Dynamic Partitioning of Processing and Memory Resources in Embedded MPSoC Architectures

Liping Xue, Ozcan ozturk, Feihui Li, Mahmut Kandemir
Computer Science and Engineering Department
Pennsylvania State University
University Park, PA 16802, USA
{lxue,ozturk,feli,kandemir}@cse.psu.edu

I. Kolcu
Computer Department
UMIST
Manchester M60 1QD, UK
ikolcu@umist.ac.uk

Abstract

Current trends indicate that multiprocessor-system-on-chip (MPSoC) architectures are being increasingly used in building complex embedded systems. While circuit/architectural support for MPSoC based systems are making significant strides, programming these devices and providing suitable software support (e.g., compiler and operating systems) seem to be a tougher problem. This is because either programmers or compilers will have to make code explicitly parallel to run on these systems. An additional difficulty occurs when multiple applications use an MPSoC at the same time, because MPSoC resources should be partitioned across these applications carefully. This paper explores a proactive resource partitioning scheme for parallel applications simultaneously exercising the same MPSoC system. The proposed approach has two major components. The first component includes an offline preprocessing of applications which gives us an estimated profile for each application. Each application to be executed on our MPSoC is profiled and annotated with the profile information. The second component of our approach is an online resource partitioning, which partitions both the processing cores (i.e., computation resources) and on-chip memory space (i.e., storage resource) among simultaneously-executing applications. Our experimental evaluation with this partitioner shows that it generates much better results than conventional operating system based resource management. The results also reveal that both memory partitioning and processor partitioning are very important for obtaining the best results.

1. Introduction

Researchers from both academia and industry agree that the future system-on-chip (SoC) architectures will be mostly built using multiple cores. These architectures, called multiprocessor-system-on-chip (MPSoC), lend themselves well to simpler design, faster validation, cleaner functional partitioning, and higher theoretical peak performance [10, 2, 15, 8]. Also, single-chip designs typically draw less power, an area of critical importance in devices employed by power-hungry multimedia environments such as smart phones and feature phones. While, from a hardware perspective, it is possible to put together multiple processing cores and connect them to each other using an on-chip memory, programming such chips is an entirely different matter. Prior studies targeting MPSoC based system mostly considered single application based scenarios, where a single application is executed in the MPSoC at a time.

While this execution scenario is certainly useful in certain embedded domains, there also exist many scenarios where multiple applications can exercise the same chip at the same time. A crucial design question in this case is how the MPSoC resources should be partitioned among multiple applications that execute in parallel. In particular, suitable partitioning of processing cores (i.e., computation resources) and memory space (i.e., storage resource) is critical as it dictates both performance and power consumption. Clearly, one simple option is to let an operating system (OS) to do this partitioning. An advantage of this approach is that it is entirely transparent to applications and programmers. A potential drawback is that the OS policies are generally application oblivious and reactive, i.e., they are fixed and based on the past behavior of the applications currently executing in the system.

Our work explores a proactive resource partitioning scheme for parallel applications simultaneously exercising the same MPSoC system. The proposed approach
has two major components. The first component includes an offline preprocessing of applications, which gives us an estimated profile for each application. Simply, an estimated profile for a given application is a concise description of its predicted processor (computation) and memory space requirements. Each application to be executed on our MPSoC is profiled and annotated using this profile information. The second component of our approach is an online resource partitioner, which partitions both the processing cores and on-chip memory space among simultaneously-executing applications.

To quantify the potential benefits of our approach, we implemented it and compared it to several alternate schemes. Our experimental findings so far are very encouraging and indicate that the proposed resource partitioning scheme is very effective in practice. More specifically, our results show that the proposed resource partitioner generates much better results than conventional operating system based resource management. The results also reveal that both memory partitioning and processor partitioning are very important for obtaining the best results.

The next section discusses related work. Section 3 explains our execution model and gives a high level view of the two components of our approach: offline profiler and online resource partitioner. Section 4 discusses the technical details of our approach and gives an example to demonstrate how it works. Section 5 provides experimental data. Finally, Section 6 gives a summary and concludes the paper.

2. Related Work

Chip multiprocessing is most promising in highly competitive and high volume markets, for example, embedded communication, multimedia, and networking domains. This imposes strict requirements on performance, power, reliability, and costs. There exist various prior efforts [5, 12, 13, 18] on chip multiprocessing, and they improve the behavior of an MPSoC-like architecture from different aspects, for example, memory performance, communication, reliability, etc. As these multiprocessor systems post a new challenge for compiler researchers, they also provide new opportunities as compared to traditional architectures. Optimizing resource partitioning through cooperation between offline profiling and runtime partitioning is one such opportunity which is explored in this work.

Vanmeervect et al [16] present a hardware/software partitioning of an embedded system. There exist a group of studies that focus on partitioning memory space among processors/threads [4, 11, 7]. In comparison to these previous efforts, our approach partitions both computation and storage resources and, as against the mentioned previous work, we optimize resource partitioning across multiple applications, not among the threads of a given application.

3. Execution Model

An abstract view of the MPSoC architecture considered in this work is depicted in Figure 1(a). This architecture contains a small number of processor cores (usually between 4 and 16) and a shared on-chip memory space among other on-chip components. The shared memory space may be banked for energy reasons. When an application is to be executed on this architecture, we need to assign a number of processors to it and allocate a certain fraction of the on-chip memory space. This can be done under OS control or can be explicitly managed by a customized resource partitioner. This second option is the one explored in this work. Clearly, an important issue in this context is to partition the processor and on-chip memory resources in such a fashion that no application starves and applications complete their executions as quickly as possible. In this work, each application is assumed to manage the on-chip memory space allocated to it as a software-managed memory (though our approach can be used with a hardware-mapped cache-based memory system without any major modification).

Figure 1(b) shows a simple resource partitioning scheme that divides all on-chip resources evenly across two applications. Our approach, in contrast, can come up with a partitioning such as the one shown in Figure 1(c). Note that, in this case, one of the two applications is given only 3 processors (out of a total of 8 processors) but it is allocated a larger portion of the on-chip memory space. When a new application is introduced into the system, our approach re-partitions the processor and memory resources adaptively, as illustrated in Figure 1(d) for a possible scenario. It needs to be emphasized that such a nonuniform
partitioning makes sense, especially when one considers the fact that, while some applications are compute-intensive and thus need more computation resources, there exist some other applications which are memory-intensive, i.e., they need a large on-chip memory space for the best results.

4. Components of Our Approach

Our approach has two major components, as shown in Figure 2. The first component is offline and generates an estimated profile, which captures the behavior of each application under the different processor counts and different on-chip memory capacities. More specifically, each application is annotated with two types of profile statistics. The first statistic gives the ideal number of processors and the second one captures the on-chip memory space requirements for each epoch of the application (we will define the concept of epoch shortly). The second component of our approach is a runtime resource partitioner. This partitioner takes application annotations as input and determines a suitable partitioning of processors and on-chip memory space. The rest of this paper discusses these two components in more details.

4.1. Estimator

In this subsection, we explain how we characterize an application prior to its run. Recall from our discussion above that, our approach operates at an epoch granularity. Specifically, we divide the execution of an application into epochs of the same size and capture the ideal number of processors to use at each epoch. This information can be collected through static analysis or profiling (the latter is used in this paper). What we mean by the "ideal number of processors" is the processor count beyond which the execution time of the program does not improve in any significant way. As an example, Figure 3 shows the speedup curve for a given epoch (actually the third epoch) of one of our benchmarks (named VB 2.0). The x-axis of this figure corresponds to the number of processors used and the y-axis is the amount of on-chip memory space allocated to the application in that epoch. We see that, beyond 4 processors, there is not a significant increase in execution cycles, and consequently, 4 is the ideal number of processors for this epoch of this application. Similarly, we see from the same figure that, beyond 12KB on-chip memory, there is not a significant performance (speedup) gain that can be obtained by increasing memory size allocated. Based on these observations, one can draw for this application (and others) a graph such as the one shown in Figure 4, which gives us the ideal number of processors and the amount of on-chip memory space for each epoch of a given application (VB 2.0 in this case). Note that, in this graph, the y-axis is normalized with respect to the maximum number of processors (8) and the maximum amount of available on-chip memory (32KB). We can see from these results that this application does not need all available processor and memory resources in any of its epochs. Therefore, allocating a larger number of processors and a larger on-chip memory space (than necessary) to it will not improve its performance significantly. The important point here is that the computation/storage resources that are not needed by an application can be given to the others. This is the flexibility exploited in this paper.

At this point, we want to explain why an epoch does not normally need all the processors and the largest amount of available on-chip memory for the best performance. From the processor side, it is well known that increasing the number of processors tends to increase interprocessor synchronization and, beyond a point, the synchronization overheads simply start to dominate execution time. As for
amount of memory required, it is also known that an application works best when the amount of on-chip memory allocated to it can hold its entire working set. Increasing the memory space further, while may continue to bring some marginal benefits, does not lead to significant speedups. Overall, when we look at the results shown in Figure 4 carefully, we see that each epoch has different on-chip memory space and processor requirements, and the amount of processor and memory resources allocated to an application should be varied as the application moves from one epoch to another during its execution. While these results in Figures 3 and 4 are for a particular application (VB 2.0), all our applications exhibit similar trends. Following the profiling phase, each application is annotated with the information similar to those captured in Figure 4 and this information is fed to the second component of our approach, which is explained next.

4.2. Resource Partitioner

Our runtime partitioner is invoked whenever an application starts its execution, ceases to execute, or moves from one epoch to another during its execution. The main job of the resource partitioner is to divide the number of processors and the on-chip memory space among the simultaneously-executing applications. In doing so, it needs to consider two important issues. First, it needs to be fast in doing this allocation (partitioning) since it executes at runtime and any extra overhead on its part returns as a performance degradation. Second, it needs to do a reasonably good job in partitioning, since a poor partitioning can be detrimental from both performance and energy angles. Our partitioning algorithm, whose pseudo code is given in Figure 5 partitions the on-chip resources among the applications based on their requirements captured by the first component of our approach. Suppose that $\alpha_i$ and $\beta_i$ represent the ideal processor count and the ideal amount of on-chip memory space for the current epoch $i$ of each active application

$$\theta_i \leftarrow \left\{ \frac{\alpha_i}{\sum_j \alpha_j} \times 100 \right\}$$

$$\gamma_i \leftarrow \left\{ \frac{\beta_i}{\sum_j \beta_j} \times 100 \right\}$$

allocate $\theta_i/\%$ of processors for application $i$
allocate $\gamma_i/\%$ of on-chip memory space for application $i$
using an LFU-based selection of victim regions

Figure 5. A sketch showing the functioning of our resource partitioner. This code is executed whenever a new application enters the system, ceases to exist, or moves from one epoch to another during execution.

Note that this approach partitions the resources in a proportional manner based on the requirements of the applications. While the approach is easy to explain, there are several issues that need to be addressed.

The first issue is due to the fact that we need to perform a new partitioning each time an application starts execution, finishes its execution, or changes its epoch. As a result of this new partitioning, a processor or a memory location may need to be taken from that application. For example, suppose that an application was using 3 processors to execute its current epoch. If, as a result of a new application introduced into the system (i.e., starting its execution), the partitioner needs to reduce the processor allocation of the former application to 2, the selection of the processor to take away can be critical in certain cases. For example, if each processor also has a small private on-chip memory, then we need to select the processor (the victim) that exhibits the worse performance when considering all three processors. Since the MPSoC architecture we consider in this work does not have such private on-chip memory components, this scenario does not arise in our case. However, a similar problem occurs when an application is expected to return some of its allocated memory space to the partitioner. A suitable approach would return the memory locations that it uses least frequently (LFU based) or least recently (LRU based). Our current implementation employs an LFU based strategy and operates as follows. We assume that the on-chip memory space is divided into fine granular regions and the memory allocations/deallocations are carried out at a region granularity. Specifically, during execution, we keep track of the number of accesses to each region (using a set of counters) and when we are to return $k$ regions
<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Brief Explanation</th>
<th>Input Size (KB)</th>
<th>Execution Cycles (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss-Seidel</td>
<td>Gauss-Seidel Computation</td>
<td>5731.4</td>
<td>386.9</td>
</tr>
<tr>
<td>Weather</td>
<td>Weather Prediction</td>
<td>5982.2</td>
<td>901.1</td>
</tr>
<tr>
<td>Edge</td>
<td>Edge Detection Algorithm</td>
<td>5418.5</td>
<td>513.2</td>
</tr>
<tr>
<td>Jacobi</td>
<td>Jacobi Iterative Solver</td>
<td>1545.0</td>
<td>155.7</td>
</tr>
<tr>
<td>VB 2.0</td>
<td>Vertex Blending</td>
<td>7816.4</td>
<td>995.1</td>
</tr>
<tr>
<td>Map 1.1</td>
<td>Cube Mapping</td>
<td>2998.9</td>
<td>372.3</td>
</tr>
<tr>
<td>RB-SOR</td>
<td>Red-Black Successive-Over-Relaxation</td>
<td>4116.5</td>
<td>664.8</td>
</tr>
<tr>
<td>TDer</td>
<td>Terrain Detection</td>
<td>6605.1</td>
<td>756.9</td>
</tr>
</tbody>
</table>

Table 2. Benchmarks used in our experiments.

Table 3. Different workloads.

- **EQUAL**: In this scheme, both the memory space and processors are divided equally among the applications in the workload.
- **MEM-OPT**: In this scheme, the memory space is partitioned carefully among the applications in the workload, but processors are partitioned equally among them.
- **PROC-OPT**: This is the opposite of the previous scheme. We partition the memory space equally among the applications in the workload, whereas the processors are partitioned carefully, based on the dynamic needs of the applications.
- **OPT**: This is the optimization approach proposed in this paper. It partitions both memory space and processors carefully among the applications.

The execution modes MEM-OPT and PROC-OPT are obtained from the OPT scheme by disregarding, respectively, the processor and memory partitioning returned by OPT. The reason that we make experiments with these two modes (MEM-OPT and PROC-OPT) is to check whether both memory space and processor partitioning are really necessary for achieving the best results.

The main metric we use in this paper to evaluate each execution mode is **cumulative execution cycles** (or CEC for short), which captures the amount of time it takes for the last application in the workload being executed to finish its execution.\(^1\)

### 5.2. Results

The graph in Figure 6 gives the execution cycles taken by different workloads under the execution modes explained above. As defined earlier, the term “workload” refers to a set of applications simultaneously running on the system. The contents of the workloads (W1 through W8) used to obtain the results shown in Figure 6 are given in Table 3. Note that each workload in Table 3 has three applications in it. Each bar in Figure 6 is normalized with respect to the last column of Table 2. One can see from the results in Figure 6 that EQUAL does not perform well since it treats all applications in the system the same way, without taking into account

---

\(^1\) While our approach will normally lead to energy savings as well, in this paper our focus is on performance.
their individual processor/memory space requirements. We also see that, while both MEM-OPT and PROC-OPT bring improvements over the BASE execution mode, the savings are not very significant (11.8% with MEM-OPT and 11.1% with PROC-OPT) when averaged over all eight workloads. When we look at the results with the OPT scheme, on the other hand, we see that it reduces CEC by 26.6% on average, that is, it performs much better than both MEM-OPT and PROC-OPT. This result says that it is important to partition *both* memory space and processors carefully, i.e., it is not sufficient to optimize only memory space or processor partitioning.

We also want to mention here briefly why the OS based approach does not perform as good as ours. This is mainly due to the reactive nature of the OS based approach. More specifically, the OS based resource partitioning implements a strategy that is based on the immediate needs of the applications executing in the system. Therefore, while its resource partitioning is accurate for the immediate future, it may lag behind in the long run and optimization opportunities are lost while the OS tries to adapt to the new dynamic requirements put forward by the applications. In contrast, our partitioner is a customized one and makes use of the information collected through profiling, thanks to the particular application domain we are dealing with. We want to emphasize that our work does not suggest that future embedded MPSoC architectures should operate without any OS. Rather, what we want to say is that, depending on the application domain targeted, a specialized resource partitioner can generate better results that could be obtained from a general OS based strategy.

### 6. Summary

Proper programming and runtime support for MPSoCs stands as one of the exciting research topics in embedded computing. While most of the papers that dealt with this problem focused mainly on single application execution scenarios, it is also very important to address the problem when multiple applications are exercising the same MPSoC system at the same time. Focusing on this problem, this paper proposes an approach that partitions processing and memory resources across multiple applications. This approach, which has both static (compile-time) and dynamic (run-time) components and adapts resource partitioning to the dynamically changing needs of the applications. The experiments with eight embedded applications show that not only this adaptive approach generates better performance than an OS based approach, but it also outperforms a simpler scheme that divides all on-chip resources among applications evenly.

### References


