Hardware/Software Co-Synthesis with Memory Hierarchies

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

Abstract

This paper introduces the first hardware/software co-synthesis algorithm of distributed real-time systems that optimizes *memory hierarchy* along with the rest of the architecture. Our algorithm synthesize a set of real-time tasks with data dependencies onto a heterogeneous multiprocessor architecture that meets the performance constraints with minimized cost. Our algorithm chooses cache sizes and allocates tasks to caches as part of co-synthesis. Experimental results, including examples from the literature and results on an MPEG-2 encoder, show that our algorithm is efficient and compared with existing algorithms, it can reduce the overall cost of the synthesized system.

1 Introduction

This paper describes a new system-level algorithm for *hardware-software co-synthesis* of *multi-rate real-time systems* on *heterogeneous multiprocessors*. Unlike most of the previous work in hardware-software co-synthesis, the algorithm not only synthesizes the hardware and software parts of the applications, but also the memory hierarchy: it takes into account the impact of memory hierarchies on system performance and cost in the co-synthesis process. The algorithm targets *periodic* real-time applications running at multiple rates. The target architecture is a heterogeneous multiprocessor architecture that consists of multiple processing elements (PEs) of various types (i.e., general-purpose processors, domain-specific CPUs such as DSPs, and custom hardware), memory components at different levels of memory hierarchy, and communication links. The algorithm synthesizes the hardware, software and memory hierarchy based on a multiprocessor target architecture to meet the performance constraints with minimal cost.

With embedded CPU cores becoming increasingly common in VLSI systems, and with increasing use of multiple embedded cores on a single chip (*systems on a chip*), system designers need to implement major subsystems using real-time system design techniques such as multiple, prioritized tasks sharing CPUs. The design of these systems (*core-based systems*) is complex and requires sophisticated analysis and optimization. Hardware-software co-synthesis can be used to explore the design space and synthesize the application into hardware and software cores that meet design constraints (performance, cost, power, etc.).

Memory hierarchies, in particular caches, are essential for modern RISC embedded cores to obtain sustained high performance. As the functionality of embedded systems increases, caches and memories represent a significant portion of the cost, size, weight, and power consumption of many embedded systems. Ineffective use of the memory hierarchy requires extra transfers of data and program and can significantly increase both execution time and power consumption.

Memory hierarchy must be taken into consideration in system-level design to minimize the overall system cost. For example, to improve the performance of a system, the designer may use a faster and usually more expensive CPU, or add a piece of custom hardware, or use a bigger cache. It is important for the designer to evaluate the tradeoffs among these different design options in order to find the optimized design. Although many processor chips already include caches, they still provide several choices of cache sizes for the same CPU type. In *core-based design* for *systems-on-a-chip*, the designer has the option of adjusting the cache sizes of the CPU cores. However, most previous research in co-synthesis has ignored the cache's impact and only concentrated on the synthesis of PEs for software (processors) and hardware (ASICs). So far, there is no systematic approach for the design of memory hierarchies in co-synthesis. In our previous work [8], we designed a task-level cache performance model and concentrated on analysis and scheduling with memory hierarchy but not co-synthesis.

To handle memory hierarchies in a multi-tasking environment, we need a **high-level** model that can efficiently model the application performance in presence of memory hierarchy. In this paper, we first present a **task-level model** that efficiently bounds the cache performance of tasks running in a *multi-tasking* environment (see Sec.3). We incorporate this model into hardware-software co-synthesis and propose a new co-synthesis algorithm that optimizes the use of memory hierarchy and synthesizes cache memory together with hardware and software to optimize the total system cost (see Sec.4). Sec.5 discusses the experimental results of our algorithm.

2 Previous Work

Related work includes studies from hardware-software partitioning, hardware-software co-synthesis, performance analysis with caches, and real-time computing.

Hardware-software partitioning [3, 4, 14, 16] has been a major topic in the area of **hardware-software co-design**. Most of the partitioning algorithms implement the system based on a template of a CPU (software) and an ASIC (hardware). Recent work in co-synthesis has used a more generalized model consisting of heterogeneous multiprocessors with arbitrary communication links. The SOS algorithm developed by Prakash and Parker [12] used an integer linear programming (ILP) approach. Yen and Wolf's work [15, 17] used a faster iterative improvement approach. The co-synthesis algorithms developed by Dave *et al.*[2] can

handle multiple objectives such as cost, performance, power and fault tolerance. However, all of these algorithms ignore memory hierarchy.

Recent research, such as the path-based analysis algorithm of Li *et al*.[10] has developed cache 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 (or may not) change the state of the cache at a point in a way that compromises the performance of the originally-executing model. For preemptive real-time systems, such interactions are critical to evaluate during system-level architecture design.

Lee,*et.al*.[7] proposed a technique to analyze cache-related preemption delays of tasks that cause unpredictable variation in task execution time for preemptive scheduling. Kirk and Strosnider [6] developed a SMART (strategic memory allocation for real-time) cache design that partitions the cache to provide predictable cache performance. Danckaert,*et al*.[1] studied memory optimization aiming to reduce the dominant cost of memory in hardware-software co-design of multi-media and DSP applications. Their algorithm concentrated on reducing data storage and did not consider multi-level memory hierarchy. All these approaches [1, 6, 10] relies on program-level analysis, and are too expensive to be used in design space exploration of multiple tasks.

Research in the area of **real-time scheduling** provides an important foundation to our co-synthesis algorithm which targets multi-rate real-time tasks. 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)* [11]. For distributed real-time systems, Ramamritham [13] used an task-graph unrolling approach and developed a heuristic allocation and scheduling algorithm that considered data dependencies, communication, and fault-tolerance requirements; Li and Wolf [9] developed an efficient hierarchical algorithm to schedule and allocate multi-rate tasks with precedence constraints.

3 Task-Level Memory Hierarchy Model

Accurate estimation of memory hierarchy (cache) behaviors requires *program-level* or *trace-level* analysis, which are too expensive to be used in the design exploration of *multiple tasks* on a multiprocessor architecture. A *high-level model* of memory hierarchy performance is critical for integrating memory hierarchy into co-synthesis of multiple tasks. The model should be able to:

1. efficiently model the *multi-tasking* environment, which may be further complicated by preemptions;
2. efficiently model cache behavior (hits/misses) when *cache size* changes.

In our earlier work [8], we proposed the first **task-level model** of memory hierarchy performance for system-level synthesis, and allocation/scheduling algorithms with memory hierarchies. This model treats each task as an entity, *partitions* the caches, and reserves some partitions exclusively for certain tasks to guarantee predictable performance of these tasks. While it provides a fast means to bound the cache performance of tasks running in a **multi-tasking** environment, the cache partitioning/reservation approach results in inefficient utilization of the caches. Furthermore,

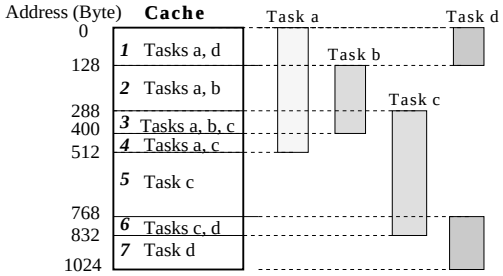

Figure 1: Mapping of tasks onto a 1KB cache.

the model is not flexible in terms of the memory allocation of tasks: for tasks with cache partitions on the same cache, the compiler has to make sure that they do not map to overlapped cache locations.

We have developed a task-level cache performance model that handles *arbitrary mapping* of tasks to caches. Fig.1 shows how tasks can map to a cache. For simplicity, we make the following assumptions about the tasks and the caches:

Assumption 1: Only one-level cache is modeled and tasks are well-contained in the level-1 cache (each task's program size and data size are no bigger than the instruction and data cache size, respectively). This may not be a reasonable assumption in a general-purpose system, but it is plausible for many embedded systems. The kernels of time-critical operations are frequently small enough to fit into a modest-sized cache. Even when a task is too large to be contained in a level-1 cache, it can be specified at a finer granularity to satisfy the assumption.

Assumption 2: The caches are direct-mapped and the cache sizes are powers of two.

Assumption 3: A task's program is allocated a continuous region of the memory and is, therefore, mapped into a continuous region of the cache. A task's data can be scattered in several regions of the memory.

Due to the first assumption, when a task executes on a processor, if not preempted by other tasks, the only cache misses are *compulsory misses* [20]. As opposed to capacity and conflict misses, the number of compulsory misses of a task does not change with cache size.

We now analyze the cache performance of multiple tasks for a fixed cache size. Note that only compulsory misses can happen because of *Assumption 1*. The cache performance of a task depends on the history of task execution on the processor: if the task is executed on the processor for the first time, it is initially loaded into the cache (cold start), with compulsory misses; if the task has been executed before and has not been overwritten by other tasks, then there are no cache misses; if it has been partly overwritten by other tasks, then there are compulsory misses associated with the cache regions that were overwritten. It is important to monitor the change of the cache status to tightly bound the cache performance of tasks.

As shown in Fig.1, when tasks are mapped to a cache, there can be overlap between tasks. These overlaps determine all the possibilities of task overwriting. We divide the cache into several regions according to distinct task boundaries. Suppose there are n tasks mapped to the cache, the number of tasks boundaries is bounded by $O(2n)$, which means the cache is divided into at most $O(2n+1)$ regions.

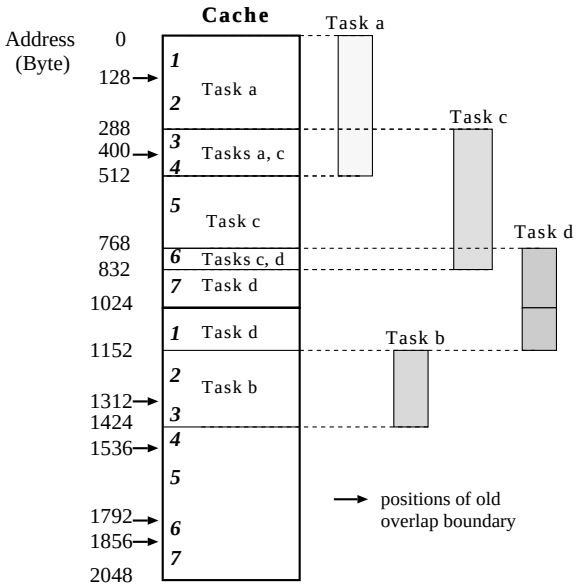

Figure 2: Mapping of tasks onto a 2KB cache.

In the example of Fig.1, the cache is divided into seven regions by four tasks, with each task spanning several regions of the cache.

We define the *state* of a cache region as the task currently loaded in that region, and the *cache state* as a tuple of the states of all the regions. In the example of Fig.1, after executing a, b, d, a, the cache state is $\{a, a, a, a, 0, 0, d\}$, where 0 indicates the region has never been accessed. During a multi-tasking execution, we can look up the current cache state to determine a task's number of misses and, therefore, its execution time. Let $WCET\_base$ be the worst-case execution time of a task assuming no cache misses, and $\#miss$ the number of misses. The $WCET$ considering cache misses is shown in Eq.(1). The number of cache misses is shown in Eq.(2), where $\#miss\_comp(i)$ is the number of the task's (say task x) compulsory misses associated with region i.

$$WCET_x = WCET\_base_x + \#miss_x \cdot miss\_penalty \quad (1)$$

$$\#miss_x = \sum_{state(i) \neq task\_x} \#miss\_comp_x(i) \quad (2)$$

In summary, for a fixed cache, the cache performance model

1. map tasks to the cache and divide the cache into regions according to task overlaps;
2. for each task and each of its related regions, obtain the number of compulsory misses of that task associated with that region;
3. in the multiple task execution, monitor the cache state to compute the number of cache misses and $WCET$s for the tasks in their execution context, using Eq.(1) and Eq.(2).

When we change the cache size, the overlap between tasks may change. In Fig.2, we double the cache size of Fig.1 and tasks a-d map differently onto the new cache and generate different divisions of cache. However, an important observation is that doubling cache size does not incur more divisions on the cache: the number of regions that a task spans can only stay the same, or decrease. Since the number of compulsory misses of a task on a particular region does not change with cache size [20], we do not need to re-compute

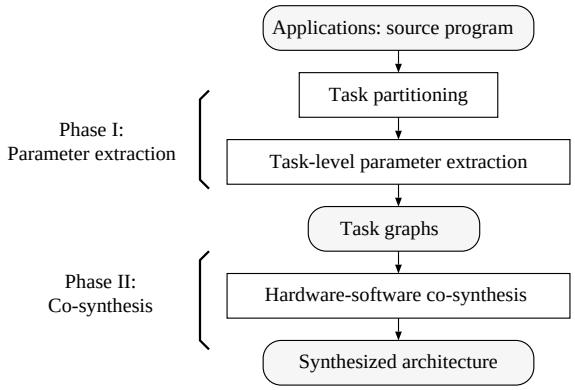

Figure 3: System flow.

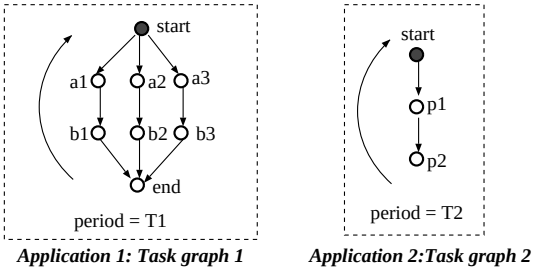

Figure 4: Task graphs.

the compulsory miss numbers for the tasks. For example, in Fig.1, task a spans four regions 1-4; when cache size is doubled, as shown in Fig.2, since task b no longs overlaps a, task a now spans two regions (1,2) and (3,4). The compulsory misses of a for these two regions can be easily computed by adding up the compulsory misses of their correspondent sub-regions 1-4. Based on this observation, to analyze all possible cache sizes, we can start from the smallest cache that satisfies *Assumption 1*, the analysis of any other cache size can be inductively done from the cache half of its size.

The above discussion is based on the assumption that each task is mapped to one continuous region of the cache. While this is true for task program, it is not valid for task data which may occupy several disjoint regions (*Assumption 3*). The only difference is that the multiple data regions for one task will result in more divisions on the data cache, but a similar analysis still applies.

4 Hardware/Software Co-synthesis with Memory Optimization

Based on the task-level model for cache performance, we have built a framework for hardware/software co-synthesis with cache. Fig.3 shows the flow graph of our framework. It has two main phases: the first phase, *parameter extraction*, prepares for co-synthesis—it extracts task graphs and task-level parameters from the original application specifications (source programs); these parameters are then used by the second phase—design space exploration (co-synthesis) to synthesize the architecture. Sec.4.2 and Sec.4.3 will describe these two steps respectively.

4.1 Problem Specification

The problem specification of our co-synthesis algorithm includes two components: a set of real-time applications and a technology database. The real-time applications are periodic, running at multiple rates. Each application is represented by an *acyclic task graph*, as shown in Fig.4, where

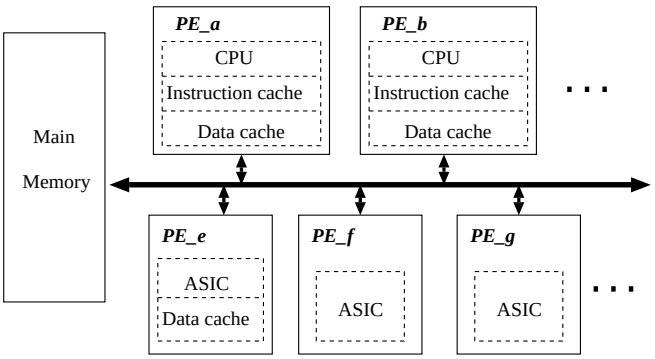

Figure 5: An example architecture.

nodes represent tasks, and directed edges represent data dependencies between tasks. Different tasks may share program or data in the memory. The data dependencies can be either read-after-write (RAW), write-after-read (WAR) or write-after-write (WAW).

Tasks in one application run at the same rate. We assume that the deadline of the tasks is equal to their period. Each task can have several implementation options differing in area cost and execution time. The *technology database* provides the tasks a number of choices for the types of processors, ASICs, and caches, each associated with a certain cost.

We use a heterogeneous *shared-memory multiprocessor* as the template architecture (see Fig.5). The architecture has a number of PEs of various types. Each processor has its private instruction cache and data cache. An ASIC may have a private data cache. Lower-level caches and memory are shared. PEs and memory components are linked by a shared bus.

The **goal** of the algorithm is to:

1. choose the number and types of components in the target architecture from the technology database, such that the applications can be scheduled to meet their performance constraints (deadlines) and the total cost of the result system is minimized.
2. return the allocation and scheduling of the tasks on the result architecture.

4.2 Parameter Extraction

For each task, from its program-level description, we **extract task-level parameters** that are essential for evaluating the task's execution and caching behaviors. These parameters include: worst-case execution time when there are no cache misses ($WCET\_base$), the task's instruction and data address ranges in memory ($program\_region$, and $data\_region1$, $data\_region2$, ...). Then separately for data and instruction caches, we compute the smallest cache size that satisfies *Assumption 1*, and assuming all tasks are allocated to this one cache, we divide the cache into regions according to task overlaps in cache and compute the tasks' compulsory misses on each of its relevant regions (as described in Sec.3). In the co-synthesis process, only a subset (say T) of all the tasks will be allocated to a given PE (say P), this will result in fewer regions in P's cache. The compulsory misses for each task in T associated with each cache region can be obtained by removing boundaries related to tasks that are not allocated to P.

In our framework, task execution times and memory addresses are obtained by behavior simulation tools, and the number of cache compulsory misses are easily obtained with a cache simulator. These parameters can be also obtained using performance analysis tools such as *Cinderella* [10].

Cache coherency. In a shared-memory multiprocessor architecture, caching of shared data introduces the *cache coherency* problem. In our algorithm, we use the *write invalidate protocol*. A write on one PE will invalidate all other copies of the same data on other PEs to ensure this PE has exclusive access to the data. After a task finishes its execution, its data is written back to the main memory such that the updated data can be used by other tasks. Note that there is no need to write to the main memory during the execution of a task (say a), because any other tasks that are data-dependent on a do not start running until a is finished.

4.3 Hardware/Software Co-synthesis

Based on the task-level cache model described in Sec.3, we have designed an *iterative improvement* algorithm that uses the task-level parameters as inputs and outputs a design that meets the performance constraints with minimal cost.

The total cost C of the system is evaluated as the sum of the component costs ($C(...)$):

$$\begin{aligned} C &= \sum_{i \in CPUs} (C(CPU_i) + C(I\_cache_i) + C(D\_cache_i)) \\ &\quad + \sum_{j \in ASICs} (C(ASIC_j) + C(D\_cache_j)) \\ &\quad + \sum_{k \in links} C(communication\_link_k) \end{aligned} \tag{3}$$

Performance evaluation. We have used two different methods at different points in the design process to evaluate the performance of a design. One method is to compute the workload (Eq.(4)) on each PE to quickly check its feasibility. The workload on a PE is the sum of the workload of all the tasks allocated to this PE:

$$Workload(PE) = \sum_{i \in Tasks} WCET(task_i, PE)/Period(task_i)$$

where $Tasks$ is the set of tasks allocated to PE. If any PE in the system has a workload of higher than 100%, then the design is not feasible. Workload analysis is used in the intermediate steps of the design space search to quickly weed out infeasible designs. However, due to data dependencies and bus contention, a PE can rarely achieve a 100% utilization. A design is validated only when a schedule can be constructed without violating task deadlines.

Synthesis refers to the exploration of the design space. It is integrated with the cost/performance evaluation and scheduling algorithm to find the optimized design. Our synthesis algorithm consists of the following steps:

1. Find an initial solution.
2. Iterative PE and cache cost reduction.
3. Allocate and schedule tasks and bus transfers for the final design.

In step 1, the *initial solution* is constructed by assigning each task in the task graphs the fastest PE that is available for the task. The PE with the least $WCET\_base$ is chosen. If the PE chosen is a CPU, instruction and data caches of the task's program and data sizes are added; if it is an ASIC, a data cache of the size of the task's data size is added. The performance of the initial solution is evaluated. If it cannot meet the real-time deadlines, then for the given task graphs, there exists no feasible design given the current technology database and the algorithm returns without a solution.

The PE and cache cost reduction step is the core step of the algorithm and Sec.4.3.2 describes the details of this step.

4.3.1 Task Allocation and Scheduling

Task allocation and scheduling are important aspects of the co-synthesis algorithm. The scheduling routine is used not only to generate the allocation and schedule of the final design, but also to evaluate the performance of intermediate solutions, and to help generate new solutions. A schedule that utilizes the PEs well is critical to lower the system cost. A fast scheduler is important to shorten the performance evaluation time of a design and, therefore, allows the design space to be more thoroughly searched.

Scheduling of multiple real-time tasks onto heterogeneous multiprocessors is a difficult problem in itself. The addition of caches make it even more complicated. We built our scheduling algorithm based on the *hierarchical scheduling* algorithm (referred as HS-algorithm) developed by Li and Wolf [9]. This HS-algorithm uses the hierarchical structure of the system's task graphs to hierarchically allocate and schedule tasks on the multiprocessors and memory transfers on the bus, to meet the real-time constraints. The HS-algorithm targets the same task model and architecture model as used by our framework, but did not originally consider memory hierarchies. We added caches to the PEs and integrate our memory hierarchy model to HS-algorithm. In the HS-algorithm, the task's execution time on a certain PE is assumed to be fixed. This is no longer valid when caches are added—the execution time of a task $task_i$ on a PE PE_j not only depends on the speed of the PE, but also the speed of the cache and the current cache size and cache state. Therefore, instead of using a fixed $WCET(task_i, PE_j)$, we dynamically compute it according to the the current cache state (Sec.3). This change is reflected in the calculation of *dynamic urgency*, a measure used by the HS-algorithm to decide the next task to schedule. In the following equation, $WCET(task_i, PE_j$ should be computed according to the current cache state. *Dynamic urgency* encourages a task to re-use the cache state to reduce cache misses.

$$dynamic\_urgency(task_i, PE_j) =$$
$$static\_urgency(task_i) - earliest\_available\_time(PE_j)$$
$$+(average\_WCET\_base(task_i) - WCET(task_i, PE_j))$$

Other parts of the equation, as well as other parts of the scheduling algorithm remain the same and are not discussed in this paper.

4.3.2 PE and Cache Cost Reduction

PE and cache cost reduction is the most critical step in the co-synthesis algorithm. We used an *iterative improvement* strategy to search for the optimized design by cutting PE and cache cost interactively.

A single iteration of cost reduction is shown in Fig.6. This step tries to eliminate lightly loaded PEs by moving the tasks on those PEs to other PEs. The PEs are ordered by their workload (line 3). Starting from the most lightly loaded PE, we identify the tasks on it that can be executed on other PEs (line 6); these tasks are then moved to the other PEs that provide the best performance for the tasks (line 7); the cache sizes of the other PEs increase to accommodate the tasks that are newly moved there (line 8). The PE is removed if it becomes empty (line 10-11). When there are tasks on a PE that cannot be moved to other PEs, the algorithm tries to implement the remaining tasks with a cheaper PE (line 12-13). If such a PE cannot be found, the current PE is kept in the design, but an attempt is made to cut its instruction and data cache sizes (line 14-16).

```
1.PE_&_cache_cost_reduction(design) {
2.  foreach PE_i in design, calculate workload;
3.  sort PEs by increasing workload;
4.  foreach PE_i in sorted list {
5.    foreach task_j allocated to PE_i {
6.      other_PEs =other PEs in design with enough
                 workload left to execute task_j;
7.      move task_j to fastest PE_x in other_PEs;
8.      increase PE_x's I-cache/D-cache size by
             task_j's program/data size;
9.    }
10.   if PE_i is empty
11.     remove PE_i and its caches;
12.   else if exists a cheaper PE_x to
            implement all tasks left on PE_i
13.     replace PE_i with PE_x;
14.   else
15.     keep PE_i;
16.     if feasible cut PE_i's cache size by half;
17.  }
18. return the new design;
19. }
```

Figure 6: One iteration of PE/cache cost reduction.

```
iterative_pe_cache_cost_reduction(initial_design){
 last_cost = cost(initial_design);
 last_design = initial_design;
 do {
  thisdesign=PE_&_cache_cost_reduction(last_design);
  last_design=allocate_and_schedule(thisdesign);
 } while(!stop_condition);
 /*stop condition: no cost improvement in 3
   consecutive iterations*/
}
```

Figure 7: The iterative PE/cache cost reduction.

In the single-iteration procedure, when we move tasks from one PE to another, the performance constraint may be violated. We use the quick workload bound method (Eq.4) to check the utilization of PEs. In summary, a single iteration of cost reduction is achieved by:

- elimination of PEs that become empty after moving all their allocated tasks to to other PEs;
- replacing PEs with cheaper ones; and
- reducing cache cost.

The iterative algorithm is shown in Fig.7. Starting from the initial design, the algorithm performs the PE/cache cost reduction step-by-step, until there is no improvement in three consecutive iterations. For each new design returned by a single iteration of PE and cache cost reduction, we call the allocation and scheduling procedure to:

- check the validity of the design;
- if it is valid, we generate a new allocation and schedule that is customized to the current system. This is important because the single iteration of cost reduction moves tasks between PEs, eliminates and replaces some PEs. The resultant design may not have a balanced allocation of tasks on the PEs included in the design. We re-allocate and re-schedule the task graphs to achieve a better utilization of the PEs in the current architecture. The newly re-allocated design is used as the starting point of the next cost reduction iteration.

Fig.8 gives an illustrative example of the synthesis process. Starting from the task graph in (a), an initial solution of three PEs and two caches are generated (Fig.8(b)). The algorithm then iteratively reduces the PE and cache cost to obtain a new solution with one less PE and smaller cache

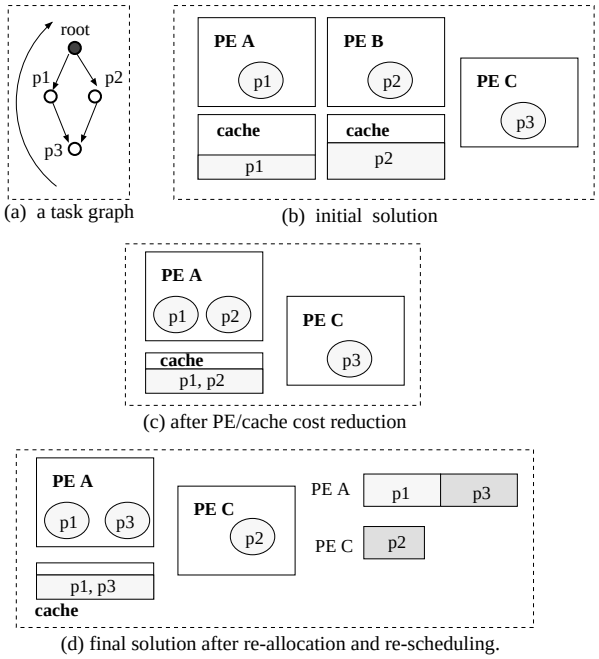

Figure 8: An example of the iterative improvement algorithm.

(Fig.8(c)). Fig.8(d) shows the result of the final design after the tasks are re-allocated and re-scheduled.

4.3.3 Complexity of Our Algorithm

To determine the time complexity of our framework, we recognize that the dominant part is the parameter extraction phase, where program-level estimation or simulation is needed to estimate the $WCET\_base$ of tasks, and the program/data memory locations for tasks on different types of PEs. For one task, the complexity of parameter extraction is either related to the size of its program-level representation if estimation tools are used, or related to the size of its execution trace if simulation tools are used. In our framework, program-level or trace-level analysis is only needed once to extract task-level parameters; the design space search is performed on a task-level abstraction which is much more manageable.

We analyze the worst-case complexity of our co-synthesis algorithm. Suppose there are m task graphs, each with at most k tasks. So the total number of tasks, n, is bounded by mk. Let P be the number of different PE types. Let p be the number of PEs in a design. Because the maximum number of PEs in any design will not exceed the number of tasks, $p = O(n)$. Each full allocation/scheduling step has the complexity of $O(n^2 p) = O(n^3)$. The complexity for a single iteration of PE/cache cost reduction is $O(pnP) = O(n^2 P)$. The total number of iterations is bounded by $O(pP) = O(nP)$ because for each PE, it is either eliminated or can be replaced by a cheaper PE at most P times. Therefore, the worst-case complexity of the co-synthesis algorithm is $O(nP \times (n^2 P + n^3)) = O(n^4 P + n^3 P^2)$.

5 Experimental Results

We conducted two sets of experiments: synthetic task graphs from the literature, and real-life examples including a real MPEG-2 encoder. To compare with existing co-synthesis algorithms, we used examples from the literature [2, 5, 12], as shown in Table 1. We used the same technology database (PE database) as those used in the corresponding references. Table 1(a) shows the results (CPU time and the synthesized system cost) of these examples using several existing algorithms: Prakash and Parker's algorithm [12], Yen and Wolf's algorithm [5], and COSYN by Dave *et.al*. [2]. We ran the same examples on our algorithm, but with three different setups (see Table 1(b)):

Without cache: while running our algorithm, we set the cache part in the technology database to be null, so that the synthesized architecture does not have caches. The results show that even without the benefits of caches, our algorithm can achieve comparable results.

With fixed-size caches associated with each processor: Similar to a typical design practice, we manually picked fixed cache sizes to be used in the target architecture The results show improvements in term of system cost, compared to the no-cache results.

Co-synthesis with cache optimization: This allows the full potential of our algorithm to synthesize software, hardware as well as caches simultaneously. The results show further cost reduction over the fixed-size cache approach.

For the second and the third setups, we needed more input parameters required by our algorithm, such as the memory regions of programs and data for tasks. These parameters were generated because the examples from the literature only have the task-graph representations.

We applied our algorithm to a real MPEG-2 video encoding algorithm. MPEG encoding involves both intensive computation and large amount of data transfers. In real time, image frames arrive at the rate of 30 frames per second. We used the MPEG-2 encoding software from *MPEG Software Simulation Group*. We first extracted the task graph that is composed of 1350 blocks with 12 tasks per block. The graph is huge but the blocks share the same structure, which our algorithm can take advantage of. The technology database consists of SPARC processors, ASICs for DCT, IDCT, various-length encoder and motion estimation, and SRAM to be used as first-level caches. For the SPARC processors, $WCET\_base$, program and data memory regions are obtained using a SPARC behavior simulator $Sparcsim$ [19]. A cache simulator was used to obtain the compulsory miss numbers. We assumed a cache and memory access time ratio of 1:20. We used the retail prices of SPARC processors and SRAMs. $WCET$s and the cost for ASICs were estimated with high-level synthesis. Synthesis results of the MPEG encoder is shown in Table 2. Even in this example, with the huge number of tasks in the task graph, our algorithm was able to find a solution of good quality (average PE utilization 95%) in a short period of time. The short CPU time of our algorithm on such a big design is made possible by the efficient task-level cache performance model, and the hierarchical scheduling methodology which takes advantage of the task graph structures.

An interesting fact of the MPEG experiment is the CPU time spent in different phases of our frame work: in the parameter extraction phase, for all the tasks, generating their execution traces (about 100M instructions in total), computing $WCET\_base$ and program/data memory mapping took about about 4 hours in total; using the task execution traces, it took a little less than 1 hour to compute compulsory misses for all tasks. In contrast, the co-synthesis algorithm itself only took minutes (see Table 2). This shows that the task-level abstraction and the task-level cache model greatly speed up the design space exploration, which would have been impossible with program-level analysis tools that spend hours to evaluate just one single design for the MPEG

Examples,#tasks	Prakash/Parker		Yen/Hou/Wolf		COSYN	
	#PEs / Cost($)	CPU time on Solboume 5/e/900 (sec)	#PEs / Cost($)	CPU time on Sparc 20 (sec)	#PEs / Cost($)	CPU time on Sparc 20 (sec)
Prakash&Parker, 4	1/5	37	N/A	N/A	1/5	0.20
Prakash&Parker, 9	1/5	3691.20	1/5	59.15	1/5	0.40
Prakash&Parker, 9	2/10	7.2hrs	3/10	56.79	2/10	0.54
Yen&Wolf Ex, 6	N/A	N/A	3/1765	10.63	3/1765	0.74
Hou&Wolf Ex1, 20	N/A	N/A	2/170	14.96	2/170	5.10

(a) Experimental results for three existing co-synthesis algorithms.

Examples	Without cache		With fixed cache		Our algorithm	
	#PEs/Cost($)	CPU time(sec)	#PEs /Cost($)	CPU time(sec)	#PEs /Cost($)	CPU time(sec)
Prakash/Parker	1/5	0.30	1/4.2	0.36	1/2.6	0.45
Prakash/Parker	1/5	0.71	1/4	0.73	1/2.4	0.89
Prakash/Parker	2/10	0.75	2/7.4	0.75	1/4.8	1.10
Yen/Wolf Ex	3/1765	1.16	3/1640	2.02	2/1220	2.38
Hou&Wolf Ex1	2/170	6.59	2/190	6.60	2/145	7.12

(b) Experimental results for our co-synthesis algorithm.

Table 1: Experimental results for examples from literature.

Example/Results	Without cache		Fixed caches		Co-synthesis w/ caches	
	#PEs/Cost($)	CPU time(sec)	#PEs/Cost($)	CPU time(sec)	#PEs/Cost($)	CPU time(sec)
MPEG video encoder	6 / 840	121	5 / 680	157	4 / 520	203

Table 2: Co-synthesis results for the MPEG-2 video encoder.

encoder.

6 Conclusions

In this paper, we described a task-level model for bounding cache performance of tasks in a multi-rate, multi-tasking environment. This model is used by our algorithm for hardware-software co-synthesis with cache memory optimization. The algorithm is the first co-synthesis algorithm that considers the impact of memory hierarchy on the system performance and cost. Our algorithm synthesizes complex multi-rate real-time applications onto a heterogeneous multiprocessor architecture to meet real-time deadlines at minimal cost. The co-synthesis algorithm works at the task level, does not require a detailed program analysis, and is very computationally efficient.

Future work may include: developing co-synthesis algorithms with a more generalized memory hierarchy model. We plan to model set associative caches, and extend the one-level cache model to multiple level caches. Secondly, our memory-hierarchy model optimizes context switching at task-level, which not only helps reduce computation time, but also power consumption. We plan to develop a quantitative model for power consumption at system level and use power as another objective of co-synthesis.

Acknowledgments

This work was supported by the NSF under grant MIP-9424410. The authors would like to thank Zhao Wu of Princeton University for his constructive comments on drafts of the paper.

References

[1] K. Danckaert, F. Catthoor, and H.De Man, "System level memory optimization for hardware-software co-design", in *Proceedings, International Workshop on Hardware/Software Co-Design*, 1997.

[2] B. Dave, G. Lakshminarayana, and N. Jha, "COSYN: Hardware-software co-synthesis of embedded systems", in *Proceedings, 34th Design Automation Conference,* 1997.

[3] R.Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers", *IEEE Design and Test of Computers*, vol.10, no.4, pp.64-75, Dec. 1993.

[4] R. Gupta and G. De Micheli, "Hardware-software Cosynthesis for Digital Systems", *IEEE Design and Test of Computers*, vol.10, no.3, pp.29-41, Sept. 1993.

[5] J. Hou and W. Wolf, "Process partitioning for distributed embedded systems," in *Proceedings, International Workshop on Hardware/Software Co-Design*, 1996.

[6] D. Kirk and J. Strosnider, "SMART(strategic memory allocation for real-time) cache design using the MIPS R3000," in *Proceedings, 11th Real-time Systems Symposium*, 1990.

[7] C.-G. Lee *et al.*, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," in *Proceedings, 17th Real-Time Systems Symposium*, 1996.

[8] Y. Li and W. Wolf, "A Task-Level Hierarchical Memory Model for System Synthesis of Multiprocessors," in *Proceedings, 34th Design Automation Conference*, 1997.

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

[10] Y.-T. Li, S. Malik and A. Wolfe, "Performance Estimation of Embedded Software with Instruction Cache Modeling," in *Proceedings, ICCAD '95*, 1995.

[11] C.L. Liu and J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," *J. ACM*, vol.20, no.1, pp.46-61, 1973.

[12] S. Prakash and A. Parker, "SOS: synthesis of application-specific heterogeneous multiprocessor systems," *Journal of Parallel and Distributed Computing*, vol.16, pp.338-351, 1992.

[13] K. Ramamritham, "Allocation and scheduling of complex periodic tasks," In Proceedings, *International Conference on Distributed Computing Systems*, 1990.

[14] F. Vahid, J. Gong, and D. Gajski, "A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning," in *Proceedings, EuroDAC'94*, 1994.

[15] W. Wolf, "An Architectural Co-Synthesis Algorithm for Distributed, Embedded Computing Systems," *IEEE Transactions on VLSI Systems*, vol.5, no.2, pp.218-229, Jun. 1997.

[16] W. Wolf. "Hardware-software co-design of embedded systems," *Proceedings of the IEEE*, July 1994.

[17] T.-Y. Yen and W.Wolf, "Communication Synthesis for Distributed Systems," In *Proceedings, International Conference on Computer Aided Design*, 1995.

[18] Motion Pictures Experts Group ISO-IEC/JTC1/SC29/WG11. "Information technology-Coding of moving pictures and associated audio for digital media at up to about 1.5 Mbit/s," 1992.

[19] W. Ye *et al.*, "Fast Timing Analysis for Hardware-Software Co-Synthesis,", *ICCD93*, 1993.

[20] J. Hennessy and D. Patterson. "Computer architecture, a quantitative approach", second edition, Morgan Kaufmann, 1995.