4. Here we shall take two tasks T1 and T2 executing concurrently at PE-0 and PE-1 respectively (Figure 4(a)). Task T1 is the same program as shown in Figure 2(a). We introduce a new task T2 as shown in Figure 4(a). A careful illustration similar to Figure 2(a) reveals that WCET of T2 is 480 cycles. Both the PEs have private SPMs (SPM-0 and SPM-1). We assume both the tasks start execution at time 0. Our goal is to minimize the overall WCRT of the application containing tasks T1 and T2.

Our technique exploits the lifetime of variable access to efficiently use the shared SPM space. Variable lifetime is indicated by

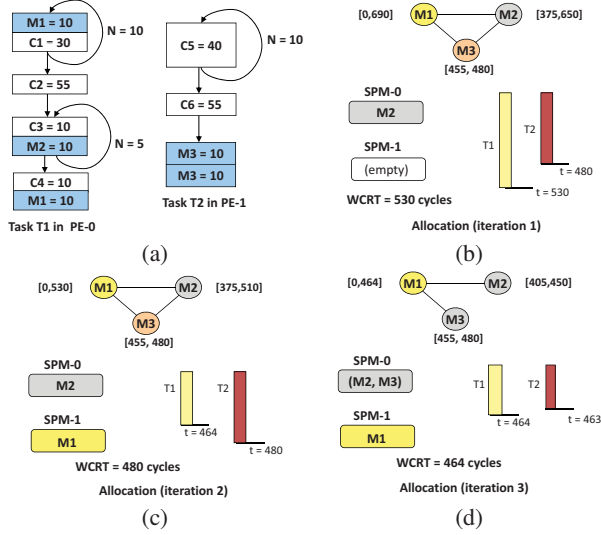


Figure 4. Iterative SPM allocation scheme shown on two tasks T1, T2 running on different processors. Task T1 is same as the example in Fig. 2.

an interval as shown in Figures 4(b)-(d). The interval represents the time span from the *earliest time* the variable could possibly be accessed to the *latest time* it could possibly be accessed. We construct an interference graph from these intervals and produce a coloring of the graph. Each individual color represents a group of variables which are accessed at disjoint time intervals. Interference graph is a globally unified graph which considers all the variables accessed in different tasks running at different PEs. In very first iteration of allocation (Figure 4(b)), we observe that the interference graph is a complete graph. We choose M2 to allocate in SPM-0 as task T1 is in the critical path (having larger WCET than T2) and previously, we observed that M2 suffers more memory latency to access than M1. After allocation of M1 into SPM-0, WCRT reduces (becomes 530 cycles) but the critical path does not change (i.e., task T1) and the interference graph still remains to be a complete graph (Figure 4(c)). Therefore, our choice was to allocate M1 (accessed in task T1) into SPM-1 instead of M3 (accessed in task T2). This leads to a reduced WCRT of 480 cycles and we also observe that the critical path has switched to task T2 (Figure 4(c)). More importantly, the interference graph is no longer a complete graph as M2 and M3 are being accessed at disjoint time interval (Figure 4(d)). Consequently, M2 and M3 can share the same space in SPM-0 and we allocate M3 too in SPM-0 (Figure 4(d)). After this final allocation, WCRT further reduces to 464 cycles (recall that remote SPM latency was assumed to be 4 cycles).

Now assume the presence of a bus-unaware SPM allocator which locally optimizes per-processor execution time (as described in [14]). It would have allocated M1 in SPM-0 (for M1 having higher access frequency than M2) and M3 in SPM-1 (as remote SPM allocation is not considered), resulting in a final WCRT of 581 cycles. This example demonstrates the effectiveness of our approach, as we can considerably improve the WCRT (464 cycles) compared to [14].

5. Bus aware WCRT analysis

For bus-aware SPM allocation, we need to first perform a bus-aware WCET analysis of each individual task. We use our previous work on bus-aware, cycle accurate WCET analysis in [15] for the allocation framework.

The outcome of a single task analysis is a metric C_v attached to each variable v accessed in the task. C_v represents the total contribution of variable v towards WCRT of the task. This contribution includes the total waiting time to access the shared bus as well as the total off-chip memory latency for all references of variable v in the worst-case path. A TDMA bus scheduling policy is used where a bus slot is interleaved among all the PEs in a round-robin fashion. For rest of the discussion, we shall assume that there are a total \mathcal{J} number of processors and the bus slot length assigned to each processor is S_i .

Computation of C_v A bus aware analysis computes the bus delay for each memory reference that may potentially access the shared bus. However, precise computation of this bus delay requires a virtual unrolling of all the loops. Our previous work in [15] uses an approximation to align the start of each loop iteration at the beginning of a new bus schedule. The alignment avoids the virtual unrolling but it requires additional alignment cost for each loop iteration and this cost is included in the WCET computation. Artificial alignment of a loop iteration is not necessary if the loop does not contain any memory reference that may access the shared bus. Let us denote the average alignment cost of a single iteration of loop lp by Δ^{lp} . To use the analysis for bus aware SPM allocation, let us assume that $freq^{mem}$ represents the frequency of an off-chip memory reference mem along the worst case execution path (WCEP) and δ^{mem} is the bus delay computed by the analysis for memory reference mem . Further assume that $MEM(v)$ returns the set of all off-chip memory references of variable v and $LP(mem)$ returns the immediately enclosing loop of memory reference mem . Given the above, we compute C_v as follows:

$$C_v = \sum_{mem \in MEM(v)} (\delta^{mem} + LAT + \Delta^{LP(mem)}) \times freq^{mem} \quad (1)$$

LAT represents the off-chip memory latency. Note that $\Delta^{LP(mem)}$ will disappear after all the variables accessed inside $LP(mem)$ are allocated in SPM. Therefore, variables incurring high memory latency and accessed inside a loop with high alignment penalty are preferred for SPM allocation. Consequently, $\Delta^{LP(mem)}$ is added as a component for computing C_v . It is interesting to notice that the SPM allocator takes only the value of C_v as input. Therefore, a more accurate analysis for computing C_v might easily improve the result generated by our SPM allocator.

Lifetime of a task WCET analysis is carried out initially and after each iteration of SPM allocation algorithm. Let us assume $wcet(t_i, A)$ denotes the WCET of task t_i under allocation A . For lifetime computation, we assign four parameters to each task as follows: $eStart(t_i, A)$ (*earliest start time*), $eFinish(t_i, A)$ (*earliest finish time*), $lStart(t_i, A)$ (*latest start time*) and $lFinish(t_i, A)$ (*latest finish time*). Given an allocation A and the corresponding value of $wcet(t_i, A)$, we can estimate the lifetime of task t_i , defined as the interval between the lower bound on the start time (i.e., $eStart(t_i, A)$) and the upper bound on the finish time (i.e., $lFinish(t_i, A)$) of t_i . This estimation takes into account the dependencies among the tasks (partial ordering imposed by the task graph) as well as preemptions. WCRT of the whole application containing N tasks under allocation A is thus given by the following equation:

$$WCRT_{final} = \max_{1 \leq i \leq N} lFinish(t_i, A) - \min_{1 \leq i \leq N} eStart(t_i, A) \quad (2)$$

WCRT analysis of a single task We consider a fixed-priority preemptive scheduling. Therefore, we need to consider the preemption cost of task t_i . An application is periodic in nature. An application is modeled as a task graph and one activation of the application is the completion of this entire task graph. Therefore, all tasks in the task graph have a common period and deadline which is the period of the entire application. We denote the priority of a task t_i by $pr(t_i)$. Lower numbers are considered to be higher priority. The assigned PE to a task t_i is denoted by $PE(t_i)$. Assume that the set of tasks which may preempt task t_i is denoted by $hp(t_i)$. $hp(t_i)$ is

defined as follows:

$$hp(t_i) = \{t_j \mid t_i \notin \mathcal{D}(t_j) \wedge t_j \notin \mathcal{D}(t_i) \wedge PE(t_j) = PE(t_i) \wedge pr(t_j) < pr(t_i) \wedge [eStart(t_j, A), lFinish(t_j, A)] \cap [eStart(t_i, A), lFinish(t_i, A)] \neq \emptyset\} \quad (3)$$

$\mathcal{D}(t_i)$ denotes the set of tasks which depend (directly or indirectly) on task t_i according to the partial order imposed by the task graph. Therefore, $hp(t_i)$ denotes all higher priority tasks whose lifetimes may overlap with that of t_i in the same PE. WCRT of the task t_i is then computed by the following:

$$wcr(t_i, A) = wcr(t_i, A) + \sum_{t_j \in hp(t_i)} wcr(t_j, A) + |hp(t_i)| \times \mathcal{J} \times S_i \quad (4)$$

Since each task has the same period and deadline, a higher priority task can preempt a lower priority task executing in the same PE at most once. Preemption of a lower priority task will also disturb the external memory access profile of the preempted task beyond the preemption point, which may lead to additional bus delay. Consequently, the delay encountered for a preemption can be at most the worst case execution time of the preempting task together with any additional bus delay encountered for preemption. Note that $|hp(t_i)| \times \mathcal{J} \times S_i$ bounds the additional bus delay. We ignore the operating system overhead due to context switch. Nevertheless, an upper bound on the context switch cost can easily be accounted during the WCRT computation of t_i ($wcr(t_i, A)$).

We have for each task t_i : $lFinish(t_i, A) = lStart(t_i, A) + wcr(t_i, A)$. Further, the partial ordering of tasks in the task graph imposes the constraint that a task t_i can start execution only after all its predecessors have completed execution. In other words, $lStart(t_i, A) \geq lFinish(u, A)$ for all tasks u preceding t_i in the partial order imposed by the application task graph.

For WCRT analysis, we also need to compute the best case execution time (BCET) of each task t_i . BCET of t_i under allocation A is denoted by $bcet(t_i, A)$. We use the following for BCET computation: (i) unless a variable is already allocated to some remote SPM, its location is considered to be the private SPM (i.e., no external memory access is considered when computing $bcet(t_i, A)$). (ii) no preemption cost needs to be considered for BCET (the best-case scenario). Therefore, for each task t_i : $eFinish(t_i, A) = eStart(t_i, A) + bcet(t_i, A)$. Further, due to the partial ordering of tasks in the task graph, $eStart(t_i, A) \geq eFinish(u, A)$ for all tasks u preceding t_i in the partial order imposed by the application task graph.

6. Bus-delay aware Scratchpad allocation

In this section, we describe our iterative SPM allocation algorithm in details. An optimal solution in our setting is clearly *infeasible*. In presence of Q processing elements in the MPSoC, each variable has $Q + 1$ possible places to reside (one in each SPM and the external memory). Consequently, an exhaustive search requires to explore $(Q + 1)^n$ possibilities with n variables, which is clearly infeasible even if n is relatively small. Therefore, in the following discussion, we propose an iterative heuristic which computes a solution very fast and still overpowers previously proposed local scratchpad allocation schemes.

In the following discussions, *private* SPM of a task refers to the private SPM of the PE in which the task is running and with respect to an SPM spm_i , all tasks having private SPM spm_i are considered *local*. Similarly, *remote* SPM of a task refers to any SPM available in the MPSoC other than the *private* SPM of the task.

Computation of variable lifetime An interval $[lo(v), hi(v)]$ represents the lifetime of a variable v . $lo(v)$ indicates the *earliest* pos-

sible time v could possibly be accessed and $hi(v)$ indicates the *latest* possible time for an access to v . These intervals are computed initially and everytime after an SPM allocation decision is finalized. Under allocation A , $lo(v)$ and $hi(v)$ are computed as follows (assuming v is accessed in task t_i):

$$lo(v) = eStart(t_i, A) + \min_{r_i \in fref(v)} bcet(t_i, r_i, A) \quad (5)$$

$$hi(v) = lStart(t_i, A) + \max_{r_i \in lref(v)} wcr(t_i, r_i, A) + \sum_{t_j \in hp(t_i)} wcr(t_j, A) + |hp(t_i)| \times \mathcal{J} \times S_i \quad (6)$$

Recall that $eStart(t_i, A)$ and $lStart(t_i, A)$ are the earliest and latest start times of task t_i under SPM allocation A . $fref(v)$ and $lref(v)$ represent the set of *first* and *last* references (in topological order) to variable v in task t_i respectively. $bcet(t_i, r_i, A)$ and $wcr(t_i, r_i, A)$ represent the *best case* and *worst case* execution time spent from the beginning of task t_i to reference r_i , under allocation A , respectively. To compute the *latest* reference time of variable v , we need to consider the preemption cost. Recall that $hp(t_i)$ represents the set of all tasks which may preempt task t_i and $|hp(t_i)| \times \mathcal{J} \times S_i$ bounds any additional bus delay introduced due to preemption. Therefore, Equation 6 finds the *latest* possible time at which the variable v is accessed.

Interference graph We use the lifetime information of variables to construct an interference graph $G_I = (V_I, E_I)$. Nodes of this graph correspond to different variables. Recall that $\mathcal{D}(t_i)$ denotes the set of tasks which depend (directly or indirectly) on task t_i . There exists an edge between two nodes depicting variables u (accessed in task t_i) and v (accessed in task t_j) if the following condition $pred_{uv}$ holds:

$$pred_{uv} = [lo(u), hi(u)] \cap [lo(v), hi(v)] \neq \emptyset \wedge t_i \notin \mathcal{D}(t_j) \wedge t_j \notin \mathcal{D}(t_i) \wedge u \neq v \quad (7)$$

The condition $pred_{uv}$ represents the scenarios where two different variables u and v might be *live* at the *same* time and thus cannot share the same memory space. As shown in the preceding, two variables from two dependent tasks can never interfere. If u and v are accessed by the same task t_i , number of edges in G_I is reduced by checking whether u and v can be simultaneously live using classical liveness analysis.

SPM allocation using the interference graph Interference graph is used for sharing the available SPM space as much as possible. Each node of the interference graph is assigned a weight. Nodes having higher weight values are given preference for SPM allocations. We want to place data items which incur high memory latency (including bus delay) into the SPM so that external memory access is not needed. At the same time, we want to optimize the critical path of the application and consequently, we want to place data items which are accessed in the critical path, into the SPM. Therefore, we assign a weight ($gain_v$) to each vertex (v) in the interference graph as follows:

$$gain_v = \begin{cases} 0, & \text{if } v \text{ is not accessed in the critical path} \\ & \text{or } v \text{ is allocated in SPM} \\ C_{v,}, & \text{otherwise.} \end{cases} \quad (8)$$

These weights are computed initially and everytime an SPM allocation decision is made. Before going into the formal description

of the technique, we define the following notations that will be used for rest of the discussion:

- $area : V_I \rightarrow \mathbb{N}$, $area(v_i)$ denotes the size of a variable represented by interference graph node v_i .
- $size : 2^{V_I} \rightarrow \mathbb{N}$, $size(S)$ denotes the size of largest variable in set S that resides in external memory. Note that, if S forms an *independent set* in the interference graph, $size(S)$ is the total space needed to allocate the entire set of variables S into the SPM.
- $ref_i \subseteq V_I$: Set of variables accessed in some task assigned to PE i .
- spm_i : Private SPM of PE i .
- SP : Set of all SPMs available in the MPSoC.
- $capacity : SP \rightarrow \mathbb{N}$, $capacity(spm_i)$ denotes the free space in spm_i .
- $location : V_I \rightarrow \{SP \cup \perp\}$, $location(v_i)$ denotes the location of a variable v_i , $location(v_i) = \perp$ if v_i is in external memory.
- $\wp : V_I \rightarrow 2^{V_I}$, $\wp(v_i)$ denotes the set of variables sharing the same SPM space with v_i due to their disjoint lifetimes.

Formal description of the overall technique is given in Algorithm 1. There are mainly two decisions associated with every iteration of Algorithm 1: first, finding a set of variables for allocating in SPM (*maxIndependentSet* function in Algorithm 1) and secondly, finding space in shared SPM to allocate this set of variables (*findSPMspace* function in Algorithm 1). Broadly, our technique exhibits a search algorithm with limited backtracking. A choice made by the algorithm can be either *final* or can be *backtracked* depending on whether the choice improves application performance. The search algorithm terminates when no new choice can be made.

We apply *graph coloring* to the interference graph, the resulting colors will give us groups of variables which are accessed at disjoint time interval. Graph coloring using the minimum number of colors is known to be NP-complete. Therefore, we employ Welsh-Powell algorithm [16], a heuristic method that assigns the first available color to a node without restricting the number of colors to be used. Algorithm 1 follows a reduced backtracking technique. Let us define *weight* of a particular color CL as the sum of weights ($gain_v$) of all vertices colored with CL . Each color in the interference graph represents an independent set and the independent set corresponding to the maximum weighted color contribute a bigger chunk to the application's overall WCRT. Therefore, in each iteration of the algorithm, we choose a color that has the maximum weight. If allocating an independent set \mathcal{IS} into SPM reduces the WCRT of the application, we finalize the allocation of \mathcal{IS} and the location of variables representing set \mathcal{IS} is never changed further. However, if allocating \mathcal{IS} into SPM does not reduce the WCRT, we maintain it in a list *backlog* as long as enough SPM space is available for WCRT improvement. When we run out of space, we search through the *backlog* list to find a victim and reclaim the SPM space assigned to it. The victim is chosen to be the one which occupies maximum amount of space among all other elements in *backlog* list (the term $\max_{(*, occ, *)} backlog$ in Algorithm 1 computes this victim). If

the list *backlog* is empty and there is not enough SPM space to allocate an independent set, the largest variable from the chosen independent set is removed to find a smaller independent set that can be accommodated in free SPM space. Size of *backlog* list represents the maximum depth of backtracking. One could argue about the backtracking depth being nonzero (for zero backtracking depth, an independent set is never allocated to shared SPM unless it reduces the overall WCRT). However, we observe that more than one independent sets (say \mathcal{IS}_1 and \mathcal{IS}_2) are often able to reduce the

Algorithm 1 MIS: SPM allocation by exploiting variable lifetime

```

1: Perform initial WCRT analysis to get the WCRT and critical path;
2: Construct interference graph  $G_I$  and assign weight  $gain_v$  to all its vertices;
3:  $backlog := \phi$ ;
4: repeat
5:   repeat
6:      $\mathcal{IS}_{max} := \text{maxIndependentSet}(G_I)$ ;
7:      $gain := \sum_{v \in \mathcal{IS}_{max}} gain_v$ ;
8:     /*  $gain$  is reset in two conditions: (a) all variables in critical path are already allocated in SPM, (b)  $G_I = \phi$ , and consequently  $\mathcal{IS}_{max} = \phi$ . The allocation is terminated at this point */
9:     if ( $gain = 0$ ) then
10:       Finalize SPM allocation;
11:       return;
12:     end if
13:      $(\mathcal{IS}, occ, R_{spm}) := \text{findSPMspace}(\mathcal{IS}_{max})$ ;
14:     /* If SPM space cannot be found for set  $\mathcal{IS}_{max}$ , some previously allocated space is reclaimed if available. Otherwise, largest variable in  $\mathcal{IS}_{max}$  is removed from  $G_I$  to find a smaller independent set */
15:     if ( $R_{spm} = \phi$ ) then
16:        $(\mathcal{IS}_m, occ_m, spm_i) := \max_{(*, occ, *)} backlog$ ;
17:       if ( $occ_m > 0$ ) then
18:          $capacity(spm_i) := capacity(spm_i) + occ_m$ ;
19:         recompute the critical path and the weights  $gain_v$ ;
20:          $backlog := backlog \setminus (\mathcal{IS}_m, occ_m, spm_i)$ ;
21:       else
22:          $V_{max} := \{v_i \in \mathcal{IS} \mid area(v_i) = size(\mathcal{IS}_{max})\}$ ;
23:          $G_I := G_I \setminus V_{max}$ ;
24:       end if
25:     end if
26:     until ( $R_{spm} \neq \phi$ )
27:      $capacity(R_{spm}) := capacity(R_{spm}) - occ$ ;
28:     recompute the critical path and the weights  $gain_v$ ;
29:     if (WCRT is reduced after allocating  $\mathcal{IS}$  in  $R_{spm}$ ) then
30:        $backlog := \phi$ ;
31:       recompute  $G_I$ ;
32:     else
33:       /* remove the previously selected independent set from the interference graph and continue allocation with the remaining graph */
34:        $backlog := backlog \cup \{(\mathcal{IS}, occ, R_{spm})\}$ ;
35:        $G_I := G_I \setminus \mathcal{IS}$ ;
36:     end if
37:   until ( $G_I = \phi$ )

```

WCRT if allocated together into the SPM, whereas, WCRT might not reduce if either \mathcal{IS}_1 or \mathcal{IS}_2 is allocated to SPM but not both. Therefore, even if the WCRT is not reduced after an allocation decision, we expect that WCRT will reduce in future iterations and we only discard such decision when there is not enough space for a new allocation (recall that allocation decisions that did not lead to WCRT improvement, are maintained in a separate list *backlog*). The above-mentioned situation is encountered very often when the cardinality of an independent set is very small or the expected *gain* from the corresponding allocation decision is low. Consequently, WCRT may improve only by allocating more than one independent sets together. Finally, the interference graph G_I is recomputed only if the WCRT is reduced. This is to ensure that the set of edges in G_I monotonically decreases — a crucial property that maintains the correctness of our algorithm (Theorem 6.2).

We use a heuristic as described in Algorithm 2 to find SPM space for a given independent set \mathcal{IS}_{max} . Note that we only need to find an SPM to allocate the independent set $\mathcal{IS}_{max} \setminus V_{spm}$, where $V_{spm} (\subseteq \mathcal{IS}_{max})$ is a set of variables already allocated in

Algorithm 2 *findSPMspace*: Finding SPM space for a set of variables \mathcal{IS}_{max} having disjoint lifetimes. Total number of processors is \mathcal{J} .

```

1: /* If some variable  $\in \mathcal{IS}_{max}$  is already allocated in SPM, it is checked
   whether the space can further be shared with the current set of variables
    $\mathcal{IS}_{max}$  */
2:  $V_{spm} := \{v_i \in \mathcal{IS}_{max} \mid location(v_i) \in SP\}$ ;
3:  $\mathcal{IS} := \mathcal{IS}_{max} \setminus V_{spm}$ ;
4: /* Required space in SPM for independent set  $\mathcal{IS}_{max}$  */
5:  $occ := size(\mathcal{IS}_{max})$ ;
6: if  $(\exists v_i \in V_{spm}. occ \leq area(v_i) \wedge \bigwedge_{x,y \in \mathcal{IS} \cup \phi(v_i)} \neg pred_{xy})$  then
7:   return  $(\mathcal{IS}, 0, location(v_i))$ ;
8: end if
9: /* Try to minimize the latency incurred by the costliest subgroup in
    $\mathcal{IS}_{max}$  */
10:  $cg(i) := \sum_{v \in \mathcal{IS}_{max} \cap ref_i} gain_v, \forall i \in [1, \mathcal{J}]$ ;
11: if  $(\exists i \in [1, \mathcal{J}]. cg(i) = \max_{k \in [1, \mathcal{J}]} cg(k) \wedge capacity(spm_i) \geq occ)$ 
then
12:   return  $(\mathcal{IS}, occ, spm_i)$ ;
13: end if
14: /* Find a scratchpad having maximum remaining space and has least
   interference from locally executing critical tasks */
15:  $intf := \{u \mid u \in V_I \wedge \exists v \in \mathcal{IS}. pred_{uv}\}$ ;
16:  $hr(spm_i) := \sum_{v \in intf \cap ref_i} \frac{capacity(spm_i) - occ}{gain_v}, \forall i \in [1, \mathcal{J}]$ ;
17: if  $(\exists i \in [1, \mathcal{J}]. hr(spm_i) \geq 0 \wedge hr(spm_i) = \max_{k \in [1, \mathcal{J}]} hr(spm_k))$ 
then
18:   return  $(\mathcal{IS}, occ, spm_i)$ ;
19: end if
20: return  $(\mathcal{IS}, 0, \phi)$ ;

```

the SPM space. Choosing an SPM for allocating a group of non-interfering variables has a space vs quality trade-off. Since, a group of variables are sharing the space, it will create opportunities for more variables to be accommodated in SPM. On the other hand, as the interference graph is a globally unified graph, a group may consist of variables that are accessed in different processors. Therefore, if the group is allocated the same space, some variables in the group might be accessed *remotely* and thereby limit the WCRT improvement. Since a very limited amount of SPM is normally available in a processor, our primary focus is to utilize the available space with maximum possible sharing. Therefore, in the first step of our heuristic, we check whether the set of variables \mathcal{IS}_{max} can share SPM space with some variables already allocated in SPM. However, if our first step is unsuccessful, we try to improve the WCRT by minimizing the latency incurred by the *costliest subgroup* in \mathcal{IS}_{max} . A costliest subgroup is a set of variables in \mathcal{IS}_{max} that are accessed in the same processor and have maximum *cumulative weight* (sum of the elements' weight $gain_v$). Consequently, if sufficient space is available, we allocate $\mathcal{IS}_{max} \setminus V_{spm}$ in the private SPM of the processor accessing this costliest subgroup. In our final step, we choose an SPM that has the maximum remaining space and has minimum interference with $\mathcal{IS}_{max} \setminus V_{spm}$ from locally executing critical tasks. We try to minimize the possibility of high interference in the private SPMs of critical tasks at this final stage.

Following three theorems highlight certain crucial properties of our allocation technique. We only provide the proof sketches here, detailed proofs have been removed due to space constraints.

Theorem 6.1. *Set of edges in the interference graph monotonically decreases over different iterations of Algorithm 1.*

Proof. We prove this by contradiction. In Algorithm 1, we recompute G_I if and only if WCRT is reduced. Let us assume a specific recomputation of G_I as $G_I^{m+1} = (V_I^{m+1}, E_I^{m+1})$ and assume that the set of variables allocated to SPM is A_{m+1} . Further assume, the immediate last recomputation of G_I was $G_I^m = (V_I^m, E_I^m)$ and had a set of SPM-allocated variables A_m . Clearly, $A_m \subseteq A_{m+1}$ and $V_I \setminus A_{m+1} \subseteq V_I \setminus A_m$ where V_I is the set of all variables. More over, locations of the set of variables A_m are never changed after computing G_I^m . By contradiction, assume $E_I^m \subset E_I^{m+1}$. When computing WCRT with allocation A_m (A_{m+1}), location of the set of variables $V_I \setminus A_m$ ($V_I \setminus A_{m+1}$) is taken as off-chip memory to exploit the worst-case scenario. On the other hand, BCET computation under allocation A_m (A_{m+1}) takes the location of the set of variables $V_I \setminus A_m$ ($V_I \setminus A_{m+1}$) as private SPM to exploit the best-case situation. Close inspection of Equation 7 reveals that the property $E_I^m \subset E_I^{m+1}$ can only be satisfied in following two conditions: first, WCRT of some task (or task fragment) is comparatively higher with allocation A_{m+1} than with allocation A_m . It is not possible as $A_m \subseteq A_{m+1}$ and on-chip SPMs have lower latencies than off-chip memory. Secondly, BCET of some task (or task fragment) is more with allocation A_m than with allocation A_{m+1} . By a similar reasoning we argue that it is also not possible as $V_I \setminus A_{m+1} \subseteq V_I \setminus A_m$ and private SPM has the lowest latency. \square

Theorem 6.2. *Set of variables sharing the same space in SPM can never have interfering lifetimes across different iterations of SPM allocation in Algorithm 1.*

Proof. Two variables v_i and v_j could be allocated at the same space in shared SPM only if the edge $(v_i, v_j) \notin E_I$. However, according to Theorem 6.1, set of edges in G_I monotonically decreases. Therefore, the property $(v_i, v_j) \notin E_I$ must be satisfied in all future iterations of Algorithm 1 (i.e., after the iteration where v_i and v_j had been allotted the same SPM space). Consequently, set of variables occupying the same space can never have interfering lifetimes. \square

Time complexity We propose the following theorem to analyze the complexity of our iterative allocation framework:

Theorem 6.3. *Let us assume $|V_I|$ is the total number of variables in the interference graph. Total number of iterations in our framework (bound of the outer loop in Algorithm 1) cannot exceed $\frac{|V_I|(|V_I|+1)}{2}$.*

Proof. Let us assume that after a specific recomputation of G_I , X is the number of variables residing in off-chip memory and $T(X)$ is the number of remaining iterations in Algorithm 1. In the worst case scenario, $T(X)$ follows the recurrence $T(X) = T(X-1) + X$. In the worst case, interference graph could be a complete graph in every iteration, making the size of selected independent set by *maxIndependentSet* exactly 1. Consequently, at most X iterations might be required to finalize an SPM allocation decision (because all previous $X-1$ choices may not lead to WCRT reduction and subsequently put into the *backlog* list). Assume that the variable v_X is chosen for SPM allocation at X -th iteration. Note that, if WCRT is not improved after allocating v_X , Algorithm 1 will be terminated. Similarly, if WCRT improves after successfully allocating all X variables in SPM, Algorithm 1 also terminates as there are nothing more to allocate in SPM. Therefore, to visualize the worst case situation, we assume that *backlog* list is emptied out at X -th iteration to accommodate v_X in SPM and allocation of v_X improves the WCRT. Since allocation of v_X leads to $X-1$ variables in off-chip memory, it will require $T(X-1)$ iterations more for Algorithm 1 to terminate. Solving the recurrence we get

$T(X) = \frac{X(X+1)}{2}$. Since there are a total of $|V_I|$ nodes in the interference graph, maximum number of iterations in Algorithm 1 is bounded by $\frac{|V_I|(|V_I|+1)}{2}$. \square

In practice, though, above theoretical bound is not reached. It is mostly because of the two reasons: first, the interference graph is hardly a complete graph in *any* iteration and secondly, search depth to finalize an allocation decision is much lower than the number of variables residing in off-chip memory. We shall see in the experimental section that our framework converges quickly.

7. Experimental evaluation

Benchmarks We have used two real-life embedded applications to evaluate our scratchpad allocation schemes. Our first case study corresponds to a large fragment of DEBIE-1 DPU Software [17], an in-situ space debris monitoring instrument developed by Space Systems Finland Ltd. We model this fragment as a task graph, shown in Figure 5. The number beside each task in Figure 5 shows the assignment of tasks to different PEs. Code size of the tasks varies from 448 bytes to 23288 bytes (average code size 8825 bytes) whereas the data size varies from 18 bytes to 66972 bytes (average data size 55448 bytes).

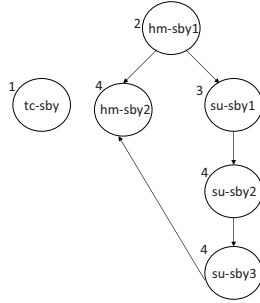


Figure 5. Task graph extracted from DEBIE-DPU

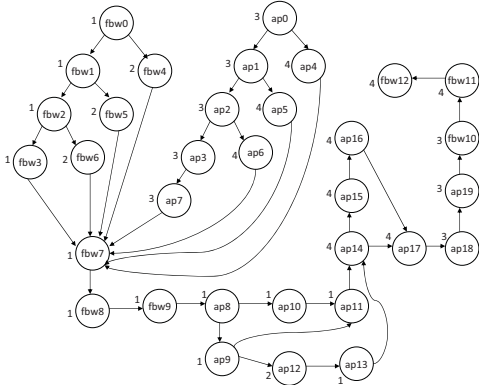


Figure 6. Task graph of *papabench*

Our second case study is the Unmanned Aerial Vehicle (UAV) control application from *papabench* [18], a derivation from the real-time embedded UAV control software Paparazzi. The controller consists of two main functional units, *fly_by_wire* and *autopilot*, which are inter-connected by SPI serial link. *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. One scenario in the manual mode is modeled as a task graph and is shown in Figure 6. The number beside each

task in Figure 6 shows the assignment of tasks to different PEs. Code size of the tasks varies from 96 bytes to 6468 bytes (average code size 1903 bytes) whereas the data size varies from 130 bytes to 1878 bytes (average data size 1105 bytes).

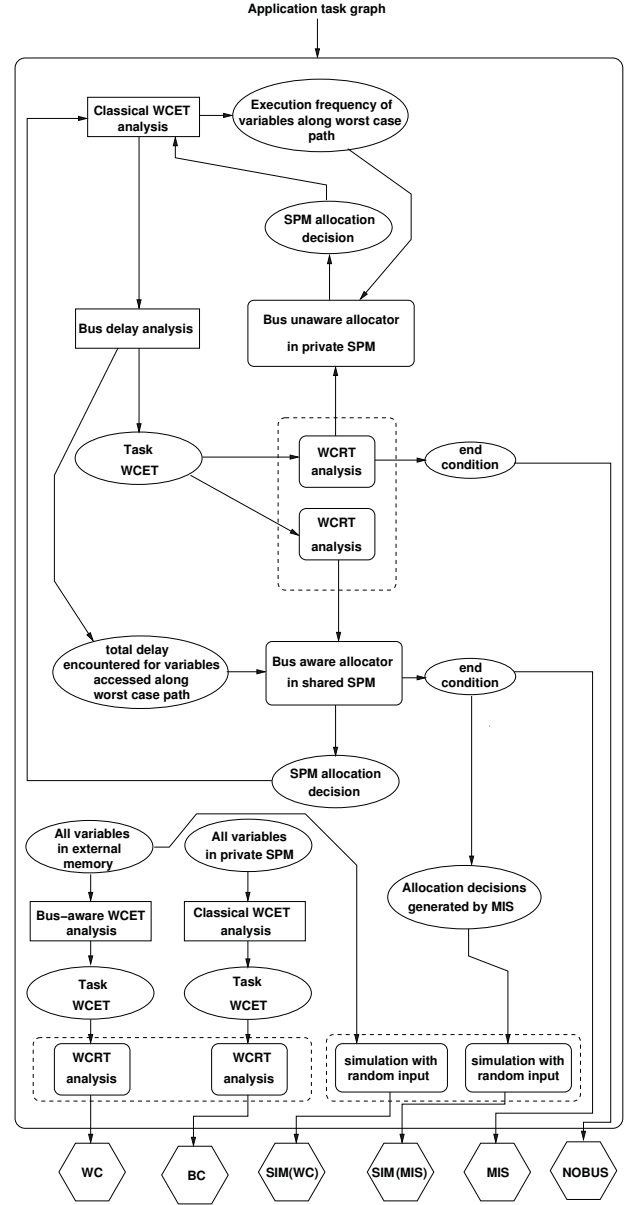


Figure 7. Experimental setup

Experimental setup We have implemented our allocation algorithm inside a cycle accurate WCRT analyzer. Our full experimental setup is shown in Figure 7. We shall use the terminologies shown in Figure 7 for rest of the discussion in this section. Let us assume *WC* represents the scenario where all variables are accessed from external memory. Similarly, *BC* represents the scenario where all variables are accessed from private SPM. Therefore, *WC* and *BC* provide upper and lower bound of optimized WCRT value respectively. To check the improvement by our SPM allocator (result shown by “*MIS*” in Figure 7), we measure the ratio $\frac{WC}{MIS} - 1$. Clearly, *BC* is a measurement of best possible scenario when all variables are accessed from private SPM and $\frac{WC}{BC} - 1$ bounds

Benchmark	Size of interference graph		Analysis statistics		
	Nodes	Edges	Iterations	Time	WC (cycles)
<i>debie</i>	283	27688	99	88 secs	3773×10^6
<i>papabench</i>	506	16872	210	119 secs	515×10^3

Table 1. Problem size, analysis time and WCRT

the best possible improvement. We compare our improved WCRT with a *bus-unaware* allocator (result shown by “*NOBUS*” in Figure 7) that optimizes the content selection in individual private SPMs (similar to the SPM allocator described in [14]). Improvement from *NOBUS* is similarly measured as $\frac{WC}{NOBUS} - 1$. We also check the effect of our Algorithm on average case response time (ACRT) by running the application using random inputs with and without our final SPM allocation decision (results shown by “*SIM(MIS)*” and “*SIM(WC)*” in Figure 7 respectively). Both “*SIM(WC)*” and “*SIM(MIS)*” are obtained using a cycle accurate simulator. ACRT improvement using our technique is measured as $\frac{SIM(WC)}{SIM(MIS)} - 1$.

We assume a single in-order pipeline for each processor. Each processor can access its private SPM in a single cycle. Our experimental results (*i.e.* WCRT of the application) mainly depend on four different micro-architectural parameters whose default values are configured as follows: i) total size of shared scratchpad space (relative to total data size in the application): 10%, ii) remote SPM latency: 4 cycles, iii) off-chip memory latency: 30 cycles, iv) round-robin TDMA bus slot length: 50 cycles and v) number of PEs: 4. We have carried out experiments with two processors and with four processors. For all experiments with two PEs, we combine the tasks running in PE 2 and PE 3 to run in one PE and combine rest of the tasks to run in another PE. We perform all our experiments in a 3 GHz Pentium IV machine having 1 GB of RAM and running Ubuntu 8.10 as the operating system. Table 1 gives an idea about the problem size and time taken by our iterative SPM allocator in default configuration. The time shown in Table 1 features the total time, including the time taken by repeated computations of bus-aware WCET and SPM allocation decisions by Algorithm 1. In general, none of our reported experiments takes more than 2 minutes to complete.

Sensitivity of WCRT reduction with respect to SPM size Figures 8(a)-8(b) demonstrate WCRT improvement for different SPM size. SPM size is chosen in a way such that sufficient amount of interferences take place among all data items (to accommodate them in shared SPM space). Above figures clearly demonstrate that we can obtain significant WCRT reduction by using our SPM allocator. For *debie*, an upper bound on WCRT improvement or the measured ratio $\frac{WC}{BC} - 1 \times 100\%$ is 1500% (700%) using 4 PEs (2 PEs). Similarly, for *papabench*, the upper bound on WCRT improvement is 710% (410%) using 4 PEs (2 PEs). WCRT is consistently improved with bigger SPM size, which is expected as the interferences among data items reduce with bigger SPM size. Interferences among data items also reduce when more processors are used. Since *debie* has much smaller number of tasks compared to *papabench*, the reduction in interferences for *papabench* is much higher compared to *debie* when more processors are used. We observe the situation in our result – for *debie*, WCRT improvement hardly gets affected with more processors, whereas for *papabench*, WCRT is improved upto 100% when 4 processors are used instead of 2. Finally, our SPM allocator can improve the WCRT considerably compared to a bus-unaware allocator — with a maximum improvement being more than 60%.

Sensitivity of WCRT reduction with respect to bus slot length We have also measured the sensitivity of our allocator with bus slot length. This measurement is shown in Figures 8(c)-8(d). Average improvement from our SPM allocator is 52% (46%) for *debie*

(*papabench*) when compared with a bus-unaware SPM allocator over a bus slot length range of 40-80 cycles.

Summary of other results To test the robustness of our approach, we have measured its sensitivity with different remote SPM latencies. Figure 8(e) demonstrates this result both for *papabench* and *debie*. In contrast to the bus unaware allocator, WCRT improvement from our SPM allocator decreases with increased remote SPM latency, as fetching memory blocks from remote SPM now takes more time. Nevertheless, the rate of decrement is quite low (maximum 5%) and the average improvement over bus unaware allocator remains at 55% over a range of 4-12 cycles remote SPM latency. We have also checked the WCRT improvement by varying off-chip memory latency. When checked with different off-chip memory latencies over a range of 10-50 cycles, the percentage reduction in WCRT remains similar (to Fig. 8). Finally, we have also measured the effect of our WCRT oriented optimization on average case response time (ACRT). Unfortunately, the inputs to *debie* are not available in public domain, which prevents us from running simulation and producing ACRT in *debie*. Therefore, we present the result of ACRT improvement for *papabench* (in Figure 8(f)). As our SPM allocator aims to optimize WCRT and the critical path, we observe that reduction in ACRT is not much compared to the same in WCRT. As evidenced by Figure 8(f), ACRT is reduced by 130% on average over a varying range of scratchpad size.

8. Extensions and Future Work

Applications using shared variables Our current implementation does not handle shared variables among different tasks. More precisely, our allocation framework only considers the set of variables which are accessed by *exactly one* task. However, our allocation method can be modified to deal with shared variables as follows: first, there will be *at most one copy* of each shared variable in the SPM space to maintain *coherency*. Secondly, a shared variable may be accessed by *critical* as well as *non-critical* tasks. Therefore, when we compute the metric $gain_v$ (refer to Equation 8) for a shared variable v , $gain_v$ is set to be the sum of C_v values only in the critical tasks (*i.e.*, references to shared variable v in all non-critical tasks are ignored). Thirdly, lifetime of a shared variable must take into account all the tasks (in application) in which the shared variable might possibly be accessed. In future, we plan to extend our work to include shared variables.

Other multi-processor architectures Our underlying architecture contains a crossbar to access fast on-chip memories. However, some of the architectures [19] use a fast on-chip bus for accessing a remote SPM. Since the on-chip buses operate on high bandwidth, remote SPM latency is still bounded by a small constant. Consequently, our SPM allocation framework can be applied without modification.

ACRT optimization Interference graph in our SPM allocation framework is used for finding a group of variables having disjoint lifetimes. Therefore, interference graph can also be used for other kind of optimization which allows SPM space sharing among different variables. Only driving factor for WCRT oriented optimization is the assigned $gain_v$ metric for each variable v . For ACRT optimization, a *trace* can be collected using a simulator, which will include the external memory access profile along the most frequently accessed path π . $gain_v$ will represent the total latency incurred (including the bus delay) to access variable v along π . Only non-trivial task is to efficiently recompute $gain_v$ after an allocation decision. In future, we plan to check the efficacy and scalability of our allocation framework for ACRT guided optimization.

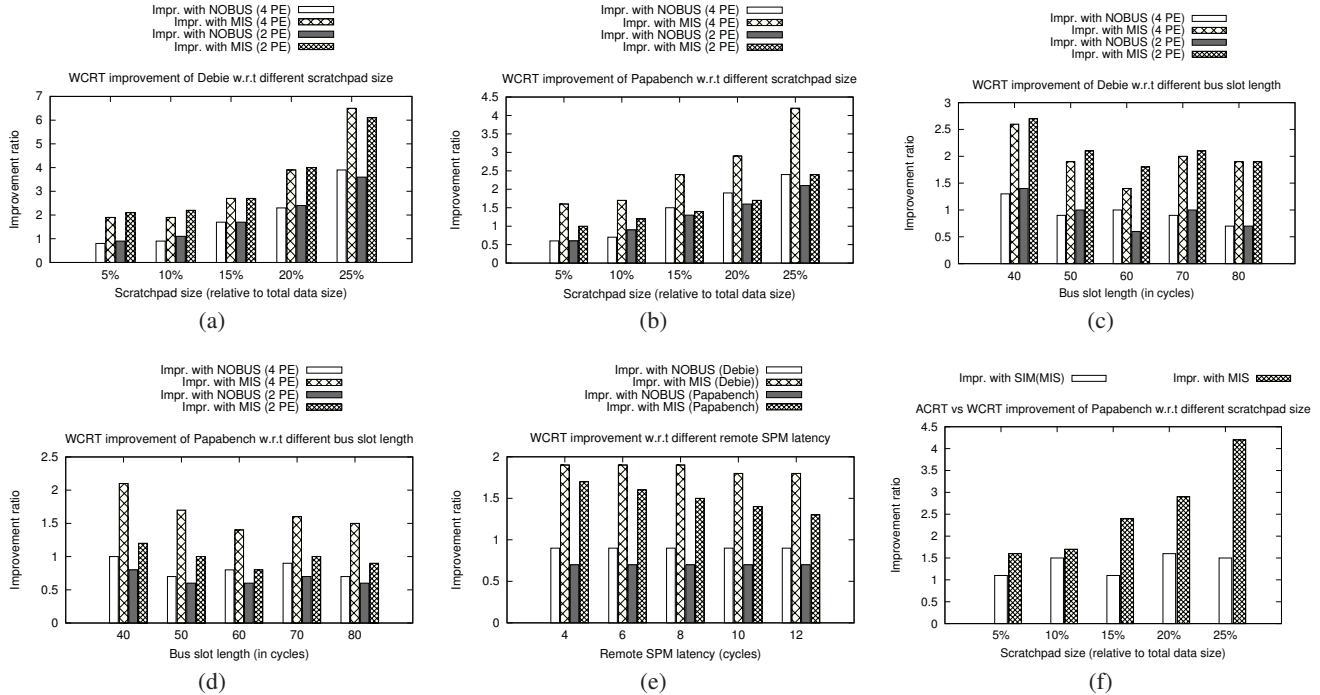


Figure 8. Experimental evaluation of our allocation framework

9. Conclusion

In this work, we have presented our scratchpad allocation framework for multi-processor system-on-chip (MPSoC) platforms. The prime novelty in our work is to incorporate the bus schedule into the multi-processor scratchpad allocation scheme. Our allocation framework exploits the shared scratchpad space available in MPSoCs, and considers variable lifetimes to efficiently utilize the available shared scratchpad space. As evidenced by our experiments, our scratchpad allocation scheme is able to significantly reduce the WCRT of real-life embedded applications. Our results are considerably better when compared with an existing SPM allocation framework. Our allocation method is efficient and thus the scalability of our framework is evident.

10. Acknowledgement

This work was partially supported by NUS grants R252-000-321-112 and R-252-000-416-112.

References

- [1] M.T. Kandemir, J. Ramanujam, and A.N. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *DAC*, 2002.
- [2] J.-F. Deverge and I. Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *ECRTS*, 2007.
- [3] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3), 2000.
- [4] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *DATE*, 2004.
- [5] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE*, 2002.
- [6] S. Udayakumar and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES*, 2003.
- [7] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *CODES+ISSS*, 2004.
- [8] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *RTSS*, 2005.
- [9] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *ECRTS*, 2006.
- [10] H. Falk and J.C. Kleinsorge. Optimal static WCET-aware scratchpad allocation of program code. In *DAC*, 2009.
- [11] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *ESTImedia*, 2005.
- [12] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for MPSoC architectures. In *CASES*, 2006.
- [13] M. Kandemir, O. Ozturk, and M. Karakoy. Dynamic on-chip memory management for chip multiprocessors. In *CASES*, 2004.
- [14] V. Suhendra, A. Roychoudhury, and T. Mitra. Scratchpad allocation for concurrent embedded software. *ACM Trans. Program. Lang. Syst.*, 32(4), 2010.
- [15] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *SCOPES*, 2010.
- [16] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1), 1967.
- [17] European Space Agency. DEBIE – First standard space debris monitoring instrument, 2008. Available at: <http://gate.etamax.de/edid/publicaccess/debie1.php>.
- [18] F. Nemer, H. Cassé, P. Sainrat, J.P. Bahsoun, and M. De Michiel. Papabench: a free real-time benchmark. In *WCET Workshop*, 2006.
- [19] M. Gschwind. The Cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35(3), 2007.