Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures

Vivy Suhendra, Chandrashekar Raghavan, Tulika Mitra
School of Computing
National University of Singapore
{vivy, chandra1, tulika}@comp.nus.edu.sg

ABSTRACT

Multiprocessor system-on-chip (MPSoC) is an integrated circuit containing multiple instruction-set processors on a single chip that implements most of the functionality of a complex electronic system. An MPSoC architecture is, in general, customized for an embedded application. A critical component of this customization process is the on-chip memory system configuration. Embedded systems increasingly employ software-controlled scratchpad memory (SPM) due to its inherent advantages in terms of area, energy, and timing predictability compared to caches. An application-specific flexible partitioning of the on-chip SPM budget among the processors is critical for performance optimization. Moreover, scheduling the tasks of an application on to the processors and partitioning the SPM are inter-dependent even though these steps are decoupled in the traditional design space exploration process. In this work, we design an integrated task mapping, scheduling, SPM partitioning, and data allocation technique based on Integer Linear Programming (ILP) formulation. Our ILP formulation explores the optimal performance limit and shows that integrated task scheduling and SPM optimization improves performance by up to 80% for embedded applications.

Categories and Subject Descriptors

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

General Terms

Design, Performance

Keywords

MPSoC, scratchpad memory, task mapping, scheduling

1. INTRODUCTION

Increasing concerns about the energy and thermal behavior of embedded systems is leading to designs with multiple homogeneous/heterogeneous cores or processors on a single chip. Significant research efforts have been invested in partitioning, mapping, and scheduling the tasks corresponding to an embedded application on to multiple processors. However, the increasing performance gap between on-chip and off-chip memory implies that the design of on-chip memory hierarchy has the maximum impact on the performance of an application. In this paper, we focus on customization of on-chip scratchpad memory (SPM) for multiprocessor system-on-chip (MPSoC) architectures.

SPM refers to the memory residing on-chip that is mapped into the memory address space disjoint from the off-chip memory. SPM is usually implemented using SRAM technology and allows fast access to the data, whereas an access to off-chip memory incurs longer latency. The main difference between the conventional cache and SPM is that the SPM guarantees single-cycle access latency whereas an access to the cache may result in a miss thereby incurring longer latency due to off-chip access. As the memory access latencies are predictable, scratchpad memories have become popular for real-time embedded systems. The other advantages of scratchpad memory include reduced area and energy consumption compared to caches [4]. However, now the burden of allocating data to scratchpad memory lies with the compiler.

In this paper, we assume an MPSoC architecture where each processor has its private SPM. Moreover, a processor can also access another processor's private SPM albeit with an increased latency. Given an application and a budget for total on-chip SPM, our goal is to come up with the appropriate configuration and data mapping for the private SPMs of all processors so as to maximize the performance of the application. The appropriate configuration of a processor's private SPM critically depends on the tasks mapped to that processor. Therefore, task mapping/scheduling and SPM configuration are dependent on each other. Traditional design space exploration frameworks implement these two phases separately, leading to sub-optimal performance. In this paper, we propose an integer linear programming (ILP) based technique for integrated task mapping/scheduling, SPM partitioning, and data mapping. Our results indicate that flexible partitioning of the SPM budget among the processors compared to equal partitioning results in up to 60% performance improvement. Moreover, integrated memory optimization and task scheduling can improve performance by up to 80% compared to implementing these two phases separately.

The main contributions of our work compared to previous research are (1) we investigate the interaction between task scheduling, SPM partitioning and data allocation; (2) we propose an integrated task scheduling, SPM partitioning and data allocation solution based on integer linear programming (ILP) formulation.

CASES'06, October 23–25, 2006, Seoul, Korea.

2. PROBLEM FORMULATION

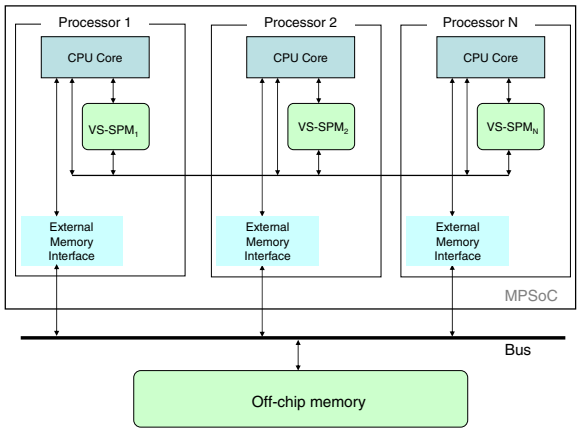


Figure 1: Embedded single-chip multiprocessor with virtually shared scratchpad memory.

Architectural Model. In this paper, we focus on embedded single-chip multiprocessor architecture as shown in Figure 1. The architecture contains multiple processor cores on chip. The cores can be homogeneous or heterogeneous. The processor cores communicate with the shared off-chip memory via a bus. The architecture uses scratchpad memory (SPM), which is fast SRAM managed by software (application and/or compiler). In the single-chip multiprocessor setting, each processor core can access its *private SPM* as well as SPMs of other processors (*remote SPMs*). Such a setup, as described in [10], is called *virtually shared scratchpad memory* or VS-SPM. A processor core has dedicated access to its private SPM with minimum latency — usually a single cycle. Access to a remote SPM also incurs low latency (e.g., 4–16 cycles) due to the fast on-chip communication link among the processor cores. However, off-chip memory access has very high latency (e.g., 100 cycles) due to the performance gap between processor and DRAM technology. Access conflicts between multiple processors may also arise in the bus, adding non-trivial variable delays to the off-chip memory accesses. For simplicity, in this work we assume that the latency incurred by every off-chip memory access is a constant. To avoid coherency issues, we also make the assumption that a memory location can be mapped to at most one SPM. The Cell processor [8] is an example of a real system with similar architecture even though its recommended programming model is somewhat different.

In this work, we focus on SPM for data, but our strategy applies to SPM for instructions as well. That is, our formulation can be easily configured for allocating general memory objects in the form of data variables or blocks of program code.

Task Graph. We assume that the embedded application is specified as a task graph. The task graph is a directed acyclic graph that represents the key computation blocks (tasks) of an application as nodes and the communication between these tasks as edges. A task can be mapped to any of the processing cores. Therefore, associated with each task T are the execution times corresponding to running the task T on each of the processing cores. In case of homogeneous cores, there is only one execution time associated with each task. Note that for our problem, the execution time of a task T on a processor P depends on the placement of its data variables in the SPM. Therefore, we estimate the execution time assuming that all the data variables are accessed from the off-chip memory. An edge from task T to T' in the task graph represents data transfer between these tasks. Therefore, associated with each edge is the amount of data transferred along that edge.

As memory hierarchy design is the main focus of this paper, each task is also associated with the sizes and access frequencies of data variables obtained through profiling. Note that the data access pattern of a task can be different depending on which processor it gets mapped to (if the processors are heterogeneous). In that case, for each task we maintain multiple access patterns corresponding to the different processor cores.

Pipelined Scheduling. Most streaming applications, such as multimedia and digital signal processing (DSP) applications, are iterative in nature. For these applications, the execution of the task graph is evoked repeatedly for a stream of input data. Hence, these applications are amenable to pipelined implementation for greater throughput [6]. The pipelined implementation benefits from allowing multiple processors execute multiple iterations of the task graph at the same time. Even though pipelined scheduling is more challenging, we consider it in our problem formulation. Note that the objective for sequential implementation is to minimize the execution time of a single iteration of the task graph. In contrast, the objective of pipelined implementation is to minimize the initiation interval (II), which is the time difference between the start of two consecutive iterations of the task graph. Minimizing the initiation interval results in optimal throughput for a streaming application.

Problem Statement. Given a task graph, the architectural model and a bound on the total available on-chip SRAM, our goal is to find the optimal SPM configuration that results in minimum initiation interval (II). This problem can be decomposed into three sub-problems.

- *Mapping/Scheduling* of the tasks to the processors as well as communication among the tasks.
- *SPM Partitioning:* Finding the optimal size for each private SPM.
- *Data allocation:* Allocating data variables corresponding to the tasks to the SPMs.

We present a flexible approach that explores the solution space of possible task mapping/scheduling, SPM configuration and data allocations together.

3. MOTIVATING EXAMPLE

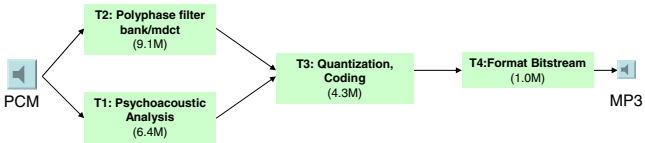


Figure 2: Task graph of LAME MP3 encoder. The numbers in brackets indicate the execution time of the tasks (in cycles). We assume homogeneous processors for this example.

To illustrate the interaction among task scheduling, SPM partitioning and data allocation, we will use the task graph shown in Figure 2. The task graph corresponds to LAME MP3 encoder from

the MiBench benchmark suite. It consists of four tasks and encodes a sequence of MP3 audio frames. Due to the task level parallelism, this application can take advantage of task pipelining as well as multiprocessing. The execution time of the tasks (assuming all the variables are located in off-chip memory) are obtained through profiling. Our MPSoC architecture has two homogeneous on-chip processors and a total on-chip SPM budget of 4KB.

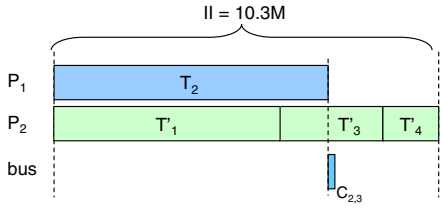


Figure 3: Optimal pipelined schedule for the task graph in Figure 2 without considering data allocation. Non-primed labels indicate tasks/communications from the current instance, while primed labels indicate tasks/communications from the previous instance.

Existing design-space exploration strategies will first map and schedule the tasks and communication onto the two processors without considering the allocation of variables to the SPMs. Figure 3 shows the optimal pipelined schedule for the task graph in Figure 2. This schedule can process one audio frame every 10.3M cycles, which is the initiation interval II for the schedule.

Now, we consider allocating the data variables to on-chip SPMs. The common practice is to partition the total SPM budget equally between the two processors, i.e., each processor has 2KB private SPM. The variables of task T_2 are allocated to the 2KB SPM of processor $P1$. Similarly, the variables of tasks T_1, T_3, and T_4 are allocated to the 2KB SPM of processor P_2. We call this the **EQ strategy**, which stands for equal partitioning of SPM. The allocation reduces the total execution time of T_1, T_3, T_4 to 8.2M cycles. Therefore, the II reduces to 8.2M cycles. However, we notice that the combined execution time of T_1, T_3, T_4 on P_2 determines the II in both cases (with or without allocation). That is, the reduction of task T_2's execution time does not have any effect on the global throughput of the application. This example indicates that allocating a bigger share of SPM to processor P_2 would have been a better strategy.

So we now explore an integrated SPM partitioning and data allocation strategy. We call this **PF strategy**, which stands for partially flexible strategy. Note that in this case the task mapping and scheduling have also been performed beforehand. As expected, this strategy allocates a larger SPM space to processor P_2 and reduces the II to 7.6M cycles. The 1152 bytes allocated to P_1 is used to keep its execution time below this II value. Given the schedule shown in Figure 3, this is the optimal II achievable with 4KB on-chip SPM.

However, fixing the task schedule a-priori without considering the effect of data allocation on the execution time may miss the global optima. For example, task T_2 has the longest execution time without data allocation and hence it is mapped onto a separate processor. However, its execution time may reduce considerably after data allocation and hence it may not be optimal to allocate this task on a separate processor. Exploring the design space of task scheduling, SPM partitioning and data allocation together could potentially reach a design point that is not possible through decoupled scheduling and SPM allocation.

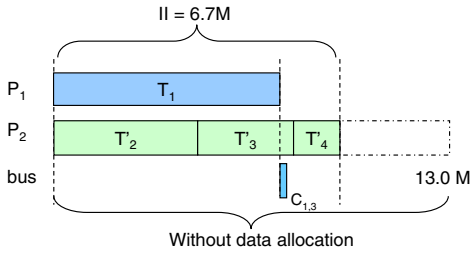


Figure 4: Optimal pipelined schedule for the task graph in Figure 2 through integrated task scheduling, SPM partitioning and data allocation. Non-primed labels indicate tasks/communications from the current instance, while primed labels indicate tasks/communications from the previous instance.

Therefore, we devise a flexible approach, which essentially combines the task scheduling, SPM partitioning, and data allocation phases. We will call this completely flexible strategy (**CF strategy**). Figure 4 shows the schedule produced by CF strategy for the same task graph. The schedule is different from the schedule shown in Figure 3, which does not consider data allocation. In particular, task T_1 has been mapped to a separate processor instead of task T_2. A decoupled scheduling phase can never produce the schedule in Figure 4 as its performance without data allocation is extremely poor (II = 13M cycles). However, with data allocation, this schedule produces an optimal II=6.7M cycles. Incidentally, the entire SPM space is allocated to processor P_2.

4. RELATED WORK

In this section, we briefly review previous research in the area of scheduling task graphs on multiprocessors and data allocation for scratchpad memory (SPM).

The problem of scheduling a task graph on multiple homogenous processors in order to minimize execution time (or energy) has been studied extensively. In its general form, this problem is NP-complete and hence a number of heuristics have been proposed; see [12] for a comprehensive comparison of these heuristics. These works mostly consider non-pipelined scheduling. Benini et al. [5] propose a hybrid constraint programming and integer programming based approach for finding the optimal pipelined schedule. The related problem of mapping and scheduling tasks to a set of heterogenous processing elements has been studied in the context of hardware/software co-design [14]. Technically, this partitioning and scheduling problem for co-design is quite similar to our multiprocessor scheduling problem. Niemann and Marwedel [15] propose an ILP-based solution for the co-design problem. Recently, various research groups have proposed pipelined scheduling solutions for this problem as well, especially in the context of streaming applications. Chatha and Vemuri [6] propose a branch-and-bound solution whereas Kuang et al. [11] propose an ILP-based solution. However, the objective of both these works is to minimize the total component cost and hence the number of pipeline stages.

Scratchpad memory allocation techniques in uniprocessor setting have been studied extensively in the past. Panda et al. have developed a comprehensive allocation strategy for scratchpad memory [19, 18] to improve the program performance. Avissar et al. [3] propose a 0-1 ILP solution to optimally allocate global and stack variables. Their more recent work extends this approach to heap memory [7]. Sjodin et al. [20] also propose a 0-1 ILP solution for

scratchpad allocation. Their goal is to reduce the code size through allocation. Similarly, Steinke et al. [21] formulate an ILP-based allocation strategy to reduce the overall energy consumption. Angiolini et al. [1] propose an alternative approach where the SPM is optimally partitioned into multiple banks in order to improve energy efficiency through a dynamic programming solution.

An excellent reference to memory system design in the context of chip multiprocessors is [9]. [13] proposes an optimal memory allocation technique based on ILP for application-specific SoCs. The closest to our work is the flexible SPM design for MPSoCs investigated in [10, 16, 17]. However, they focus on data parallelism as opposed to task parallelism in the context of homogeneous multiprocessors. Therefore, they do not need to explore the impact of scheduling on the SPM design.

5. INTEGER LINEAR PROGRAMMING (ILP) FORMULATION

In this section we present the integer linear programming (ILP) formulation for an integrated task mapping/scheduling, scratchpad memory (SPM) partitioning, and data allocation. We assume that the application is specified as a task graph. We first formulate the problem of scheduling the tasks on multiple processors without considering the presence of SPM. This formulation is then extended to handle the pipelined scheduling. Finally, we formulate the problem of SPM partitioning and data allocation and integrate it with the formulation for task scheduling.

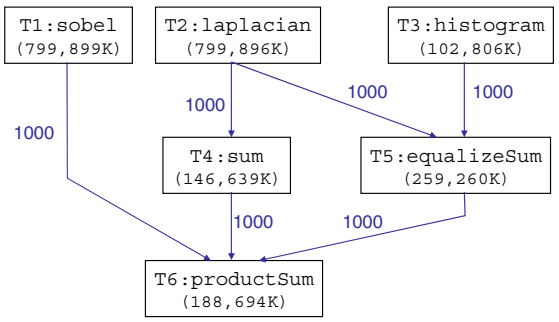


Figure 5: An example task graph. The numbers in brackets indicate the execution time of the tasks (in cycles). We assume homogeneous processors for this example. The edges are labeled with the communication costs (in cycles) between the tasks.

Throughout our discussion on task mapping and scheduling, we will use the task graph shown in Figure 5 for illustration purposes. For simplicity of illustration, we assume an MPSoC architecture consisting of four homogeneous processors so that the execution times of a task on all the processors are identical. However, we note that our formulation handles heterogeneous processors as well.

5.1 Task Mapping/Scheduling

We present here an ILP formulation to optimize performance through integrated task mapping and scheduling. We present extensions for pipelined scheduling and SPM partitioning in the next subsections.

Setting. The task graph for an application has N tasks denoted as $T_1, \ldots, T_N$. Without loss of generality, let T_N be the last task (the task without successors) in the task graph. If there are multiple last tasks in the task graph, then we add a dummy last task as the successor of all the original last tasks. We have M available processors (homogeneous or heterogenous) denoted as $P_1, \ldots, P_M$. Associated with each task is its execution time on each of the available processors — $time_{i,j}$ denotes the execution time of task T_i on processor P_j, assuming that all the data variables are available in off-chip memory (no SPM data allocation is considered at this point).

Let the binary decision variable $X_{i,j} = 1$ if task T_i is mapped to processor P_j and 0 otherwise. A task can be mapped to exactly one processor.

$$\sum_{j=1}^{M} X_{i,j} = 1 \tag{1}$$

The execution time of task T_i is given by

$$Time_i = \sum_{j=1}^{M} X_{i,j} \times time_{i,j} \tag{2}$$

Let $StartTask_i$ and $EndTask_i$ denote the starting time and the completion time, respectively, of task T_i. Then

$$EndTask_i = StartTask_i + Time_i - 1 \tag{3}$$

Objective Function. The objective is to minimize the critical path through the task graph. That is, the objective is to minimize the completion time $EndTask_N$ of the last task T_N.

Task Dependencies. Let $preds(T_i)$ denote the set of predecessors of task T_i in the task graph. T_i can only start execution after all the tasks $T_h \in preds(T_i)$ have completed execution. Further, if T_i and T_h are mapped to two different processors, then T_i has to wait for the completion of any data transfer from T_h to itself, incurring a communication cost $comm_{h,i}$.

We model inter-task communications as special tasks running on a shared bus. Let $C_{h,i}$ be the communication task between T_h and T_i, and let $StartComm_{h,i}$ and $EndComm_{h,i}$ be the starting time and the completion time of $C_{h,i}$. Then we have the following constraints.

$$StartComm_{h,i} \geq EndTask_h + 1 \tag{4}$$

$$StartTask_i \geq EndComm_{h,i} + 1 \tag{5}$$

Note that task dependencies are indirectly enforced via the communications between the tasks. To reflect the fact that the communication cost between T_h and T_i is incurred only when T_i and T_h are mapped to different processors, we have the following constraint.

$$EndComm_{h,i} = StartComm_{h,i} + L_{h,i} \times comm_{h,i} - 1 \tag{6}$$

where $L_{h,i} = 1$ if and only if T_h and T_i are mapped to different processors. The linearization of this definition is given in the Appendix.

Resource Constraint. The previous constraints effectively prevent two dependent tasks from competing for processor time. We should also ensure that any two *independent* tasks mapped to the same processor have disjoint lifetimes. Mirroring our treatment of dependent tasks, for every pair of independent tasks T_i and $T_{i'}$, let the binary variable $L_{i,i'} = 1$ if and only if T_i and $T_{i'}$ are mapped to different processors. If $L_{i,i'} = 0$, either task T_i executes before $T_{i'}$ or vice versa. Let binary variable $B_{i',i} = 0$ if T_i and $T_{i'}$ are mapped to the same processor and $T_{i'}$ executes after T_i. Similarly let $B_{i,i'} = 0$ if T_i and $T_{i'}$ are mapped to the same processor and

T_i executes after $T_{i'}$. Then we ensure disjoint lifetime for the two tasks through the following constraints.

$$B_{i,i'} + B_{i',i} - L_{i,i'} = 1 \quad (7)$$

$$StartTask_i \geq EndTask_{i'} - \infty \times B_{i,i'} + 1 \quad (8)$$

$$StartTask_{i'} \geq EndTask_i - \infty \times B_{i',i} + 1 \quad (9)$$

As all the communications take place on a shared bus, we should also ensure that the communications do not overlap with each other. Analogous to constraints (7) through (9), for all pairs of distinct communication tasks $C_{h,i}$ and $C_{f,g}$, we have the following constraints.

$$V_{h,i,f,g} + V_{f,g,h,i} = 1 \quad (10)$$

$$StartComm_{h,i} \geq EndComm_{f,g} - \infty \times V_{h,i,f,g} + 1 \quad (11)$$

$$StartComm_{f,g} \geq EndComm_{h,i} - \infty \times V_{f,g,h,i} + 1 \quad (12)$$

Here binary variable $V_{h,i,f,g} = 1$ if $C_{f,g}$ happens after $C_{h,i}$ and 0 otherwise. Similarly, binary variable $V_{f,g,h,i} = 1$ if $C_{h,i}$ happens after $C_{f,g}$ and 0 otherwise.

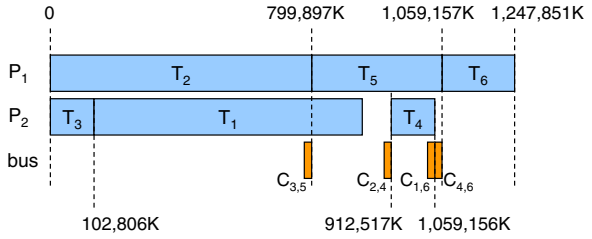


Figure 6: An optimal non-pipelined schedule for the task graph in Figure 5 on four processors. Processors P_3 and P_4 are not assigned any task.

EXAMPLE 1. *Figure 6 shows an optimal schedule for the task graph shown in Figure 5 on four processors obtained using the ILP formulation. Note that only two out of four available processors are utilized. This is because utilizing more processors will increase the length of the critical path through additional communication costs. We assume that the computation and communication can proceed in parallel (e.g., execution of task T_1 on processor P_2 in parallel with communication $C_{3,5}$ from processor P_2 to P_1). Also note that the tasks on processor P_1 determine the critical path; therefore, tasks on processor P_2 are scheduled with slacks.*

5.2 Pipelined Scheduling

In this section, we extend the task scheduling formulation in the previous subsection to take into account pipelined scheduling. In a synchronous pipelined execution, tasks are distributed across pipeline stages of uniform length. The length of the pipeline stages, called *Initiation Interval* (II), is determined by the maximum time needed to complete all the tasks in any of the stages. Thus the objective of pipelined scheduling is to distribute tasks into stages so as to minimize the II, while respecting task dependencies and resource constraints.

EXAMPLE 2. *Figure 7 shows an optimal pipelined schedule for the task graph in Figure 5 on four processors. Comparing this pipelined schedule with the non-pipelined schedule in Figure 6, we can notice the difference in the objective function. The time taken to execute a single instance of the task graph increases from 1,248M cycles for non-pipelined execution to 1,600M cycles for pipelined execution (Figure 7(a)). However, in the steady state the pipelined execution can process a task graph every 800M cycles (Figure 7(b)), which is the II for this schedule. That is, the throughput increases significantly compared to the non-pipelined execution.*

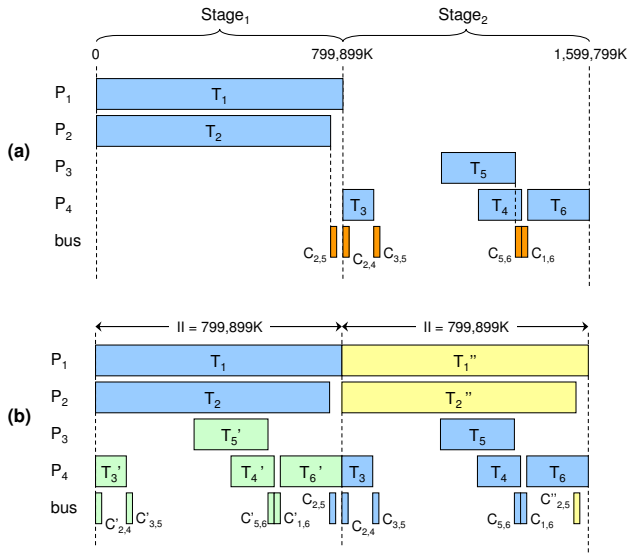


Figure 7: An optimal pipelined schedule for the task graph in Figure 5 with (a) single-instance execution view, and (b) steady-state execution view. Non-primed labels indicate tasks/communications from the current instance, primed labels indicate tasks/communications from the previous instance, while double-primed labels indicate tasks/communications from the next instance.

An important constraint that becomes apparent here is that any processor can be exploited in exactly one pipeline stage, because all the stages will execute in parallel (on different task instances) in the steady state. On the other hand, a stage can exploit more than one processor. This also implies that the maximum number of pipeline stages we can have is equal to the number of processors, M. Based on this observation, we extend our previous formulation to solve the problem of pipelined scheduling. Our strategy is to perform task mapping and scheduling onto processors as before, and then assign processors to the pipeline stages. Communication tasks, which are scheduled on a shared bus, will be mapped directly to pipeline stages as we will elaborate later.

Let the binary variable $W_{j,s} = 1$ if processor P_j is assigned to s^{th} pipeline stage (denoted $Stage_s$). Each processor is mapped to exactly one pipeline stage.

$$\sum_{s=1}^{M} W_{j,s} = 1 \quad (13)$$

Note that in the summation term we have implicitly defined the number of pipeline stages to be M, the maximum allowed number. This is necessary to keep the formulation linear. The solution may contain stages which have no processor assigned to them (that is,

one of the other stages may have more than one processor assigned to it); these are 'invalid' stages which will be disregarded.

Objective Function. The objective function is to minimize the Initiation Interval II, which is determined by the longest time needed to complete all the tasks in any of the stages. Let $StartStage_s$ and $EndStage_s$ denote the starting and completion time, respectively, of $Stage_s$. Then

$$II \geq EndStage_s - StartStage_s + 1 \quad \forall s : 1 \ldots M \tag{14}$$

Overlap among Pipeline Stages. A pipeline stage must not overlap with another stage. Analogous to constraints (7) through (9), for all pairs of stages $Stage_s$ and $Stage_t$, we have the following constraints.

$$B'_{s,t} + B'_{t,s} = 1 \tag{15}$$

$$StartStage_s \geq EndStage_t - \infty \times B'_{s,t} + 1 \tag{16}$$

$$StartStage_t \geq EndStage_s - \infty \times B'_{t,s} + 1 \tag{17}$$

Here $B'_{s,t} = 1$ if $Stage_t$ executes after $Stage_s$ and 0 otherwise. Similarly $B'_{t,s} = 1$ if $Stage_s$ executes after $Stage_t$ and 0 otherwise.

Length of Pipeline Stages. Now we express the constraints on the length of each pipeline stage. The length of $Stage_s$ has to encompass the entire execution period of the processor(s) assigned to it. For all pipeline stages $s : 1 \ldots M$ and all processors $j : 1 \ldots M$ we require:

$$StartStage_s \leq StartProc_j + \infty \times (1 - W_{j,s}) \tag{18}$$

$$EndStage_s \geq EndProc_j - \infty \times (1 - W_{j,s}) \tag{19}$$

where $StartProc_j$ and $EndProc_j$ denote the starting and completion time of processor P_j's execution, which in turn are determined by the earliest start time and the latest end time over all the tasks mapped to processor P_j. For example, the execution time of processor P_4 in Figure 7(a) spans from the start of task T_3 until the completion of task T_6. Recall that in the scheduling constraint the decision variable $X_{i,j}$ has value 1 if T_i is scheduled on P_j and 0 otherwise. For all processors $j : 1 \ldots M$ and all tasks $i : 1 \ldots N$:

$$StartProc_j \leq StartTask_i + \infty \times (1 - X_{i,j}) \tag{20}$$

$$EndProc_j \geq EndTask_i - \infty \times (1 - X_{i,j}) \tag{21}$$

Overlap among Communication Tasks. Communication tasks must also be accounted for in the pipeline stages. Unlike normal tasks, all the communications take place on a shared bus, *which will be utilized in all the stages*. As illustrated in Figure 7(b), communications from different pipeline stages (i.e., from different instances of the task graph) execute simultaneously within an II. Constraints (10) through (12) only ensure that the communication tasks within a single stage (i.e., from the same instance of the task graph) do not overlap with each other. However, we now need to ensure that the communication tasks do not overlap *across stages*.

This is accomplished by first *normalizing* the execution intervals of the communication tasks. The normalized interval of a communication task is its start/completion time relative to the start time of the pipeline stage that it is mapped onto. For example, the interval of communication task $C_{2,4}$ in Figure 7(a) is $[799, 899K, 799, 900K]$, while its normalized interval is $[0, 1000]$, i.e., relative to the start of $Stage_2$.

Let us define a binary variable $F_{h,i,s} = 1$ if $C_{h,i}$ is mapped to stage $Stage_s$ and 0 otherwise.

$$\sum_{s=1}^{M} F_{h,i,s} = 1 \tag{22}$$

Each communication task is then included in the interval of the stage it is mapped to.

$$StartStage_s \leq StartComm_{h,i} + \infty \times (1 - F_{h,i,s}) \tag{23}$$

$$EndStage_s \geq EndComm_{h,i} - \infty \times (1 - F_{h,i,s}) \tag{24}$$

Finally, we require mutual exclusion for all pairs of distinct communication tasks $C_{h,i}$ and $C_{f,g}$:

$$(StartComm_{h,i} - StartStage_s) \geq (EndComm_{f,g} - StartStage_t) - \infty \times V'_{h,i,s,f,g,t} + 1 \tag{25}$$

$$(StartComm_{f,g} - StartStage_t) \geq (EndComm_{h,i} - StartStage_s) - \infty \times V'_{f,g,t,h,i,s} + 1 \tag{26}$$

where $V'_{h,i,s,f,g,t} = 0$ ($V'_{f,g,t,h,i,s} = 0$) if and only if $C_{h,i}$ is scheduled in $Stage_s$, $C_{f,g}$ is scheduled in $Stage_t$, and the normalized interval of $C_{h,i}$ is scheduled after (before) the normalized interval of $C_{f,g}$. The linearization of this definition is given in the Appendix.

5.3 SPM Partitioning and Data Allocation

We now present the ILP formulation for the SPM partitioning and data allocation problem. Let the total number of variables for all the tasks be R. Some variables may be shared by several tasks. Associated with each variable v is its size in bytes, denoted $area_v$, and the number of times it is accessed in each task, obtained through profiling. Note that this value can be different depending on which processor the task gets mapped to (due to difference in ISA as well as compiler). Let $freq_{v,i,j}$ specify the number of accesses of variable v by task T_i when executing on processor P_j. Each one of these accesses incurs different latencies depending on the location of v:

- 0, if v is located in the private SPM of P_j, or
- a constant latency of $cross\_penalty$, if v is located in SPM of another processor (remote SPM), or
- a constant latency of $penalty$, if v is located in the off-chip memory

where $cross\_penalty$ will generally be less than $penalty$.

Let the binary decision variable $S_{v,j} = 1$ if variable v is allocated in the SPM of processor P_j and 0 otherwise. In our architectural model, a variable can be mapped to at most one processor's SPM.

$$\sum_{j=1}^{M} S_{v,j} \leq 1 \tag{27}$$

We have a constraint on the total available SPM area. Let $total\_area$ be the total available SPM area given as input to this problem.

$$\sum_{v=1}^{R} \sum_{j=1}^{M} S_{v,j} \times area_v \leq total\_area \tag{28}$$

Objective Function. The objective of task scheduling with SPM configuration and data allocation is also to minimize the overall completion time $EndTask_N$ where T_N is the last task, or in the pipelined setting, the initiation interval II, where II is specified as in constraint (14).

Task Execution Time. The only effect of data allocation to on-chip memory is that the execution time of each task is potentially reduced. Previously, each task takes constant execution time — $time_{i,j}$ denotes the execution time of task T_i on processor P_j assuming all the variables are in off-chip memory. Then, the execution time of a task was given by Equation (2). Now we replace this equation with

$$Time_i = \sum_{j=1}^{M}(X_{i,j} \times time_{i,j} - \sum_{v=1}^{R} freq_{v,i,j} \times gain_{v,i,j}) \quad (29)$$

$$gain_{v,i,j} = Y_{v,i,j} \times penalty + Z_{v,i,j} \times (penalty - cross\_penalty) \quad (30)$$

where $Y_{v,i,j} = 1$ if and only if task T_i and variable v have both been mapped to processor P_j; and $Z_{v,i,j} = 1$ if and only if T_i has been mapped to processor P_j and variable v has been mapped to the SPM of a processor other than P_j. In other words,

$$Y_{v,i,j} = 1 \Leftrightarrow ((X_{i,j} = 1) \quad AND \quad (S_{v,j} = 1))$$

$$Z_{v,i,j} = 1 \Leftrightarrow ((X_{i,j} = 1) \quad AND \quad (\exists k \neq j \quad S_{v,k} = 1))$$

The above two constraints can be linearized as shown in the Appendix.

6. EXPERIMENTAL EVALUATION

The ILP formulation for integrated pipelined task scheduling, SPM partitioning and data allocation generates the optimal solution. The goal of our ILP formulation is to explore the interaction among the different stages of the design space exploration process. This helps us to identify the performance limit of MPSoC architectures for embedded applications.

As explained in Section 3, we attempt three different techniques for optimal data allocation to scratchpad memory in increasing order of flexibility and complexity.

EQ: Task scheduling ignores the effect of data allocation to SPM and the on-chip SPM budget is equally partitioned among the different processors. This is simply a Knapsack problem, for which optimal solutions (e.g., dynamic programming) are known.

PF: Task scheduling ignores the effect of data allocation to SPM. However, SPM partitioning and data allocation are performed simultaneously through a simplified ILP formulation derived from Section 5. As task scheduling is performed a-priori, the assignments to the ILP variables X, B, V, W, F, and V' are known. In other words, the design space is more restricted.

CF: Task scheduling, SPM partitioning and data allocation are performed simultaneously through the ILP formulation described in Section 5.

6.1 Experimental Setup

We use five applications in our experiments, four of which are taken from embedded benchmark suites MiBench and Mediabench. `cjpeg`, `mpeg2enc` and `osdemo` from Mediabench perform JPEG encoding, MPEG-2 encoding and 3D graphics rendering (part of Mesa 3D graphics library), respectively. `lame` is an MP3 encoder from the MiBench consumer suite. `enhance`, the fifth benchmark, is a slightly modified version of the image enhancement application from [22]. We profile the applications to identify the key computation blocks. Each application is then divided into a number of tasks where each task corresponds to a computation block. The control/data flow information are used to identify the dependencies among the tasks and estimate the communication costs. This process generates the task graph for each application.

Benchmark	# Tasks	# Vars	Total var size (bytes)	Avg var size (bytes)
enhance	6	45	7,355,920	163,464
lame	4	124	301,672	2,432
mpeg2enc	6	30	12,744	424
osdemo	6	45	80,000	1,777
cjpeg	4	20	702,296	35,114

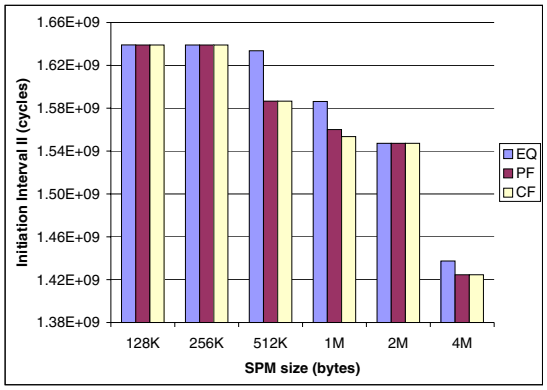
Table 1: Characteristics of the benchmarks.

We use the SimpleScalar cycle-accurate architectural simulation platform [2] for our experiments. We use an instrumented version of the SimpleScalar profiler to extract the variable sizes and access frequencies as well as the execution time (in processor cycles) of each task individually. As mentioned earlier, the profiler assumes that off-chip access latencies are constant, and does not account for bus conflicts. We consider both scalar and array variables for allocation to scratchpad memories. All the shared variables (i.e., variables that are written by one task and read by another) contribute towards the communication cost in the task graph. These shared variables are not considered for allocation into the scratchpad memories. Table 1 shows the characteristics of the benchmarks. The profile is shown only for the non-shared (i.e., non-communicating) variables. In this work, we choose data variables for allocation, but our strategy can be applied to blocks of program code as well.

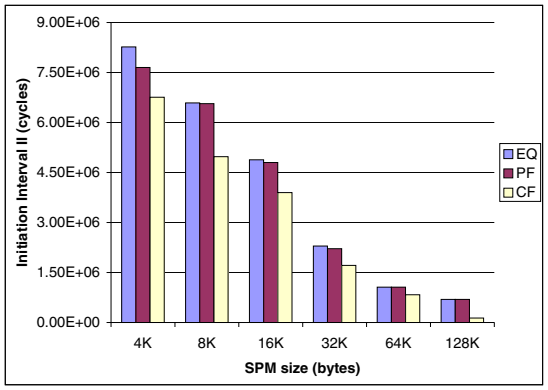
We use a 2-processor configuration for the experiments. Total on-chip scratchpad memory budget varies from 256 bytes to 4MB for different benchmarks. We use smaller SPM budgets for applications with smaller total memory requirement (e.g., `mpeg2enc`) and larger SPM budget for applications with large average variable size (e.g., `enhance` with 163KB). We assume 100-cycle latency for off-chip memory access and 4-cycle latency for accessing a remote SPM. For the ILP-based techniques, our method constructs the ILP formulation for (separate or integrated) task scheduling and data allocation as described in Section 5 and inputs this to CPLEX, a commercial ILP solver. Upon solving the ILP problem, CPLEX returns the objective value as well as the valuation of the decision variables that leads to the objective. The experiments are conducted on a 3.0 GHz Pentium 4 CPU with 2GB of memory.

6.2 Results

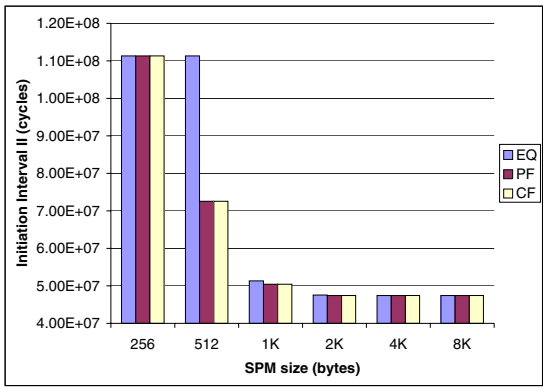
Figure 8 shows the initiation interval (II) obtained by applying *EQ*, *PF*, and *CF* techniques on the five benchmarks with varying SPM sizes. First let us compare *EQ* with *PF*. The main advantage of *PF* is that it allows flexible partitioning of SPM space among the processors. This flexibility can potentially improve the performance dramatically over *EQ*. For example, *PF* results in up to 35% performance improvement for `mpeg2enc`, 53% improvement for `osdemo`, and 60% improvement for `cjpeg` over *EQ*. `lame` enjoys a modest improvement of up to 7.5% due to the flexible SPM allocation. The only exception is `enhance`, which achieves very little improvement due to `PF` strategy. This is because `enhance`



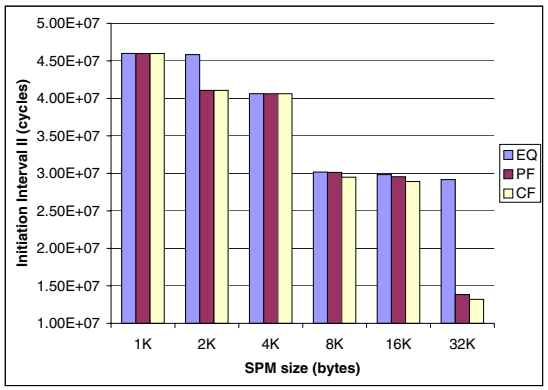
(a) enhance



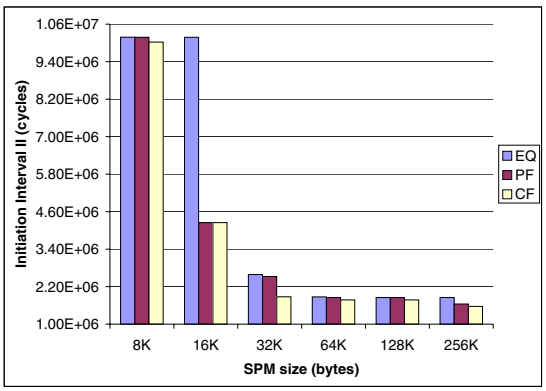
(b) lame



(c) mpeg2enc



(d) osdemo



(e) cjpeg

Figure 8: Initiation Interval (II) for the different benchmarks with EQ, PF, and CF strategies and varying on-chip SPM budgets. This setup corresponds to a 2-processor configuration.

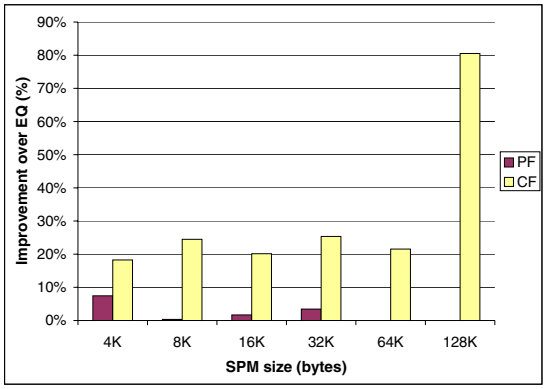


Figure 9: Improvement in Initiation Interval II due to PF and CF over EQ for benchmark `lame`.

has large variables that are harder to allocate. This is also clear from the fact that the II hardly improves with increasing SPM sizes.

It is important to note that flexibility is most important when resources are not too limited or too generous. With a restricted SPM budget, there is not much room for improvement irrespective of the SPM partitioning strategy. Similarly, when the SPM budget is bigger than the one necessary to accommodate all the frequently accessed variables, the strategy employed for SPM partitioning becomes immaterial. With a larger SPM budget, `PF` allocates more variables than `EQ`; but these variables are accessed less frequently. `PF` strategy shows maximum improvement when the on-chip SPM budget is neither too big nor too small. In those cases, only the most important variables should be accommodated and the flexibility guarantees that the most important variables can indeed be allocated.

We now compare *PF* with *CF*. The results indicate that the improvements depends heavily on the characteristics of the applications. For example, `lame` achieves as high as 80% improvement over *PF* by considering data allocation during scheduling. This is highlighted in Figure 9 that shows the performance improvement of *PF* and *CF* over *EQ*. Similarly, `cjpeg` enjoys up to 25% additional performance improvement. `osdemo` shows 2–5% improvement. `enhance` and `mpeg2enc`, on the other hand, show hardly any improvement. As explained before, `enhance` has larger variables that are difficult to allocate. `mpeg2enc` is a compute-intensive application with significant amount of communication among the tasks. As shown in Table 1, efficient use of SPM space is not important because non-shared variable accesses do not contribute significantly towards the execution time.

Finally, Table 2 shows the runtime for our scheduling and allocation techniques. *Schedule* denotes the task scheduling time irrespective of SPM consideration. This is required as input to *EQ* and *PF*. The data allocation time for *EQ* as well as SPM partitioning and data allocation time for *PF* assuming a given task schedule generated a-priori are shown in the next two columns. Finally the column `CF` gives the runtime for integrated task scheduling, SPM partitioning and data allocation. We show both the best-case runtime and worst-case runtime for each benchmark. We observe that the worst-case occurs when the SPM budget is neither too big nor too small. This is expected as these cases are the most difficult ones to schedule. Note that for `enhance` with 4MB SPM budget and `lame` with 16KB, 128KB SPM budget, the ILP solver did not terminate within 20 minutes (shown as 20+ min in the table). For these cases, we use the intermediate solution returned by the ILP solver. Interestingly, even the intermediate solutions obtained for *CF* strategy outperforms the optimal result obtained using *PF* strategy (Figure 8). This clearly indicates that it is important to consider memory optimization during task scheduling.

7. CONCLUSION

In this paper, we explore optimization of scratchpad memory (SPM) in the context of embedded chip multiprocessors. We propose an ILP formulation to capture the interaction between task scheduling and memory optimization. We show that (1) flexible partitioning of the SPM budget among the processors can achieve up to 60% performance improvement compared to equal partitioning and (2) integrating memory optimization with task scheduling can improve performance by up to 80%. In the future, we plan to use the optimal performance limits as guidelines to design effective heuristics.

8. ACKNOWLEDGMENTS

This work was partially supported by National University of Singapore research grant R252-000-171-112.

9. REFERENCES

[1] F. Angiolini, L. Benini, and A. Caprara. Polymonial-time algorithm for on-chip scratchpad memory partitioning. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2003.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[3] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1), 2002.

[4] R. Banakar et al. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *International Conference on Hardware-Software Codesign (CODES)*, 2002.

[5] L. Benini, D. Bertozzi, A. Guerri, and M. Milano. Allocation and scheduling for mpsocs via decomposition and no-good generation. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 2005.

[6] K. S. Chatha and R. Vemuri. Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Transactions on VLSI*, 10(3), 2002.

[7] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 2005.

[8] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[9] M. Kandemir and N. Dutt. Memory systems and compiler support for mpsoc architectures. In A. Jerraya and W. Wolf, editors, *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, 2005.

[10] M. Kandemir, J. Ramanujam, and A. Choudhury. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *Design Automation Conference (DAC)*, 2002.

[11] S-R. Kuang, C-Y. Chen, and R-Z. Liao. Partitioning and pipelined scheduling of embedded system using integer

Benchmark	Best runtime (sec)				Worst runtime (sec)			
	Schedule	EQ	PF	CF	Schedule	EQ	PF	CF
enhance	12.9	0.01	0.01	23.60	13.07	0.32	0.06	20+ mins
lame	0.20	0.03	0.04	15.26	0.22	0.81	0.52	20+ mins
mpeg2enc	1.56	0.01	0.01	3.75	6.69	0.02	0.07	69.98
osdemo	2.81	0.02	0.01	105.16	2.86	0.04	0.06	1467.33
cjpeg	0.33	0.01	0.01	0.55	0.37	0.04	0.05	3.6

Table 2: Best-case and worst-case runtime for the different benchmarks

linear programming. In *International Conference on Parallel and Distributed Systems (ICPADS)*, 2005.

[12] Y-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3), 1999.

[13] S. Meftali, F. Gharsalli, F. Rousseau, and A. A. Jerraya. An optimal memory allocation for application-specific multiprocessor system-on-chip. In *International Symposium on Systems Synthesis (ISSS)*, 2001.

[14] G. De Micheli, R. Ernst, and W. Wolf. Readings in Hardware/Software Co-Design. Morgan Kaufmann, 2002.

[15] R. Niemann and P. Marwedel. Hardware/software partitioning using integer programming. In *Design, Automation and Test in Europe (DATE)*, 1996.

[16] O. Ozturk et al. Customized on-chip memories for embedded chip multiprocessors. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2005.

[17] O. Ozturk et al. An integer linear programming based approach to simultaneous memory space partitioning and data allocation for chip multiprocessors. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2006.

[18] P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1999.

[19] 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 Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3), 2000.

[20] J. Sjodin and C. von Platen. Storage allocation for embedded processors. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001.

[21] S. Steinke, L. Wehmeyer, B-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, Automation and Test in Europe (DATE)*, 2002.

[22] F. Sun, N. K. Jha, S. Ravi, and A. Raghunathan. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *International Conference on VLSI Design*, 2005.

[23] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3), 1990.

Appendix: Linearization of constraints

In this section, we present the linearization of some of the constraints in our ILP formulation.

1. **$\mathbf{L_{h,i} = 1}$ iff $\mathbf{T_h}$ and $\mathbf{T_i}$ are mapped to different processors**

   Recall that $X_{i,j} = 1$ if task T_i is mapped to processor P_j and 0 otherwise.

   $$\forall j : 1 \ldots M \quad L_{h,i} \leq 2 - X_{h,j} - X_{i,j}$$

   $$\forall j : 1 \ldots M \quad \forall k : 1 \ldots M, k \neq j \quad L_{h,i} \geq X_{h,j} + X_{i,k} - 1$$

2. **$\mathbf{V'_{h,i,s,f,g,t} = 1}$ ($\mathbf{V'_{f,g,t,h,i,s} = 1}$) iff $\mathbf{F_{h,i,s} = 1}$ and $\mathbf{F_{f,g,t} = 1}$ and the normalized interval of $\mathbf{C_{g,h}}$ is scheduled after (before) the normalized interval of $\mathbf{C_{h,i}}$**

   $$V'_{h,i,s,f,g,t} + V'_{f,g,t,h,i,s} + F_{h,i,s} \geq 2$$

   $$V'_{h,i,s,f,g,t} + V'_{f,g,t,h,i,s} + F_{f,g,t} \geq 2$$

   $$V'_{h,i,s,f,g,t} + V'_{f,g,t,h,i,s} + F_{h,i,s} + F_{f,g,t} \leq 3$$

3. $\mathbf{Y_{v,i,j} = 1} \Leftrightarrow (\mathbf{(X_{i,j} = 1)} \quad \mathbf{AND} \quad \mathbf{(S_{v,j} = 1)})$

   $$Y_{v,i,j} \leq X_{i,j}; \quad Y_{v,i,j} \leq S_{v,j}; \quad Y_{v,i,j} \geq X_{i,j} + S_{v,j} - 1$$

4. $\mathbf{Z_{v,i,j} = 1} \Leftrightarrow (\mathbf{(X_{i,j} = 1)} \quad \mathbf{AND} \quad \mathbf{(\exists k \neq j \quad S_{v,k} = 1)})$

   In this case, we need to introduce an additional binary variable $U_{v,j} = 1$ iff $\exists k \neq j \quad S_{v,k} = 1$. We first linearize the definition of $U_{v,j}$.

   $$\sum_{k=1,k\neq j}^{M} S_{v,k} - \infty \times U_{v,j} \leq 0; \qquad \sum_{k=1,k\neq j}^{M} S_{v,k} - U_{v,j} \geq 0$$

   Then we linearize the original constraint in terms of $U_{v,j}$.

   $$Z_{v,i,j} \leq X_{i,j}; \quad Z_{v,i,j} \leq U_{v,j}; \quad Z_{v,i,j} \geq X_{i,j} + U_{v,j} - 1$$