Constraint Programming Modeling for the Task Scheduling Problem with Data Storage at MPSoCs

Christos Valouxis · Christos Gogos · Panayiotis Alefragis · George Theodoridis · Nikolaos Voros

Abstract Multi Processor System on Chips are expected to exhibit increased performance when extra memory chips exist close to their processing units. Given an architecture of such a system, equipped with private fast memories for each processor and a slower shared memory for all of them, we propose a constraint programming model that is capable of scheduling program workflows annotated with data usage information. Our approach examines usage of the data objects by individual tasks and arranges their positioning to certain private memories or to the shared memory during the execution. The objective is to minimize the makespan of the workflow. Experiments on artificially generated workflows show that values close to theoretical lower bounds can be achieved for moderate problem sizes under reasonable execution time.

Keywords task scheduling · workflows · constraint programming

1 Introduction

Modern multicore embedded architecture provide significant processing power that is difficult to harness. Simplified algorithms that have been used to sched-

C. Valouxis
University of Patras, Greece
Tel.: +30-2610-996434
Fax: +30-2610-996820
E-mail: cvalouxis@gmail.com

C. Gogos
Technological Educational Institute of Epirus, Greece

P. Alefragis
Technological Educational Institute of Western Greece

G. Theodoridis
University of Patras, Greece

N. Voros
Technological Educational Institute of Western Greece

ule sequential programs can not easily support the complex interdependencies between architecture components. The most used optimization goal is to minimize the total execution time. The problem is that each architecture characteristics can have significant impact in the solution approach that should be followed to create high quality solutions. To overcome this problem there are approaches that create tool flows [2],[10] that start from a high level representation of a program and in the end generate an equivalent working program that uses the available processing cores of the architecture. Most of these flows have domain specific languages to represent architectures [11], providing information about the architecture to the algorithms. In the context of the H2020 ARGO project [2], our goal is to minimize the total execution time of the worst case execution (WCET) [14], i.e. when the program fails to efficiently utilize the architecture components and more than one processors antagonize to use shared resources. This is a special case of the problem that is very important in hard real time applications, like industrial image processing, avionics, autonomous driving, safe critical control to guarantee that a program will finish its execution in a given time interval. The problem is analogous to the multi-mode resource-constrained project scheduling problem where the execution time of each task is not fixed. To make the problem even more difficult the execution time of each task has a cyclic dependency with the assignment of tasks to processing cores, the sequencing of tasks in each core and the assignment of variables to different memory resources.

The input program is represented by a Directed Acyclic Graph (DAG) with annotations on each node regarding its usage on data objects.The classic DAG scheduling problem (tasks with precedences but no data objects or memory mapping requirements) is known to be NP-complete [4], and it is very hard to be solved efficiently, and it has attracted much attention. Many algorithms have been proposed in order to address the problem [9]. Heuristics [1], evolutionary algorithms [6], mathematical programming approaches [3], [13] and constraint programming [5] approaches are common categories that most of the published methods fall into. In particular, due to their low complexity and good efficiency, heuristics seem to dominate the field. Nevertheless, other more computationally expensive methods achieve in general better results. Moreover, the case of hybridizations among the previously mentioned methods seems to be another promising approach. The inclusion of data objects and memory hierarchies to the DAG scheduling problem moves it closer to the problem as it is manifested in actual multicore platforms. Two major types of private memories used are either cache or scratchpad. The difference is that in the first case the cache memory can not be explicitly programmed while on the second case the programmer has the ability to issue commands that explicitly load and unload data objects in and out of the scratchpad memory when needed. Several researchers have worked with the both types of systems. For MPSoCs with caches relevant research can be found at [7] while for MPSoCs with scratchpad some relevant work is [12], [8]. In this work, a multi core architecture with scratchpad memories is assumed. Each processor has private access to a small fast memory and they all share a much slower larger memory.

The rest of this paper is organized as follows. The next section presents the problem description. Section 3 describes the constraint programming model that we developed in order to address the problem. Next, section 4 presents results of our model to test problems of various sizes. Finally, section 5 completes the paper with conclusions and future work.

2 Problem description

The aim of this work is to propose an approach that efficiently executes programs consisting of several interdependent tasks in systems with multiple execution units. The internal structure of each program is captured by a DAG where each node is a task and each edge from task A to task B enforces the completion of task A before the start of execution of task B. Such programs are commonly referred to as workflows. In our case, for each task two pieces of information are known: i) the execution time of each task when scheduled on each processor and ii) the set of data objects that each task uses (creates, reads or writes) alongside with the frequency of these actions and the size of each data object.

The computer system that will execute the workflows is assumed to be a simplification of an actual MPSoC. Each processor owns a private memory and the execution time of each task might vary among processors since processors are heterogeneous. Furthermore, a shared memory exists and all processors can access it, although at a much lower speed than their private memories. Memory sizes and relative speeds for shared and private memories are considered to be known. Since tasks access data objects during their execution, certain placements of data objects to private memories of processors that will execute the appropriate tasks is expected to be beneficial. Nevertheless, private memories are small and they cannot possibly accommodate all relevant data objects. Therefore, a scarce resource planning problem emerges.

Several constraints have to be satisfied in order to have a solution. Some obvious ones are the following: each task should be executed by one and only one processor, no overlap of task execution at the same processor should exist and order of execution as imposed by the problems' DAG should be respected. Furthermore, each data object should exist either in a scratchpad memory or in the shared memory and scratchpad capacities should be respected at all times. Among solutions that satisfy all constraints the one with the lowest makespan should be selected. A more detailed description of the model's variables, constraints and objective function exists in section 3.

2.1 Input data

Problem input data records the execution time of each task on each processor provided that all data objects that are accessed by the task are located at the shared memory and none of them at private memories. If a data object

that is accessed by a task is located at the private memory of the processor that will execute the task, then the task will execute faster. The time saved is computed by multiplying the number of the data object accesses by an appropriate gain factor. Both of these parameters are included in the input data for each valid task and data object combination. Moreover, each problems' input data contains information about the size of each data object, the capacity of each processors' private memory and the nodes and edges of the problems' DAG.

An estimation of a lower bound for the makespan of a workflow can be found by assuming that each task will be scheduled to the fastest processor for this task and that all required data objects by the task will exist at the private memory of this processor. The approach that will be presented in the following paragraphs manages to produce schedules that have makespans close to these theoretical and over optimistic lower bounds.

2.2 Graph generation

A simple graph generation algorithm is used in order to generate artificial graphs. Tasks are assigned to levels and edges between tasks of adjacent levels are randomly created so as the graph to have a single source and a single sink node. Then, a number of data objects are created. Each data object is assigned to tasks that either read or modify it according to the existing edges. An example of such a graph is shown in Fig. 1.

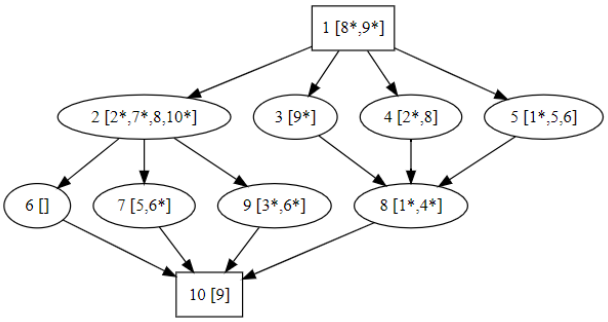

Fig. 1 graph_10_2: Artificially generated graph, 10 tasks, 10 variables (* on variables indicate modification by the task)

3 The Constraint Programming model

The Constraint Programming (CP) model that will be presented makes use of interval, binary and sequence variables. An interval variable represents an

interval of time during which something will occur. An interval variable might be optional, meaning that it might not be materialized at the final solution. On the other hand, a sequence variable defines the total order over a set of interval variables and can be used in order to enforce a constraint like no overlapping for the items of the sequence.

Variables: Several variables exist in our model which are listed bellow:

- Interval variables $task\_iv[t]$ that define the start and finish time of each task t.
- Optional interval variables $task\_oiv[t][p]$ that define the start and finish time of each task t at each processor p.
- Interval variables $load\_iv[t]$ that define the start and finish time for loading all data objects needed by task t.
- Interval variables $store\_iv[t]$ that define the start and finish time for storing all data objects that task t modifies.
- Binary variables $load\_b[t][v]$ that assume value 1 if task t needs to load data object v from the main memory or 0 otherwise.
- Binary variables $store\_b[t][v]$ that assume value 1 if task t needs to store data object v to the main memory or 0 otherwise.
- Interval variables $z\_iv[v]$ that define the start and finish time that a data object v exists at a scratchpad.
- Optional interval variables $z\_oiv[v][p]$ that define the start and finish time of a data object v at the scratchpad of processor p. For each data object v there exists one more variable $z\_oiv[v][|P+1|]$ that defines the start and finish time of v to the main memory.
- Sequence variables $task\_sv[p]$ that for each processor p define all corresponding optional interval variables $taskp\_oiv[t][p]$. These variables are used in order to avoid simultaneous task execution at each processor.

The relation among interval variables $task\_iv[t]$, $load\_iv[t]$ and $store\_iv[t]$ for the same task t is graphically depicted in Fig. 2.

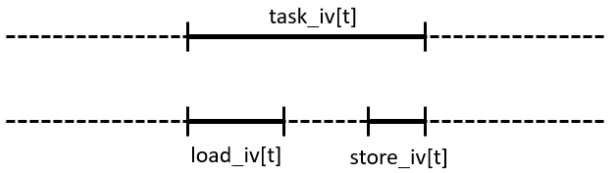

Fig. 2 Start and finish of interval variable *task_iv* should be synchronized with interval variables *load_iv* and *store_iv* respectively

Constraints: The constraints of the problem can be summarized as follows:

- c01: Each task has to be scheduled to exactly one processor.

- c02: No two tasks should be executed simultaneously at the same processor.
- c03: Task dependencies as defined at the problems' DAG should be respected.
- c04: Each task should start at the time when the data objects it uses start loading.
- c05: Each task should finish at the time that it finishes storing the data objects it modified.
- c06: Constraint that sets the size of interval variable $load\_iv[t]$.
- c07: Constraint that sets the size of interval variable $store\_iv[t]$.
- c08: No more that one load or store are allowed at the same time.
- c09: Constraint that sets the size of interval variable $task\_iv[t]$.
- c10: Each task should load a variable when it is needed.
- c11: Each task should store variables when it is needed.
- c12: Each data object should exist either at a scratchpad or at the main memory.
- c13: Scratchpad capacities should not be exceeded.
- c14: Constraint that sets the start and end times of interval variables $z\_iv[t]$.

Objective: The objective is to minimize the end time of the sink task at the problems' DAG.

3.1 Analysis of key constraints

Next, we present selected constraints that are key to the model. In the formulas that follows, predicates that are directly related to the ones defined at IBM ILOG CP solver are used. These are $sizeOf$, $startOf$, $endOf$ and $presenceOf$. The first three return the size, the start time and the end time of an interval variable, while the last one returns 1 if an optional interval variable belongs to the solution or 0 otherwise.

Moreover, the following data are expected to exist. $Exec[t][p]$ is the execution time of task t on processor p when all the data objects that task t accesses are in the shared memory. $VarsThatTaskUses[t]$ and $VarsThatTaskModifies[t]$ are the lists of data objects that task t uses or modifies in the program respectively. Finally, $StartAfter[t]$ is a list of tasks that should start after task t due to the dependencies between tasks as captured at the DAG of the problem.

Constraints c06: Each constraint of this set computes the time that a task must spent in order to load all data objects that it cannot find locally or might have been modified earlier by tasks running on other processors. The time spent for loading each data object is equal to the estimated gain (r.gain) of having the data object at the scratchpad that the task can access. Binary variable $load\_b[t][r.var\_id]$ assumes value 1 when task t needs to load data object identified by $r.var\_id$. Constraint c10 is responsible for setting $load\_b[t][r.var\_id]$ value correctly.

$$sizeOf(load\_iv[t]) = \sum_{\substack{r \in VarR: \\ r.task\_id = t}} load\_b[t][r.var\_id] * r.gain \qquad \forall t \in 1..T \quad (1)$$

Constraints c07: Like c06 constraints, each one of the c07 constraints computes the time spent on storing data objects that a task modified and are needed by other tasks.

$$sizeOf(store\_iv[t]) = \sum_{\substack{r \in VarR: \\ r.task\_id = t}} store\_b[t][r.var\_id] * r.gain \qquad \forall t \in 1..T \quad (2)$$

Constraints c09: This set of constraints sets a lower bound for the size of each task execution interval. Since the problem is of minimization for the last task this put pressure on obtaining the lowest possible values for task sizes. The constraint subtracts from the execution time of task t on processor p the time that each data object saves when it is located at the scratchpad of processor p. Moreover, sizes of possible loads and stores of data objects are added.

$$sizeOf(task\_iv[t]) \geq \sum_{p \in 1..P} presenceOf(task\_oiv[t][p]) * \Big(Exec[t][p] - \sum_{\substack{r \in varR: \\ r.task\_id = t}} presenceOf(z\_oiv[r.var\_id][p]) * r.gain * r.var\_freq\Big) + sizeOf(load\_iv[t]) + sizeOf(store\_iv[t]) \qquad \forall t \in 1..T \quad (3)$$

Constraints c10: These constraints specify whether tasks should load data objects from the main memory or not. For a task $t1$, that uses data object v this occurs when another task $t2$ which executes at a different processor than $t1$'s has modified the data object and no other task $t3$ (that executes after $t2$ and earlier than $t1$) running at the same processor as $t1$, uses the data object. In other words, if no task $t3$ exists, that has already loaded a data object v to the scratchpad of the processor that $t1$ will execute, then $t1$ has to load the data object by itself.

$$
\begin{aligned}
&presenceOf(z\_oiv[v][p]) \wedge presenceOf(task\_oiv[t1][p]) \wedge \\
&\quad \sum_{\substack{p1 \in 1..P: \\ p1 \neq p}} presenceOf(task\_oiv[t2][p1]) = 1 \wedge \\
&\quad startOf(task\_oiv[t1][p]) \geq endOf(task\_iv[t2]) \wedge \\
&\quad \sum_{\substack{t3 \in 1..T: \\ t3 \neq t1 \wedge t3 \neq t2 \wedge \\ v \in VarsThatTaskUses[t3] \wedge \\ t2 \notin StartAfter[t3] \wedge \\ t3 \notin StartAfter[t1]}} \Big(presenceOf(task\_oiv[t3][p]) \wedge \\
&\quad startOf(task\_oiv[t3][p]) \geq endOf(task\_iv[t2]) \wedge \\
&\quad startOf(task\_oiv[t1][p]) \geq startOf(task\_oiv[t3][p]) \Big) = 0 \\
&\quad \implies load\_b[t1][v] = 1 \\
&\forall (t1 \in 1..T, p \in 1..P, t2 \in 1..T, v \in VarsThatTaskUses[t1] : \\
&\quad v \in VarsThatTaskModifies[t2] \wedge t1 \neq t2 \wedge t2 \notin StartAfter[t1])
\end{aligned} \tag{4}
$$

Constraints c11: These constraints are analogous to constraints c10. They specify whether a task should store a data object to the main memory or not. For a task $t1$, that modifies data object v this occurs when another task $t2$ which executes at a different processor than $t1$'s uses the data object and no other task $t3$ (that executes after $t1$ and earlier than $t2$) running at the same processor as $t1$, modifies the data object.

$$
\begin{aligned}
&presenceOf(z\_oiv[v][p]) \wedge presenceOf(task\_oiv[t1][p]) \wedge \\
&\quad \sum_{\substack{p1 \in 1..P: \\ p1 \neq p}} presenceOf(task\_oiv[t2][p1]) = 1 \wedge \\
&\quad startOf(task\_iv[t2]) \geq endOf(task\_oiv[t1][p]) \wedge \\
&\quad \sum_{\substack{t3 \in 1..T: \\ t3 \neq t1 \wedge t3 \neq t2 \wedge \\ v \in VarsThatTaskModifies[t3] \wedge \\ t1 \notin StartAfter[t3] \wedge \\ t3 \notin StartAfter[t2]}} \Big(presenceOf(task\_oiv[t3][p]) \wedge \\
&\quad startOf(task\_oiv[t3][p]) \geq endOf(task\_oiv[t1][p]) \wedge \\
&\quad startOf(task\_iv[t2]) \geq startOf(task\_oiv[t3][p]) \Big) = 0 \\
&\quad \implies store\_b[t1][v] = 1 \\
&\forall (t1 \in 1..T, p \in 1..P, t2 \in 1..T, v \in VarsThatTaskModifies[t1] : \\
&\quad v \in VarsThatTaskUses[t2] \wedge t1 \neq t2 \wedge t2 \notin StartAfter[t1])
\end{aligned} \tag{5}
$$

The full listing of the CP model, using OPL which is the modeling language used in IBM ILOG's Optimization Studio[1], can be found at `https://github.com/chgogos/task_scheduling_patat2018`.

4 Experiments

A number of experiments that test and validate our model were run. For example, the results that are produced for the graph of Fig. 1 when scheduled on two processors are shown below:

```
// solution with objective 3714
Task1 [0 - 474] scheduled at processor 1
Task2 [474 - 1188] scheduled at processor 1
Task3 [1188 - 1852] scheduled at processor 1
Task4 [1852 - 2307] scheduled at processor 1
Task5 [474 - 1149] scheduled at processor 2
Task6 [2307 - 2997] scheduled at processor 1
Task7 [1188 - 1915] scheduled at processor 2
Task8 [2447 - 3044] scheduled at processor 2
Task9 [1915 - 2447] scheduled at processor 2
Task10 [3044 - 3714] scheduled at processor 2
var: 1 is mapped to proc: 2 start=474 end=3044[size=25]
var: 2 is mapped to proc: 1 start=474 end=2307[size=25]
var: 3 is mapped to proc: 2 start=1915 end=2447[size=26]
var: 4 is mapped to proc: 2 start=2447 end=3044[size=21]
var: 5 is mapped to proc: 2 start=474 end=1915[size=21]
var: 6 is mapped to proc: 2 start=474 end=2447[size=21]
var: 7 is mapped to proc: 1 start=474 end=1188[size=22]
var: 8 is mapped to proc: 1 start=0 end=2307[size=24]
var: 9 is mapped to proc: 1 start=0 end=3714[size=21]
var: 10 is mapped to proc: 2 start=474 end=1188[size=22]
```

The results obtained over 16 problem instances (datasets) are summarized in Table 1. The name of each dataset (T_P) provides information about the number of tasks (T) and the number of processors (P) employed. All experiments were executed on an i7 7700K, 16GB RAM workstation running Windows 10 Professional. The solver was IBM ILOG CP 12.8 and a time limit of 60 seconds was given for each run. Column LB refers to lower bounds obtained by assuming scratchpads of infinite size. Likewise, upper bounds resulted assuming no scratchpads.

For the six problem instances that reached optimality, the average distance from the initial lower bound, was 7.25%. For the rest ones, the average distance from the lower bound was 7.62%. The proximity of these two values indicate that even for the cases that did not reach optimality, the obtained solutions might be very close to the optimal ones.

[1] `https://www.ibm.com/products/ilog-cplex-optimization-studio`

Table 1 Experimental results under time limit of 60 seconds, LB=Lower Bound, UB=Upper Bound, MS=Makespan

Dataset	LB	UB	MS	MS % LB	Optimal	Time (sec)
10_2	3,575	4,240	3,714	3.89%	Yes	7
10_4	2,161	2,610	2,325	7.59%	Yes	1
10_8	1,880	2,190	2,102	11.81%	Yes	3
15_2	4,721	5,570	4,871	3.18%	Yes	20
15_4	2,883	3,570	3,135	8.74%	Yes	15
15_8	2,608	2,960	2,824	8.28%	Yes	43
20_2	6,177	7,270	6,356	2.90%	No	60
20_4	3,392	4,130	3,662	7.96%	No	60
20_8	2,677	3,400	3,060	14.31%	No	60
30_2	8,564	10,230	9,100	6.26%	No	60
30_4	4,961	5,930	5,330	7.44%	No	60
30_8	3,628	4,080	3,759	3.61%	No	60
50_2	14,064	16,790	14,761	4.96%	No	60
50_4	6,942	8,440	7,662	10.37%	No	60
50_8	3,960	4,810	4,365	10.23%	No	60
50_16	3,761	4,290	4,067	8.14%	No	60

5 Conclusions

Constraint Programming is a powerful technique. Increased computational power alongside with highly capable solvers like IBM ILOG's CP Optimizer can nowadays solve combinatorial problems that may have been difficult or impossible to solve a few years ago. In this work, the problem of scheduling tasks with dependencies and accessing data objects in shared and private memories has been solved using a Constraint Programming formulation that gives high quality results.

Acknowledgements This work was funded by the European Union under the Horizon 2020 Framework Program under grant agreement ICT-2015-688131, project 'WCET-Aware Parallelization of Model-Based Applications for Heterogeneous Parallel Systems (ARGO)'

References

1. Bittencourt, L.F., Sakellariou, R., Madeira, E.R.: DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In: Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on, pp. 27–34. IEEE (2010)
2. Derrien, S., Puaut, I., Alefragis, P., Bednara, M., Bucher, H., David, C., Debray, Y., Durak, U., Fassi, I., Ferdinand, C., et al.: WCET-aware parallelization of model-based

applications for multi-cores: The ARGO approach. In: Proceedings of the Conference on Design, Automation & Test in Europe, pp. 286–289. European Design and Automation Association (2017)
3. Emeretlis, A., Theodoridis, G., Alefragis, P., Voros, N.: A logic-based benders decomposition approach for mapping applications on heterogeneous multicore platforms. ACM Transactions on Embedded Computing Systems (TECS) **15**(1), 19 (2016)
4. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
5. Hladik, P.E., Cambazard, H., Déplanche, A.M., Jussien, N.: Solving a real-time allocation problem with constraint programming. Journal of Systems and Software **81**(1), 132–149 (2008)
6. Kelter, T., Borghorst, H., Marwedel, P.: WCET-aware scheduling optimizations for multi-core real-time systems. In: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on, pp. 67–74. IEEE (2014)
7. Liu, T., Zhao, Y., Li, M., Xue, C.J.: Task assignment with cache partitioning and locking for WCET minimization on MPSoC. In: Parallel Processing (ICPP), 2010 39th International Conference on, pp. 573–582. IEEE (2010)
8. Salamy, H., Ramanujam, J.: An effective solution to task scheduling and memory partitioning for multiprocessor system-on-chip. IEEE transactions on computer-aided design of integrated circuits and systems **31**(5), 717–725 (2012)
9. Sinnen, O.: Task scheduling for parallel systems, vol. 60. John Wiley & Sons (2007)
10. Stripf, T., Oey, O., Bruckschloegl, T., Becker, J., Rauwerda, G., Sunesen, K., Goulas, G., Alefragis, P., Voros, N.S., Derrien, S., et al.: Compiling Scilab to high performance embedded multicore systems. Microprocessors and Microsystems **37**(8), 1033–1049 (2013)
11. Stripf, T., Oey, O., Bruckschloegl, T., Koenig, R., Goulas, G., Alefragis, P., Voros, N.S., Potman, J., Sunesen, K., Derrien, S., et al.: A compilation-and simulation-oriented architecture description language for multicore systems. In: Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on, pp. 383–390. IEEE (2012)
12. Suhendra, V., Raghavan, C., Mitra, T.: Integrated scratchpad memory optimization and task scheduling for MPSoC architectures. In: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, pp. 401–410. ACM (2006)
13. Valouxis, C., Gogos, C., Alefragis, P., Goulas, G., Voros, N., Housos, E.: DAG scheduling using integer programming in heterogeneous parallel execution environments. In: Proceedings of the Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2013), pp. 392–401 (2013)
14. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) **7**(3), 36 (2008)