A Task-Level Hierarchical Memory Model for System Synthesis of Multiprocessors

Yanbing Li and Wayne Wolf
Dept. of EE, Princeton University, Princeton, NJ 08544.
email: {yanbing,wolf}@ee.princeton.edu

Abstract This paper introduces the first **high-level (task-level)** model of hierarchical memories and describes a scheduling and allocation algorithm for system-level synthesis of heterogeneous multiprocessors. Caches are essential for modern RISC embedded cores to obtain sustained high performance. However, caches have received limited use in priority-driven preemptive real-time systems due to the unpredictability of caches—average-case improvements are of no use in systems with hard deadlines. Program-level cache models do not take into account **preemptions** between multiple tasks running at multiple rates on embedded cores. Our task-level model of performance in the presence of memory hierarchies provides an efficient means to bound the guaranteed memory performance of tasks running in a **multi-rate, multi-tasking** environment. Our system synthesis algorithm uses software-based **cache partitioning** and reservation techniques to guarantee cache hits for some tasks and therefore improve task schedulability. Experimental results show that our model significantly improves schedulability of real-time tasks and can be evaluated efficiently during system-level synthesis.

1 Introduction

This paper describes a new **task-level hierarchical memory model** and a new **system-level synthesis algorithm** for hierarchical-memory systems. The tasks to be scheduled are periodic real-time tasks running at multiple rates. The architecture is a multiprocessor system with memory hierarchy. Our algorithm allocates the tasks to the processing elements (PEs) and constructs a priority-driven preemptive schedule which guarantees that all tasks meet their deadlines. Our system synthesis algorithm takes advantage of the memory hierarchy in the architecture to significantly improve the schedulability of the task set. Our scheduling and allocation algorithm can be used as a building block of hardware-software co-synthesis, or used directly in real-time system design.

With embedded CPU cores becoming increasingly common in VLSI systems—and with increasing use of multiple embedded cores on a single chip—system designer increasingly need to implement major subsystems using real-time system design techniques such as multiple, prioritized tasks sharing CPUs. Most system-level scheduling algorithms use the estimated **worst case execution time (WCET)** of the tasks to constructing a feasible schedule or as a measure of the **schedulability** of the tasks. However, a WCET computed with a no-cache assumption is very pessimistic for a system with caches. With the presence of caches, a task can usually achieve a much smaller WCET than when there is no cache. But the smaller WCET is not guaranteed in a priority-driven preemptive system because the unpredictable preemptions and interrupts keep changing the cache state and make it hard or even impossible to predict the cache behavior and the performance gain that a cache can achieve. While a real-time system needs to meet the hard deadlines, it cannot benefit from the smaller WCET estimation if it is not guaranteed.

DAC 97, Anaheim, California

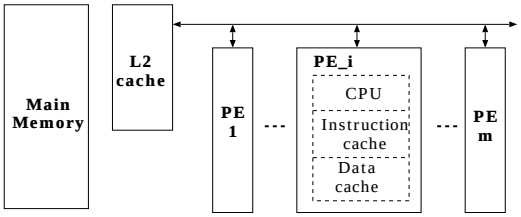

Figure 1: An example multiprocessor with memory hierarchy.

Recent research, such as work by Li, Malik & Wolfe[1], has developed hierarchical memory models for analyzing the performance of a single program. While such models provide accurate estimates of the performance of a single program, they do not take into account the effects of **preemptions** between multiple tasks, and they are much too expensive to be used in system-level synthesis and design exploration. When one task preempts another, it may change the state of the cache at a point in a way that compromises the performance of the originally-executing model. Such interactions are critical to evaluate during system-level architecture design.

In a uniprocessor environment, real-time systems commonly use one of two scheduling policies to schedule periodic tasks: *earliest-deadline-first (EDF)* and *rate-monotonic scheduling (RMS)*[7]. Many groups have developed allocation/scheduling algorithms for distributed real-time systems(Peng & Shin[6]; Burchard,etc[5]; Li & Wolf[2]). Their algorithms use WCETs as scheduling measure and do not consider the effects of memory hierarchies. Some research analyzes cache behaviors and their effects on real-time scheduling. Torrellas[4] studied cache performance of multiprogrammed workloads. His work aims to improve the overall performance and is not suitable in real-time scheduling which requires tasks to meet hard deadlines. Kirk and Strosnider[3] developed a SMART cache design that partition the cache to provide predictable cache performance. Their work targets a uniprocessor one-level cache architecture and requires to modify the cache design in hardware. The algorithm performs complicated program-level analysis, while our algorithm works at the task-level and does not look at the internal structural of the tasks while scheduling.

2 Problem Specification and Task Model

The problem specification includes two components: an architecture and a set of tasks. The algorithm needs to allocate the tasks to the PEs in the architecture and construct a feasible schedule which guarantees all tasks meet their deadlines. Fig.1 shows an example architecture. It is a multiprocessor system with each pro-

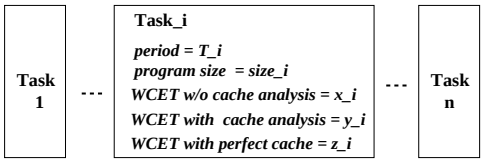

Figure 2: A set of multi-rate tasks.

cessor has its local first-level instruction cache and data cache. The system also has shared lower-level memories such as level-2 cache and main memory. For simplicity, we make the following assumptions about the caches:

1. Tasks are well-behaved in the combination of level-1 and level-2 cache.
2. Only instruction cache is modeled.

Assumption 1 would not be reasonable in a general-purpose system, but it is plausible for many high-performance embedded systems. The kernels of time-critical operations are frequently small enough to fit into a modest-sized cache. Deep submicron chips are large enough to include large level-2 cache. The techniques we will present (Section 3) such as cache partition/reservation and cache state modeling can be implemented at level-1 caches as well as lower level caches or the combination multiple levels in the memory hierarchy.

As shown in Fig.2, the tasks are a set of periodic tasks with each task runs independently at a different rate. Each task is characterized by the following parameters:

- **Task period**. We assume that the deadlines of a task is the same as its period.
- **Task size**: the code size of the task's program.
- **Worst case execution times (WCETs)** in several situations:
  - **WCET without caches**: estimated assuming the caches are turned off.
  - **WCET with caches**: estimated with the cache, assuming that the instruction cache is big enough to accommodate the whole program for a tasks and therefore only compulsory misses happen.
  - **WCET with perfect caches**: estimated assuming the whole program resides in the cache and there is no miss in any instruction memory access.

  If we need to use multiple levels of caches, *WCET with caches* and *WCET with perfect caches* should be extended to includes WCETs with different levels of caches.

These WCETs have the following relationship:

$$WCET^{w/o\_cache} \geq WCET^{cache} > WCET^{perfect\_cache} \quad (1)$$

3 Scheduling and Allocation Algorithm with Hierarchical Memory Model

Fig.3 shows the two-phase procedures of our algorithm.

1. **Allocating** the tasks to the PEs taking into account the workload balance and schedulability across the PEs.
2. Constructing a **schedule** for each PE. The scheduling algorithm uses a *deadline-based* priority-driven preemptive approach, with the following heuristics to improve results:
   - Use *cache partition/reservation* to reduce preemptions because they cause unpredictable cache flushing and reloading.
   - Use the combination of *static allocation* and *dynamic allocation*: a task with a cache reservation on a certain PE is statically allocated to that PE; otherwise the algorithm allows it reallocated from instance to instance to maximize the utilization of PEs.
   - Use *cache state model* to increase reuse of the cache contents and increase hits.

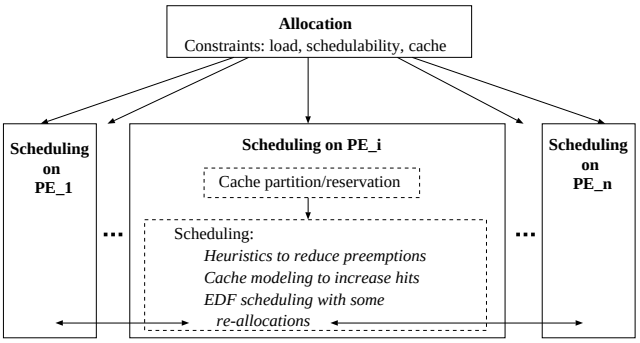

Figure 3: Allocation and scheduling procedure.

Inputs M: number of processors.

$Task(C, T)$: a task with period T and execution time C.

Global functions: $curr(m)$: returns current PE for class m.

$newproc()$: returns index of an empty processor.

Allocate(Task(C, T))

1. $m = \lfloor M(log_2(T) - \lfloor log_2(T) \rfloor) \rfloor + 1$
2. **if** $(load_{curr(m)} + C/T \leq 1 - ln2/M)$ *then*
3. $load_{curr(m)} = load_{curr(m)} + C/T$
4. **else if** $(C/T < load_{curr(m)})$ *then*
5. *curr(m) = newproc()*
6. $load_{curr(m)}$ *= C/T*
7. **else**
8. *x = newproc()*
9. $load_x$ *= C/T*
10. **endif**

Figure 4: On line task allocation algorithm by Burchard, etc.

3.1 Task Allocation

Many allocation algorithms for real-time tasks treat the processors as bins and use a bin-packing approach. The decision whether a processor is full is determined by some schedulability condition that is usually based on the *workload utilizations* (defined as a task's $WCET$ divided by its period) of the tasks packed into the processor. Our algorithm allocates as well as optimizes for cache partition to meet the workload utilization conditions. The heuristic algorithms for single-constraint bin-packing can not be applied directly to our problem with double-constraints. We used a two-step scheme:

1. Allocate tasks to PEs to meet the *utilization constraints*.
2. For each PE, partition and reserve caches to meet the local cache partition constraints (Section 3.2).

For the first step, we used the study from Burchard,etc[5]. They developed a tighter schedulability condition and a linear allocation algorithm (see Fig.4). Tasks are divided into M classes. Each processor is assigned tasks from only one class. The class membership of a task is determined (Line 1) by:

$$m = \lfloor M(log_2(T) - \lfloor log_2(T) \rfloor) \rfloor + 1 \quad (2)$$

If a new task $task_a$ from class m is added to the task set, the algorithm first attempts to accommodate $task_a$ to the current processor for class m (Line 2-3). If the attempt fails, $task_a$ is assigned to an empty processor. If the load of $task_a$ is sufficiently small (Line 4), the processor to which $task_a$ is assigned becomes the current processor of class m (Line 5). If the load factor of $task_a$ is large, no other task will be assigned to this processor. This procedure is repeated until all tasks are allocated.

3.2 Cache Partitioning

After the tasks are allocated to the PEs, the algorithm looks at each individual PE and the tasks allocated to it to construct a feasible schedule. To maximize the schedulability of the PE, the algorithm needs to find a partition of the cache associated with the PE

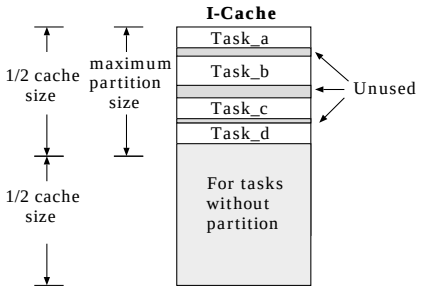

Figure 5: Cache partition and reservation.

that can maximumly decrease the total utilization of all the tasks allocated to the PE. Fig.5 shows how a partition looks like.

Let *A* be the set of tasks allocated to the PE, T_i the period of *task i*, $size_i$ the program size of *task i*, and *B* a possible partition - a set of tasks with reserved partitions on the PE and $B = \{S | S \in A\}$. To allow tasks that are not allocated a cache partition and interrupts to run efficiently, we put the following constraints on the maximum partitioned cache size: it cannot exceed half of the cache size and the unpartitioned portion should be large enough to accommodate the largest task that does not have a partition:

$$Actual\_partition\_size = \sum_{i \in B} size_i \quad (3)$$

$$Max\_partition\_size = min(1/2 \times Cache\_size, Cache\_size - max_{i \in A-B}(size_i)) \quad (4)$$

$$Actual\_partition\_size \leq Max\_partition\_size \quad (5)$$

Given these constraints, the goal of cache partition is to find a partition B such that the total utilization decrease is maximized:

$$max_B(\sum_{i \in B}(WCET_i - WCET_i^{perfect\_cache})/T_i) \quad (6)$$

Optimized B can be found using linear programming techniques. For computation simplicity, we uses a heuristic approach. Intuitively, the performance gain by reserving a partition for a task is the utilization decrease, at the cost of reserving the partition of $size(task)$, therefore, we define a *task partition priority (TPP)* as the priority measure of choosing tasks to reserve cache partitions.

$$TPP(task\_i) = \frac{(WCET_i - WCET_i^{perfect\_cache})}{T_i} \times \frac{1}{Size_i} \quad (7)$$

Tasks are assigned partitions according to their *TPPs* until all the partitionable cache has been assigned. This can be implemented at multiple levels of the memory hierarchy. In order for the tasks to map to their assigned cache partitions, the compiler should use the partition result and make sure that the tasks map to the addresses that are reserved for them. For tasks with cache partitions on a same cache, their addresses should not overlap. On a same cache, addresses of tasks without partitions may overlap each other, but they should not overlap the addresses of the reserved partitions.

3.3 Task Scheduling

After the cache partition, our algorithm uses a deadline-based priority-driven preemptive approach to construct a schedule for each PE. Unlike other algorithms that use *static allocation*, our algorithm gives freedom of re-allocation for those tasks that do not have a reserved cache partition on a PE. Re-allocation to an idle PE can increase the utilization of that PE and improve schedulability.

A simplified *cache state model* is introduced to encourage cache reuse. A cache state is a binary value that models whether the program for a task is located in a cache. For a task $task_i$ without cache partition, its program is loaded into the cache during an execution and maybe flushed out (*cache state = 0*) by tasks executed

```
   Scheduler
1. whenever_task_finish (task_i, PE_j) {
2.   if there are pending ready tasks on PE_j
3.       /* earliest-deadline first */
4.       schedule ready task w/ earliest deadline on PE_j
5.   else {
6.      /* try to find a ready task from other PEs and
7.         re-allocate it to the otherwise idled PE_j */
8.      re_allocate_task = NULL
9.      max_allocate_priority = 0
10.     for each ready task_a w/o cache reservation on
            other PEs {
11.       ap = RAP (task_a, PE_j)
12.       if (ap > max_allocate_priority) {
13.          re_allocated_task = task_a
14.          max_allocate_priority = ap
15.       }
16.     }
17.     if re_allocate_task is not NULL
18.       schedule re_allocate_task on PE_j
19.     else
20.       PE_j idle }
  }
21. whenever_task_arrive (task_i, PE_j) {
22.   task_r = running task on PE_j
23.   if (task_i has an earlier deadline than task_r)
24.       and (preemption is necessary)
25.      preempt task_r, insert it to ready queue of PE_j
26.   else if (task_r is re-allocated to PE_j from  PE_p)
27.      preempt task_r
28.      call whenever_task_arrive (task_r, PE_p)
29.   else
30.      insert task_i to the ready task queue of PE_j }
```

Figure 6: The scheduler invoked when a task arrives or finishes.

after $task_i$. However, if all these tasks executed after $task_i$ have their own cache reservation or map to different locations in cache from $task_i$, $task_i$'s program is still present in cache (*cache state = 1*). Therefore, by keeping $task_i$ on the same PE and executing it before its cache state is flushed, we can use a tighter bound of *WCET*(*WCET with perfect cache*) for this instance of $task_i$.

The decision of whether to re-allocate a task and where it is re-allocated to is made by *re-allocation-priority (RAP)*:

$$RAP(Task_i, PE_j) = -\alpha \times \Delta(i,j) + \beta \times Diff(i,j) \quad (8)$$

$$\Delta(i,j) = WCET^{w/o\_cache}_{task_i, old\_PE\_for\_task_i} - WCET^{w/o\_cache}_{i,j} \quad (9)$$

$$Diff(i,j) = cache\_state(i,j) \times (WCET^{w/o\_cache}_{i,j} - WCET^{perfect\_cache}_{i,j}) \quad (10)$$

The RAP for a task $Task_i$ on a PE PE_j is decides by 2 terms. The Δ-term considers the speed ($WCET$) difference of $task_i$ on its old PE and PE_j. The *Diff*-term considers the effect of cache state. α and β are preset constants within the range of [0,1] and they decide how much the re-allocation and cache state modeling will affect the schedule. A positive *RAP* indicates that the re-allocation is favorable. When a PE becomes idle, the scheduler will try to re-allocate tasks from other PEs by choosing a task with a maximum positive *RAP*.

Fig.6 shows the pseudo code of a scheduler. The scheduler is invoked whenever a task arrives or finishes. When a task finishes on a PE (Line 1-20), the scheduler either starts the next ready task in the queue of that PE, or as shown from Line 5-18, it compute RAPs and the scheduler decides whether it should re-allocate a task and which task to re-allocate. When a new task arrives, if it has an earlier deadline that executing task and preemption is checked to be *necessary*, it preempts the executing task instantly (Line 23-25). If the executing task is re-allocated from other PE, it has the lowest priority on its new PE and will be preempted and sent back its old PE - it is either inserted into ready queue or preempts that task running on that PE (Line 26-28).

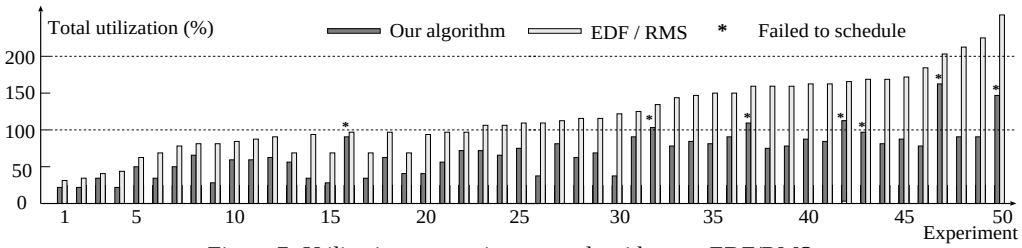

Figure 7: Utilization comparison: our algorithm vs. EDF/RMS

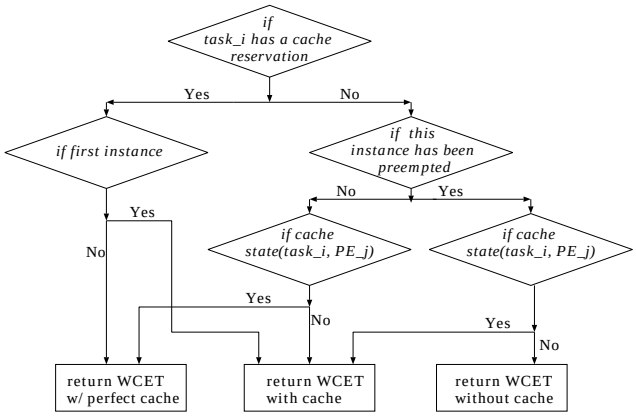

Figure 8: Which WCET to use in scheduling.

	Experiments		
Tasks	**Randomly generated 50 task sets**		
	number of tasks:	2-10	*uniform distribution*
	task period:	1-10,000	
	task size:	1/2 - 4k	
	x WCET w/o cache:	5-90% of task period	
	y WCET w/ cache:	25-100% of x	*Gaussian distribution*
	z WCET w/ perfect cache:	60-100% of y	
Architecture	1, 2 and 4 processors with private L1 caches and shared L2 caches.		

Table 1: Randomly generated experiments.

Another important issue in scheduling is which WCET to use at different points of the task executions in order to have a tight and accurate estimation of the task execution time. As shown in Fig.8, for tasks with reserved cache partitions, the decision is quite simple: for the first instance, use $WCET^{with\_cache}$, otherwise $WCET^{with\_perfect\_cache}$. For a task without cache reservation, the WCET used for the task may change in the course of scheduling. Both its preemption history and the cache state on its allocated PE are needed to make the decision.

4 Experimental Results

We have run a number of experiments on some examples and obtain promising results. One example is the MPEG-1 encoding algorithms. MPEG encoding involves both video and audio encoding that run at different rates (video: 30 frames per second, audio: sampling rate of 48 kHz). Our algorithm successfully schedules both audio and video on 4PEs. In comparison, EDF or RMS without cache modeling require at least 6 PEs.

To test the algorithm's performance in general, we randomly generated some task sets with some random parameters (see Table 1), then we try to schedule them onto 1,2 or 16 PEs with local level-1 caches and shared level-2 cache. For each task set, the number of tasks and their periods, size and *WCET* are generated by uniform distributions. *WCET with cache* and *WCET with perfect cache* are computed using *WCET* as a base value and modifying it by some percentages. The percentages are generated by Gaussian distributions with estimated parameters based on experimental results from Li,etc.[1]. Table 2 summarizes the results and Fig.7 shows the comparison of workload utilizations for the 50 experiments by using our algorithm and EDF/RMS. Our algorithm achieved a lower utilization and increased schedulability. For all the tasks that EDF and RMS can schedule so can our algorithm; for some tasks that EDF and RMS failed to schedule, our algorithm can find schedules; our algorithm does fail on some cases, because either the tasks have no feasible schedules at all (such as experiment 50) or in some cases (such as experiment 16), the tasks with very high utilizations have feasible schedules, but our algorithm, which is based on heuristics, is not guaranteed to find them.

	EDF	RMA	Our algorithm
Schedule successful rate	38%	40%	86% with static allocation 90% with partly dynamic allocation

Table 2: Scheduling results of the randomly generated experiments.

5 Conclusions

Our algorithm uses a hierarchical memory model for system synthesis and targets the scheduling of multi-rate tasks on multiprocessors. It achieves predictable caching and guaranteed hits for some tasks by cache partition and cache state modeling, therefore can use a tighter WCET for these tasks and greatly improves the schedulability of the tasks over existing algorithms. The task-level algorithm does not require detailed program analysis at run time, it is computationally efficient and can be used as a building block in hardware-software co-synthesis for design space exploration. Our algorithm can be used not only in making choice of the number and types of PEs, but also the choice of cache structure.

Future work may include: theoretically studying the bound of the schedulable workload utilization and the robustness of the algorithm in case of sporadic incoming tasks; and using our analysis procedure in the inner loop of a co-synthesis system.

References

[1] Y. Li, S. Malik and A. Wolfe. "Performance Estimation of Embedded Software with Instruction Cache Modeling", in Proc., *ICCAD '95*, IEEE Computer Society Press, 1995.

[2] Y. Li and W. Wolf. "Hierarchical Scheduling and Allocation of Multirate Systems on Heterogeneous Multiprocessors," in Proc., *European Design and Test Conference*, 1997.

[3] D. Kirk and J. Strosnider, "SMART cache design using the MIPS R3000," in Proc., *RTSS'90*, IEEE, 1990.

[4] J. Torrellas, "Multiprocessor cache memory performance: characterization and optimization," Stanford Univ., CSL-TR-92-545,1992.

[5] A. Burchard, Y. Oh, J. Liebeherr, S. H. Son, "A linear-time online task assignment scheme for multiprocessor systems," in Proceedings, *11th IEEE Workshop Real-Time Operating Systems and Software*, pp. 28-31, May 1994.

[6] D.-T. Peng and K. G. Shin. "Static allocation of periodic tasks with precedence constraints in distributed real-time systems," In Proc., *International Conference on Distributed Computing Systems*, 1989.

[7] C.L. Liu, J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment", *J. ACM*, 20(1), pp. 46-61, 1973.