Partitioning Real-Time Applications Over Multicore Reservations

Gordan Konevski
gordan.konevski@stud.hshl.de

Hochschule Hamm-Lippstadt 2022

***Abstract*—As multicore systems continue to develop in an exponential manner, the complexity and the size of applications that would run on such systems make the issue of resource management, especially of resources that are shared, evermore serious and convoluted. As such, this paper tries to delineate the issue and explore the ideas presented in the same-named paper, authored by Buttazzo et al., as a reference point for further understanding of this issue.**

I. Introduction

It is becoming clear that embedded systems have widely adopted the employment of multiple processing units - in increasing regularity - almost as a standard framework, in reaction to the ever-increasing computational demand of intensive applications. There is no reason to assume that this trend will dissipate. However, while these systems offer extra computing power, this trend also opens a new range of possibilities, or essentially a necessity, to improve the reliability of running real-time embedded applications on them, since there are many issues that may arise and have to be considered.

These issues are varied and have to be approached differently and have to consider both the hardware, and the software elements of their respective systems. Most of them, however, could easily be described in a few categories, namely: cost, efficiency, and reliability. Naturally, these categories are gross simplifications, there is a lot more that is involved in the design of multicore embedded systems, but it will be useful to make those simplifications nonetheless. Multicore systems are more efficient than their singlecore counterpart, but are more expensive to manufacture. Their reliability, however, is in some ways improved and in some ways more complicated. Running heavy applications on single core systems, on higher speeds, would cause serious heating issues, which would be much easier to maintain on multicore systems and, by that extension, save on power consumption. But, as multicore systems bring perhaps unintended solutions, they also bring the opposite. [1]

Task partitioning, is regarded as one of the most predominant problems in the process of designing multiprocessor embedded systems. [2] Many applications executed on multicore systems are designed to run in parallel. Most parallel applications contain one or more parallel sections, which are executed in parallel by different threads and, ultimately, their performance is completely determined by the slowest thread. [2] Some parallel applications running on a multicore architecture would have to take in consideration the different speeds among the threads belonging to the same system.

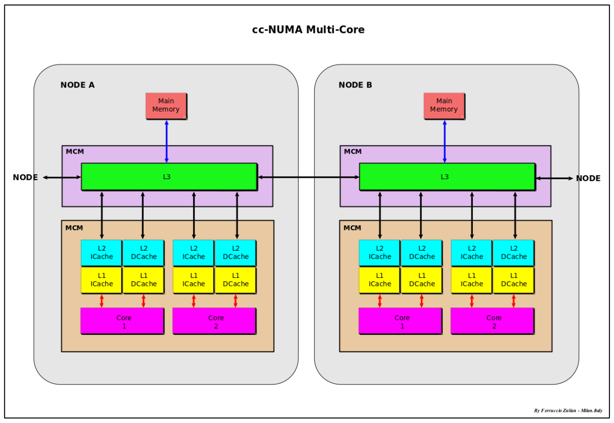

Fig. 1. Multicore System Example

But the most critical scheduling problem is deciding how applications would be prioritized if those applications happen to share the same resources, not just in terms of hardware, but also data.

II. Optimization as a Necessity

Smartphones are a concrete example of this. Most smartphones run on Unix, which has its own methods of handling different threads. Most features, including some operating system functions like task scheduling, are concealed even from application programmers. Resource management, which decides and schedules which processes and applications get CPU priority, which applications affect battery life, and internet bandwidth, is left to the OS of the systems and is taken away from the third-party applications, fringe cases notwithstanding. The majority of conventional operating systems only provide rudimentary mechanisms for allocating resources among programs, mostly through the use of priorities. Prioritization, on the other hand, prevents isolation in multiapplication systems since it necessitates a system-wide understanding of all competing applications and services for the same resources. Resource reservations offer a more user-friendly interface for assigning resources like CPU to several apps.

A resource manager can use this strategy to assign a fraction of the platform capacity to each program, which then operates

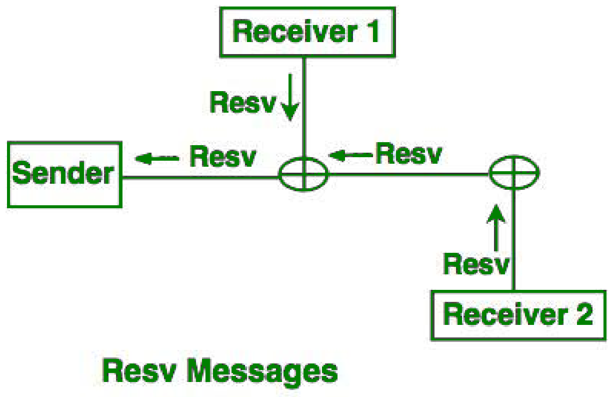

Fig. 2. Resource reservation example

as if it were running alone on a lower-performing virtual platform, regardless of the behavior of the other applications. In this way, each application's temporal behavior is independent of the others and may be studied independently. In embedded systems, one of the primary design goals is to maximize the use of the available resources while still achieving the desired level of performance. Depending on the field of application, this objective can have a sizeable influence on the total cost of the system in terms of factors such as the available budget, energy usage, weight distribution, etc. To circle back to an earlier point, multicore architectures offer a low-cost alternative that improves processing performance while simultaneously minimizing the amount of power that is consumed. In point of fact, as was mentioned before, raising the operating frequency of a single CPU would result in greater levels of both heat and power consumption. However, the manner in which tasks are distributed among processors in a multicore platform has a significant impact on the number of active cores required to run the application. As a result, minimizing resource consumption might seems as an interesting approach to handling the issues of schedulability, but this idea will be circled back to later again in this paper.

Assessments of scheduleability, have been established in order to verify that each scheduling strategy takes into account the time constraints imposed by task sets. The outcome of each test of schedulability indicates whether or not a project can be scheduled in an appropriate manner. This is because it is difficult to pinpoint the precise conditions under which a deadline may be missed (i.e., guaranteeing all executions meet their deadline requirements). In addition, the complexity of today's embedded systems is constantly growing, and software is typically segmented into a large number of concurrent applications. Each concurrent application consists of a set of activities that have unique capabilities and constraints, but they all share the same resources. In a situation like this one, isolating the temporal behavior of real-time applications is absolutely necessary in order to prevent interference between vital processes that are caused by a back and forth interaction.

$$k = Qk / Pk$$

Fig. 3. Formula

III. Outline of the Subject

As a solution, this paper will focus on the ideas and approaches discussed in a paper bearing the same title, "Partitioning Real-Time Applications Over Multicore Reservations" authored by Giorgio Buttazzo, Enrico Bini, and Yifan Wu., specifically the "Heuristic Partitioning" approach. [1]

A "Resource Reservation" approach can be used to provide temporal isolation by partitioning the CPU processing power into a number of reservations, each of which is comparable to a virtual processor with restricted speed. They define reservation as a pair (Qk,Pk) indicating that Qk units of time are available during the period Pk. The idea is to have that virtual processor be equivalent to its hardware counterpart. The fundamental benefit of this technique is that a program assigned to a virtual machine can be guaranteed to run in "isolation" (i.e., without interference from other applications in the system) based solely on its timing needs and given bandwidth. Altough this would come at a cost to the overall processing capacity, since it has to account for the processing power of the virtual machine itself. Overruns in one application do not affect the timing of other applications in this scenario, making resource reservation an effective technique for isolating real-time and non-real-time application executions. Interestingly, this method would work entirely differently on multicore systems. The authors use the multiprocessor model described by Funk et al., [4] in which a group of sequential machines is described by their speeds, as an interesting example. But it is important to note that the authors outlined specific boundaries:

- It is possible for a task that is being executed on a processor to be interrupted in the middle of its execution and for that task's execution to be restarted at a later time. It is taken for granted that there is no penalty associated with this kind of preemption.
- It is possible to resume the execution of a task that was previously interrupted on one processor by switching to another processor, or continuing the execution on the same processor. It is taken for granted that there are no penalties associated with such a relocation.
- At any given moment in time, each task can only ever be executed on a single processor.

Funk et al., demonstrated that a set of tasks scheduled using the global Earliest Deadline First (EDF) algorithm (with migrations) and requiring a total bandwidth of 120 percent has a better chance of being successfully scheduled on two virtual processors with bandwidths of 100 and 20 percent, rather than two virtual processors with the same bandwidth of 60 percent. When task transfer is not an option, however, cramming the

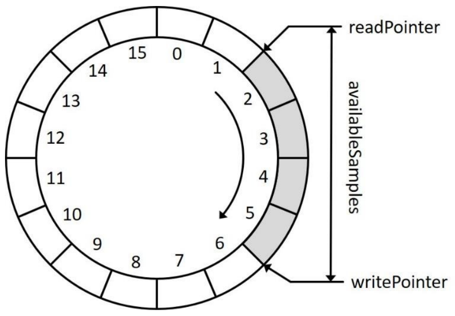

Fig. 4. Cyclic Asynchronous Buffer

bandwidth into full reservations isn't always the best strategy. Consider a periodic application with five tasks and a period of 10 (deadline = period). In this scenario, the application requires U=190 percent bandwidth, and a workable schedule can be created by combining three reservations of 80 percent, 60 percent, and 50 percent. If the bandwidth is provided by two reservations of 100 percent and 90 percent, however, there is no practicable alternative. [1]

A. Considering Resource Minimization

Before the solution by Buttazzo et al., is explored any further, it is worthwhile considering the concept of resource minimization as a possible aid. Although a number of methods for partitioning real-time applications across a set of processing elements have been suggested in the published research, these strategies have primarily concentrated on the problems of achieving a feasible schedule or minimizing the maximum response time among tasks as opposed to the problem of minimizing the amount of computational resources required. It is essential to maximize available resources in order to minimize energy consumption and maintain temperature control on chips, given the growing development of multicore embedded systems. Wang et al., [5] describe a method to achieve this using directed acyclic graphs (DAGs), which are a graphical representation that may be used to explain how a series of activities with preset priority constraints make up a real-time application. The DAG is taken as a representation of the application in its most parallel state. This indicates that each action consists of a single core component, but during the execution phase, tasks are not permitted to make use of any blocking primitives and can be preempted at any time. Avoiding the utilization of blocking primitives in task code does not necessarily imply that there will be no possibility of data sharing across tasks. Thus, in practice, it is possible for tasks to interact through non-blocking mechanisms that make use of memory duplication to avoid blocking on essential sections of the system. One useful piece of technology that falls under this category is known as the Cyclic Asynchronous Buffer (CAB), but the authors also use a policy known as Locality-Aware Dynamic Mapping (LADM). [5] The concept of LADM is when interference is discovered at two cores, for example, the application deactivates two threads and reassigns the work to the other threads. The method also detects the

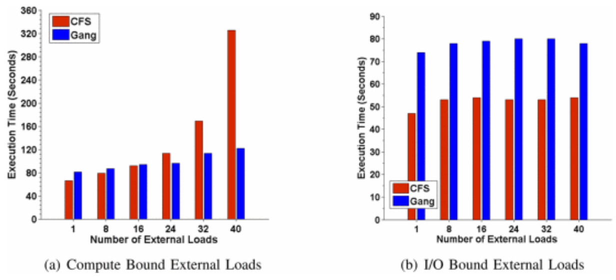

Fig. 5. Wang et al. method of resource minimization [5]

accessibility of additional resources and uses them as soon as they become available. Through a combination of all these elements mentioned above, the authors design an algorithm that successfully achieves resource minimization.

However, there is no need to delve further into this topic, as they ultimately conclude that each application has to be adapted and compatible with CAB and LADM, which stands in the way of universal compatibility. As such, the concept of resource minimization will be left on the back burner for the remainder of this paper.

IV. Heuristic Partitioning

For this approach the authors use three algorithms and then proceed to run simulations to evaluate their performance and runtime cost, but this paper will only analyze the most effective method out of the three.

When a program is being performed, the sequence in which statements, instructions, and function calls are executed and evaluated is referred to as the control flow, or simply *flow*. If *Mlow* is the lower constraint on the number of cores that are required by the application, then the application must have at least *Mlow* flows in order for the first heuristic algorithm to be effective. As a consequence of this, the algorithm initiates the process of creating *Mlow* flows by selecting the longest paths available in the precedence graph (a graph that delineates concurrency). The critical path is initially introduced into flow F1 to get things started. Then, the algorithm will attempt to squeeze the crucial path of the remaining graph into one of the flows that are already in place. If this cannot be done (meaning that the resulting schedule cannot be implemented), the current critical path will be transferred into the new flow. This process is performed several times until *Mlow* flows are produced. Then, the remaining tasks in the graph are chosen by reducing the amount of time it takes to compute them, and a Best Fit strategy is used to insert them into the current flows. The Best Fit strategy involves assigning the smallest free partition that fulfills the requesting process's requirements. When the schedulability of any of the existing flows cannot be guaranteed, a new flow will be established. A new flow will be formed in the event that the schedulability cannot be ensured inside any of the already existing flows. The authors calculate *Mlow* as shown in Fig. 6, where *nh* represents the number of heavy tasks.

$$M_{\text{low}} = \max\{n_h, \lceil U \rceil\}$$

Fig. 6. First heuristic method formula [1]

```
function HEURISTIC($\Gamma$, nPaths)
    $\Gamma_r \leftarrow \Gamma; S \leftarrow \emptyset$;
    if $D > C^s$
        $S \leftarrow \{\Gamma\}$;
        return $S$;
    end if
    while NUM($S$) $\leq$ nPaths
        let $P$ be the critical path of $\Gamma_r$;
        FITorNEW($P$, $S$);
        $\Gamma_r \leftarrow \Gamma_r \backslash P$;
    end while
    for each $\tau_i$ in $\Gamma_r$
        FITorNEW($\{\tau_i\}$, $S$);
    end for
    return $S$;
end function

function FITorNEW($P$, $S$)
    for each $F \in S$
        $F' \leftarrow F \cup P$;
        if $F'$ is schedulable
            $S \leftarrow S \setminus F \cup \{F'\}$;
            return
        end if
    end for
    $S \leftarrow S \cup \{P\}$;
end function
```

Fig. 7. First heuristic method pseudocode [1]

Each loop executes a maximum of n times (the number of paths created), and the FITorNEW function's feasibility check has O(n2) complexity, since the dbf function's maximum is performed for all activation times and deadlines. An experiment was carried out taking into consideration a fully parallel application (that is, without precedence relations) with random computation times generated using a uniform distribution in [1,10]. The goal of the experiment was "to test the runtime behavior of the search algorithm and the heuristics, as well as the effectiveness of the pruning rule". [1] Both the application deadline, which was defined as D = $(C^p + C^s)$/2, and the context switch overhead, which was defined as (delta) =1.6, were adjusted accordingly. The amount of time required to complete a method's tasks was evaluated in milliseconds and compared to the total number of jobs. The performance of the branch and bound method was evaluated based on a variety of values of the pruning parameter (delta). When smaller values of (delta) are used, it is possible to reduce the number of necessary steps by a substantial amount. If you set to a low setting, the amount of bandwidth you lose will be almost nonexistent. This can be rationalized on an intuitive level by contemplating the fact that a high total B is frequently necessary when dealing with a large number of flows. In addition, when compared with the optimal search, the heuristic approach has a substantially shorter running time than the ideal search does, and this is the case even when a harsh pruning criterion ((delta)=1) is applied.

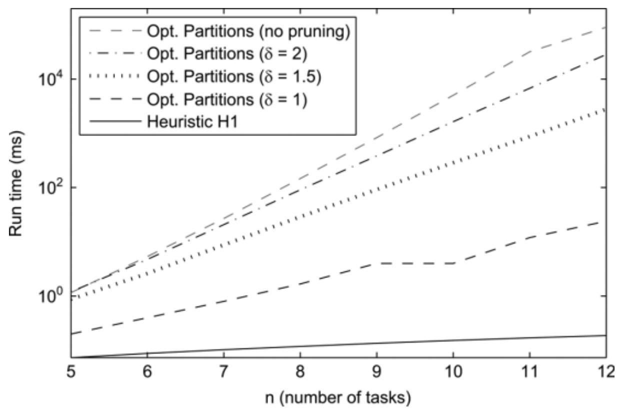

Fig. 8. First heuristic method results [1]

V. Conclusion

As the authors (Buttazzo al., [1]) have demonstrated, the heuristic approach seems to be a useful and valuable algorithm to solve the multicore scheduling problem for applications that require the use of the same resources. It is not too convoluted and it is applicable to both small and large applications alike. But, as demonstrated, the further development of scheduling algorithms for multicore systems is a necessary step, especially as the complexity and idiosyncrasy of these systems continues to develop, as well, in parallel and in an exponential manner. That is also true concerning the systems' respective applications, as they also are moving in a trajectory of complexity and inter-connectivity with other applications.

References

[1] Partitioning Real-Time Tasks With Replications on Multiprocessor Embedded Systems, Giorgio Buttazzo, Enrico Bini, etl al.

[2] Composition of Schedulability Analyses for Real-Time Multiprocessor Systems, Jinkyu Lee, Kang G. Shin and Arvind Easwaran

[3] Integrating Multimedia Applications in Hard Real-Time Systems, L. Abeni and G. Buttazzo

[4] On-Line Scheduling on Uniform Multiprocessors, S. Funk, J. Goossens, and S. Baruah

[5] Controlled Contention: Balancing Contention and Reservation in Multicore Application Scheduling, Jingjing Wang, Nael Abu-Ghazaleh, and Dmitry Ponomarev

Other material used:

- https://commons.wikimedia.org/wiki/File:Cc-NUMA_Multi-Core.svg
- https://commons.wikimedia.org/wiki/File:Real-time_scheduling_model.svg
- https://media.geeksforgeeks.org/wp-content/uploads/20191226233215/gfg_reservatin_messages.png
- https://www.researchgate.net/figure/\\
A-circular-buffer-allows-asynchronous-write-and-read-operations-and-\
\can-store-N-samples_fig6_334288006