Parallel Programs Scheduling with Architecturally Supported Regions

Łukasz Maśko[1(✉)] and Marek Tudruj[1,2]

[1] Institute of Computer Science, Polish Academy of Sciences,
Jana Kazimierza 5, 01-248 Warsaw, Poland
{masko,tudruj}@ipipan.waw.pl
[2] Polish-Japanese Academy of Information Technology,
Koszykowa 86, 02-008 Warsaw, Poland

Abstract. Scheduling of programs for hierarchical architectures of Chip Multi-Processor (CMP) modules interconnected by global data networks is the subject of this paper. The CMP modules are of double nature: architecturally specialized modules which execute time-critical computations and standard CMP modules which interconnect the specialized ones. Inside application programs, so called architecturally supported regions are identified meant for efficient execution on dedicated architecturally supported modules. Programs are represented by macro dataflow graphs built of architecturally supported nodes and program glue nodes. The paper proposes a new task scheduling algorithm for programs meant for execution in such CMP-based systems. The algorithm is based on list scheduling with modified ETF (Earliest Task First) heuristics. It is assessed by experiments based on simulation of program execution which shows parallel speedup improvements.

Keywords: Parallel programming · Program graph scheduling · Parallel architectures · Heterogeneity

1 Introduction

Putting a large number of cores inside a processor chip in the Networks on Chip (NoCs), Systems on Chip (SoCs) or Chip Multi-Processors (CMPs) technologies [1,2] sets new challenges in the design of core interconnection networks. Designers have to be conscious of technology limitations, such as power dissipation, wire delays, signal cross talks and silicon area, which make designing large monolithic CMPs problematic. The self-imposing solution are modular hierarchical structures of many CMPs interconnected by an external global network with improved efficiency and scalability. Although already viable in the current chip technology, the ideas of core clustering inside CMPs nor CMPs clustering have not been yet investigated in a mature way.

In globally interconnected systems of CMPs modules some CMPs can be strongly architecturally supported to provide high parallel speedup for some time-critical computations, while other CMPs can remain standard multicore

R. Wyrzykowski et al. (Eds.): PPAM 2015, Part II, LNCS 9574, pp. 77–89, 2016.
DOI: 10.1007/978-3-319-32152-3_8

processors. Usually, the architecturally supported CMPs are more intelligent and more difficult to be designed. Such architecturally supported CMP modules usually impose some particular requirements on program structures. This means at least identification of so called architecturally supported regions in the program code. Special features of programs imply special task scheduling algorithms to optimize program execution including adequate graph representation of programs with architecturally supported region nodes.

Scheduling algorithms have been intensively studied for years. Most of techniques like list scheduling, clustering or evolutionary algorithms focus on homogeneous architectures [3,4,8,9]. Extensive surveys of such scheduling algorithms can be found in [4–6]. There are also works dealing with heterogeneous architectures [7], but heterogeneity there is limited to different speed of processing units. In this paper, we assume a different idea of heterogeneity. The system is built of two classes of globally connected computing units (architecturally supported and standard CMP modules). Consequently, we use a macro data flow graph representation in which program graphs consist of two kinds of nodes: architecturally supported and glue nodes.

Our previous paper [10] presents an improved ETF-based list scheduling algorithm [3] for such program and system assumptions. It aims at obtaining better schedules by taking special attention of the order of in which ready graph nodes are scheduled. For this, program graph nodes are assigned scheduling priorities based on static, topological properties of the graph. They do not take task computing nor communication times into account. The priorities are first assigned to architectural nodes after their division into layers, using an analysis of an architectural task activation graph. These priorities are next propagated to glue nodes to control glue node selection during list scheduling to prevent too early execution of such glue nodes, which are not needed for execution of the topologically nearest architectural nodes.

This paper presents a new scheduling algorithm for the program and system assumptions as above, based on modified ETF heuristics with a different definition of task node priorities in the program graphs. Contrary to the previously proposed algorithm, here the priorities are not defined based exclusively on static topological properties of program graphs, but they are determined dynamically based on simultaneous scheduling of the input program graph and scheduling of an equivalent architecturally supported region activation graph. The paper examines the influence of some structural properties of program graphs on the make-spans of such defined program scheduling method and compares its results to those of standard ETF scheduling.

The paper is composed of 4 main parts. The first part presents general system architectural assumptions, the idea of architecturally-supported program regions, and describes structuring of programs for execution in the assumed system architecture. The second part presents the proposed task scheduling algorithm with architecturally supported regions. The third part introduces the program graph measures used for the selection of the adequate scheduling algorithm. The fourth part presents comparative experimental results obtained by simulated use of the proposed algorithm.

2 Architecture-Supported Regions in Application Programs

The general structure of the assumed parallel multi-CMP system with a global data exchange network is presented in Fig. 1(a). We have two kinds of CMPs in this system: Architectural CMPs – ACMPs, which have architecture optimized for execution of some critical program functions and General-Purpose CMPs – GCMPs, similar to typical commercial multicore processors. A program for such architecture can be logically divided into two types of fragments (see Fig. 1b):

- Architecturally-Supported Regions (ASR), whose execution will be accelerated using ACMPs, which can correspond to subroutines and are treated as graph nodes ("architectural nodes"). An ASR will usually have a parallel internal structure. We assume, that the ASR program graph has already been mapped to cores in an ACMP by a separate special scheduling algorithm. Papers [8,9] describe different kinds of such scheduling algorithms meant for heuristic optimization of exemplary ASR modules for efficient execution of parallel matrix multiplication in the ACMP architecture based on communication on the fly. The architecture is especially efficient for parallel programs featuring strong sharing of processed data.
- The glue code, not showing features for special hardware acceleration, which fills gaps between ASRs and will be executed using a set of GCMPs. The glue code is represented as glue nodes.

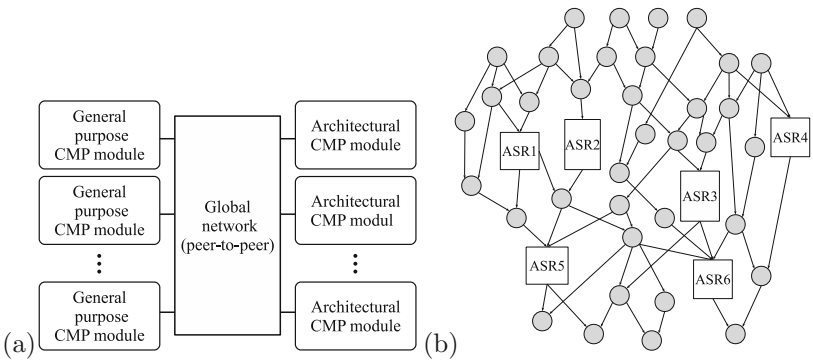


Fig. 1. The general system structure (a) and an application program graph (b)

Formally, a program is described by a macro data flow graph $G = (V, E)$, where V, E are the set of nodes and edges of the graph, respectively. The set V can be divided into two disjoint sets of nodes V_s and V_a, such that $V = V_s \cup V_a$, $V_s \cap V_a = \emptyset$, where V_a contains architectural nodes corresponding to ASRs, while V_s contains glue nodes which exist between nodes from V_a. This division may be determined automatically by a compiler, or manually by a programmer.

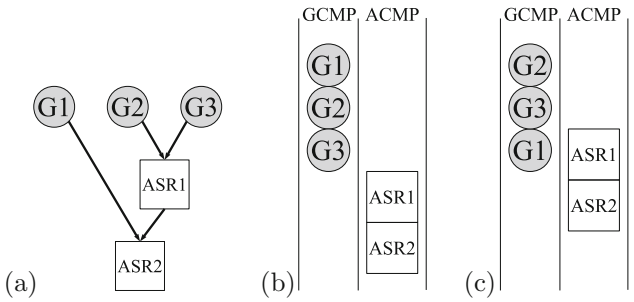


Fig. 2. Motivation for introduction of priorities in the ETF heuristics.

Each node is characterized by its weight representing time needed for execution of instructions included in this node (on ACMP or GCMP, respectively), which is determined automatically by a compiler. Each edge has a weight representing volume of data transmitted with a communication, which such edge represents.

3 Task Scheduling with Architecturally Supported Regions

The assumed multi-CMP architecture requires a proper scheduling algorithm to exploit all its advantageous features. We propose an algorithm, which is derived from the list scheduling technique with the Earliest Task First (ETF) heuristics, but was modified to adjust to the proposed computation model. This algorithm schedules glue nodes to GCMPs and architectural nodes to ACMPs to eliminate stalls of both kinds of resources.

List scheduling is a basic technique for scheduling parallel tasks and ETF [3] is one of the most popular heuristics used for list scheduling. Unfortunately, it has some disadvantages when used in scheduling for heterogeneous systems like that assumed in this paper. The ETF heuristics examine all ready task nodes and selects one with the earliest possible start time. If there are several ready nodes with the same earliest start time, any of them can be selected. It may cause that some nodes would be executed, which should be delayed since their results will be required much "later" in the graph. An example of such situation is depicted in Fig. 2a. Assuming, that the executive system has 1 GCMP with 1 core, 1 ACMP, and weights of all glue nodes G1-G3 are the same, a classical ETF-based list scheduling would give a schedule shown in Fig. 2b. But execution of node G1 should be delayed until nodes G2 and G3 are executed. Then G3 can be executed in parallel with execution of architecturally supported region ASR1, giving a better makespan as in Fig. 2c.

The proposed algorithm aims at minimal program execution time by obtaining permanent loads of ACMP and GCMP modules. The node selection method is modified by special ordering of nodes in the program graph. The nodes are so classified to assure selection in the first place of such ready glue nodes, whose

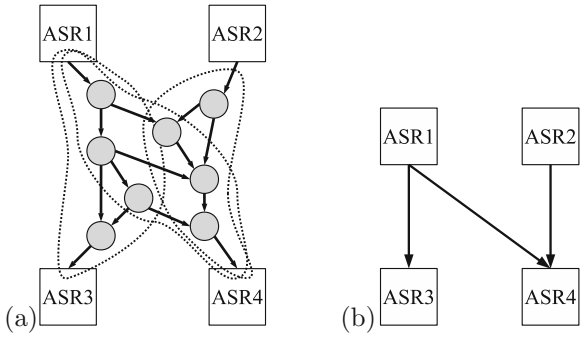


Fig. 3. Conversion of glue subgraphs (a) to edges in RAG (b)

results are required for execution of the topologically nearest ASRs in the graph. The selected glue nodes are scheduled on available processing resources without delaying higher classified graph nodes.

The general algorithm consists of 2 phases. First, the Region Activation Graph (RAG) of the input program graph is created. Then, the program graph is scheduled using list scheduling with modified ETF-based heuristics enriched by an analysis of RAG to set priorities of the ASR and glue nodes.

3.1 Region Activation Graph

A Region Activation Graph (RAG) $RAG_G = (V_a, E')$ is an acyclic unweighted directed graph derived from the original program graph described in previous sections. Nodes in RAG_G correspond to ASR nodes in an initial graph G (the V_a set), while edges depict data dependencies between the ASRs. For two nodes $u, v \in V_a$, an edge $u \rightarrow v \in E'$ exists in RAG_G, if there is a directed path between u and v in G containing only glue nodes. Two ASRs are data-dependent if there exists at least one directed path between them in G with only glue nodes. Two data-dependent ASR nodes are connected in RAG_G with an edge replacing the subgraph consisting of glue nodes and edges on all paths between the two ASRs in G. Subgraphs corresponding to different RAG_G edges may intersect, so some glue nodes can be in more than 2 edges in RAG_G.

Figure 3a presents a part of an input program graph with 4 architectural nodes (ASR1-ASR4) connected with a set of glue nodes. Figure 3b shows a RAG obtained after converting the sets of glue nodes to inter-ASR edges. There are directed paths which connect ASR1 with ASR3 and ASR4 and also directed paths from ASR2 to ASR4. Therefore, we obtain 3 edges in the resulting RAG. There is no directed paths between ASR2 and ASR3 in the input graph, thus, there is no edge between these nodes in RAG.

Algorithm 1. List scheduling algorithm with modified ETF heuristics

{Input: a program graph $G = (V, E)$}
Determine Region Activation Graph $RAG_G = (V_a, E')$, based on graph G.
Let $V_a^R \subseteq V_a$ be the set of nodes without predecessors in RAG_G.
Let V_H be the set of glue nodes without predecessors in G (ready nodes), corresponding to edges leading to nodes from V_a^R (high priority ready glue nodes). Let V_L be the set of other ready glue nodes. Let V_A be the set of ready architectural nodes from graph G.
while $V_H \cup V_L \cup V_A$ is not empty **do**
  Find the node $u \in V_A$ (if available) with the earliest possible execution start time. Let p be the index of a free ACMP, on which execution of u is possible.
  Find the node $v \in V_H$ (if available) with the earliest execution start time. Let q be the core index, on which execution of v is possible.
  Find the node $w \in V_L$ (if available) with earliest execution start time and for which execution ends before execution of the node v found in the previous step may start. If the node v was not found, select any node $w \in V_L$ with the earliest execution start time. Let r be the core index, on which execution of w is possible.
  if the node u has been found **then**
    Schedule u for execution on p^{th} ACMP and remove it from V_A.
    Virtually schedule u in RAG_G and remove it from V_a^R.
    for all descendants u' of u in RAG_G **do**
      if u' becomes ready in RAG_G **then**
        Insert u' in V_a^R. Move from V_L to V_H all nodes corresponding to edges leading to u' in RAG_G.
      end if
    end for
  else
    if the node w has been found **then**
      Schedule the node w for execution on the core r and remove it from V_L.
      Insert to V_L all the descendants of the node w, for which all their predecessors have already been scheduled.
    else
      Schedule the node v for execution on the core q and remove it from V_H. Insert to V_H (or V_L) all the descendants of the node v, for which v was the last scheduled predecessors and which correspond to edges leading to ready nodes in RAG_G (other nodes in RAG_G, respectively).
    end if
  end if
end while

3.2 Scheduling Algorithm Based on RAG Topology Analysis

The proposed scheduling algorithm assumes RAG analysis to delay execution of glue nodes not used for execution of the soonest ASR nodes. The algorithm concurrently schedules the initial program graph and its RAG. Classical list scheduling divides all nodes into three sets: already scheduled nodes, nodes which are ready for execution (with all predecessors scheduled) and nodes waiting for completion of their predecessors. In the original ETF heuristics, all ready nodes

are examined and one of them is chosen. Based on RAG analysis, we introduce two subsets of ready glue nodes: the high priority nodes needed for execution of the topologically nearest ASRs in the graph and the low priority nodes needed for execution of topologically more distant ASRs. The *topologically nearest* ASR nodes are such, which are also ready for execution in the RAG of the scheduled program graph. At every scheduling step, the glue nodes, which correspond to edges in the RAG leading to currently ready nodes in the RAG, have high priority, while other glue nodes have low priority. If an ASR node is scheduled, we also simulate its assignment to the same computing resources in the RAG of the scheduled graph. As a result of this assignment, the descendants of the scheduled node in the RAG may become ready – then all the low priority ready glue nodes on the ASR incoming edges obtain high priority.

The pseudo-code of the proposed scheduling algorithm is shown as Algorithm 1. It follows list scheduling principles. Each time, when a glue node is to be assigned to a GCMP, first the high priority nodes are considered. Low priority glue nodes are scheduled only when their execution doesn't impede high priority nodes. Such node selection strategy assumes that architectural nodes can be executed as soon as possible on ACMPs. Additionally, GCMPs if free, can execute glue nodes, which are required for further computations.

Time complexity of the algorithm remains polynomial, although with a higher degree than list scheduling with standard ETF heuristics. All the additional steps have polynomial complexity, including for instance computation of a RAG and layers using breadth first graph traversals as well as transfers of ready nodes between V_L and V_H sets in the loop.

4 Graph Metrics for Right Selection of the Scheduling Algorithm

We have compared make-spans obtained for different program graphs. Experiments show, that comparison results depend on features of the graphs in terms of topology, weights of nodes and edges but also on resources available for program execution. In our study we deal with layered program macro data flow graphs, built of node layers and edges for inter-node communication between layers. A layer in such program graph contains all architectural nodes, which have the same depth in the program RAG, plus all the glue nodes, which provide data for these architectural nodes.

In list scheduling of a program graph a node may be scheduled too early, which may lead to an un-optimal use of processor time for other ready nodes. We introduce a metrics, which we call **Cumulated Activation Stride (CAS)** of a program graph to measure the potential of the graph for this non-optimality. The metrics is determined starting with a traversal of a program graph by breadth first search, in which for every node a layer of a deepest architectural region activating this node is determined ($max\_act\_layer(v)$):

1. For each glue node v, determine its layer number, $layer(v)$ (they depend only on architectural nodes of the graph and their dependencies in RAG).

2. For each glue node v, determine $max\_act\_layer(v)$ – the maximal layer number in which this node may be activated, by computing the maximum over the following values, depending on all the predecessors u of node v:
   (a) If v has no predecessors, then $max\_act\_layer(v) = 0$.
   (b) If u is an architectural node, $max\_act\_layer(v) = layer(u) + 1$. It means, that node v should be treated in exactly the same way as glue nodes from layer $layer(u) + 1$, because it is activated within this layer.
   (c) If u is a glue node, then $max\_act\_layer(v) = max\_act\_layer(u)$. The node u can be activated earlier then needed, so we consider its max_act_layer, not its layer number.

After all the glue nodes in the graph are examined, we determine the **Activation Stride** for each glue node v:

$$activation\_stride(v) = layer(v) - max\_act\_layer(v)$$

This value will be non-zero only for nodes, which become ready before architectural nodes preceding their layer are completed.

The **Cumulated Activation Stride** metrics $CAS(G)$ for graph G is defined as the sum of node activation strides multiplied by node weights over all glue nodes, divided it by the product of sum of glue node weights and the maximal layer number in the graph ($Arch(G)$ and $Glue(G)$ correspond to architectural and glue nodes of the graph G, respectively):

$$CAS(G) = \frac{\sum_{v \in Glue(G)} activation\_stride(v) * weight(v)}{max_{u \in Arch(G)} layer(u) * \sum_{v \in Glue(G)} weight(v)}$$

So defined metrics will be equal to 0 if all the glue nodes are activated by architectural nodes, which precede their layers. The maximal value may be obtained for a graph, in which there are no glue nodes in all layers except the last one, and all these nodes are ready at the beginning of the program graph execution (they have no predecessors). Since the maximal stride cannot be greater than the maximum layer number, $CAS(G)$ cannot exceed 1. It also does not change if all the weights in the graph are multiplied by the same constant.

Figure 4 presents an exemplary program graph with layered structure. Each layer is composed as a set of uniform subgraphs containing nodes Aj^i, Bj^i and Mj^i. Nodes Aj^i and Bj^i are glue nodes, while nodes Mj^i are architectural nodes. The long, black edge corresponds to communication between layers, which activates node $B1^{i+2}$. The other activation edge of nodes Bj^i corresponds to read of initial data from shared memory. Node $B1^{i+2}$ in layer $i + 2$ is activated by architectural node from layer $i - 1$, therefore its activation stride equals 3 (layers are computed with respect to architectural nodes, not glue nodes).

5 Experimental Results

To evaluate and compare performance of the presented scheduling algorithm, the following exemplary iterative application program was considered:

```
func benchmark(stride) {
    // Let i be the iteration number
    for i=1 to N pardo
        // Let j be the path number
        for j = 1 to K pardo
            // select parts of the results of the previous iteration
            a[i, j] = A(m[i − 1, 1], m[i − 1, 2], ..., m[i − 1, N]);
            // if i ≤ stride then initial data are read
            b[i, j] = B(u[i, j], m[i − stride, j]);
            m[i, j] = M(a[i, j], b[i, j]);
        end for
    end for
}
```

This program corresponds to a computational algorithm, which includes common functions $A()$, $B()$ and $M()$ on elements of square matrices, such as matrix addition and multiplication. We assume, that functions $A()$ and $B()$ have irregular internal structure and are not promising for faster implementation in ACMP modules. Therefore they will be treated as glue nodes in the program graph. $M()$ is a parallel matrix multiplication based on recursive matrix decomposition into quarters. The stride parameter corresponds to the activation stride for computations of $B()$ functions which provide data for $M()$ regions. A vector $u[]$ and matrices $m[0, 1..N]$ are initial parameters of the program used for computation. A single iteration of the outer loop creates a layer of subgraphs (each subgraph corresponds to an iteration of the inner loop), which are mutually independent. Macro dataflow graph of a part of the considered exemplary program for $stride = 3$ is shown in Fig. 4.

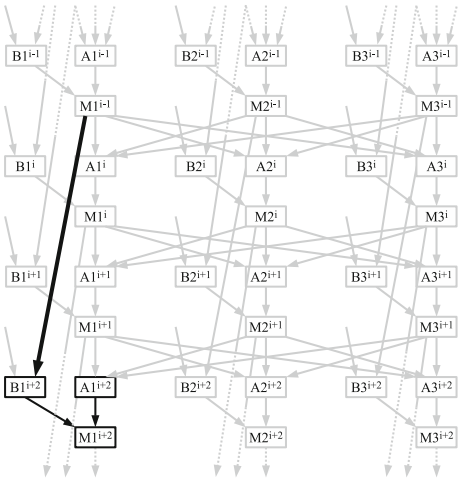


Fig. 4. Exemplary program graph with communication between layers that causes non-zero strides for nodes Bj^i; Mj^i – architectural matrix multiplication nodes.

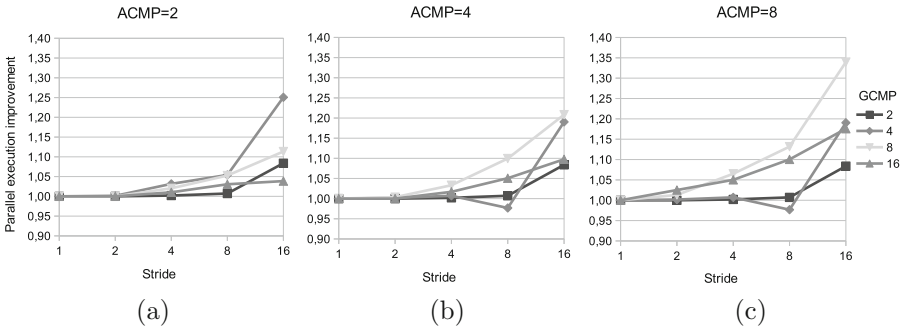


Fig. 5. Parallel execution improvement of the proposed algorithm with 2, 4 and 8 ACMPs and 2, 4, 8 or 16 GCMPs for graphs with activation strides 1, 2, 4, 8 and 16.

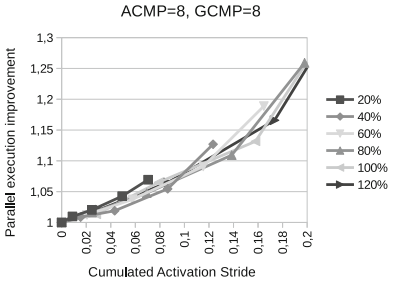
All graphs for experiments were generated with $K = 8$ (8 paths) and $N = 16$ (16 layers). For parallel execution of such graph, the maximal number of 8 ACMP and 16 GCMP modules is needed. We have considered a set of graphs for a range of values for parameter stride: 1 which corresponds to a graph with no strides, 2, 4, 8 and 16, which corresponds to a graph, in which all the B nodes are ready at the beginning of computations. The graph is uniform due to node weights, which were selected in the arbitrary way: all A and M nodes have weights equal to 8000 units, while B nodes have weights equal to 6500.

The graphs were scheduled for executive systems with a range of ACMP (2, 4 and 8) and GCMP (2, 4, 8 and 16) modules. ACMP modules execute only one ASR node at a time. GCMP modules were assumed to contain 1 computing core. We scheduled the graphs with a standard ETF-based scheduling algorithm and compared it to the schedules obtained with the proposed algorithm. We examined parallel schedule improvement computed as a ratio of execution time obtained by the reference algorithm to execution time of a graph scheduled with the proposed algorithm (Fig. 5).

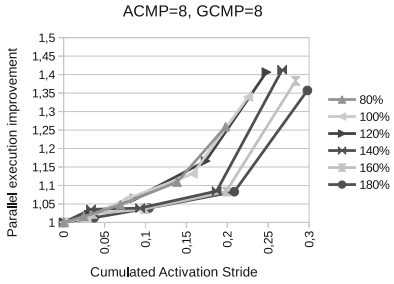
Experiments show that the proposed algorithm performs in general better than classical ETF-based list scheduling. The results depend on the number of both ACMP and GCMP modules applied. The biggest execution time improvement was equal to 1.34 for 8 ACMP and 8 GCMP modules and graphs with $stride = 16$. The smallest average improvement was obtained for the smallest (2) and the biggest (16) number of GCMP modules. This is due to the fact, that 2 GCMPs are insufficient to prepare data for 16 ACMPs on time (both algorithms are forced to serialize parallel computations), and 16 is the number of GCMPs needed for optimal execution of the graph, therefore the standard algorithm was capable of finding good schedule for such system. For some combinations of ACMP, GCMP and stride parameters one can observe improvements, which are smaller than 1. The parallel improvements are computed in comparison to ETF-based list scheduling algorithm, not the sequential execution. Therefore, it must be considered, how ETF deals with the graph for a given number of resources.

Table 1. Values of $CAS(G)$ for graphs generated for different values of the stride parameter and different relative weights of the B nodes.

Relative weight of B nodes	Activation Stride				
	1	2	4	8	16
20 %	0.000	0.009	0.025	0.049	0.070
40 %	0.000	0.015	0.043	0.086	0.123
60 %	0.000	0.021	0.058	0.115	0.165
80 %	0.000	0.025	0.069	0.139	0.198
100 %	0.000	0.028	0.079	0.158	0.226
120 %	0.000	0.031	0.087	0.173	0.248
140 %	0.000	0.033	0.093	0.187	0.267
160 %	0.000	0.035	0.099	0.199	0.284
180 %	0.000	0.037	0.104	0.209	0.298



(a)



(b)

Fig. 6. Parallel speedup improvement as a function of the $CAS(G)$ for different relative weights of the B-type nodes.

We assume, that for those configurations, ETF was able to find a solution, which was better than the one found by the proposed algorithm.

We have also examined the relation between the CAS metrics of the graph G and schedule improvement. In the assumed graph, the value of CAS depends on the stride parameter, which influences the topology of a graph, but also on weights of B nodes. We have checked a range of graphs, which differ in weights of B nodes. We have assumed a series of B node weights being a percentage of the B node weight (100 %) in the uniform graph used in the experiment discussed above. Experiments were done for the number of both ACMP and GCMP modules equal to 8. The results for other combinations of ACMP and GCMP numbers show similar tendencies to those shown in the paper. The values of CAS measures of examined graphs are shown in Table 1.

Figure 6 shows correspondance of the parallel improvement to the CAS metrics of graphs. Improvements for the graphs with CAS metrics equal to 0 ($stride = 1$) are the smallest in general and equal to 1. In such graphs all the glue

nodes are activated in their layers and therefore the standard ETF-based algorithm has no chance to make a wrong scheduling decision. With the increase of the CAS metrics we can observe a better improvement obtained by the proposed algorithm. Graphs with higher CAS contain more nodes that can be scheduled too early, when compared to their layer. Also, these nodes are heavier, therefore, they have bigger impact on the overall schedule. It makes such graphs harder to be correctly scheduled – especially with the standard ETF scheduling. Due to a different way of handling of ready nodes, the proposed algorithm shows much better resistance to such situations. Execution of the questionable nodes is delayed, which allows faster start of architectural nodes from previous layers and better resource use, leading to better schedules. The best improvements were noticed for graphs with the biggest CAS value ($stride = 16$).

6 Conclusions

The paper has presented parallel program scheduling algorithms for the modular system architecture based on globally interconnected standard and architecturally supported CMPs. The proposed scheduling algorithm is based on ETF heuristics improved by an analysis of the RAG. The additional analysis enables better use of both architectural and general purpose modules. It leads to better parallel speedups in the case of graph structures "difficult" for the standard ETF schedulers for adequate composition of the executive system.

The experiments with the proposed algorithm show, that it can deliver better schedules than standard ETF-based list algorithm. The experimental results have shown dependencies of the quality of obtained schedules on the proposed graph property metrics. Complexity of the standard ETF scheduling is smaller than complexity of the presented improved algorithm, therefore for graphs with small CAS values it is enough to use the standard ETF algorithm – the schedules are the same or very close to the schedules obtained with the improved algorithm, but they may be computed faster. For graphs with high values of CAS, it is profitable to use a better, although more complicated algorithm we propose.

References

1. Owens, J.D., et al.: Research challenges for on-chip interconnection networks. IEEE MICRO **27**, 96–108 (2007)
2. Kundu, S., Peh, L.S.: On-chip interconnects for multicores. IEEE MICRO **25**, 3–5 (2007)
3. Hwang, J.-J., Chow, Y.-C., Anger, F.D., Lee, C.-Y.: Scheduling precedence graphs in systems with interprocessor communication times. SIAM J. Comput. **18**(2), 244–257 (1989)
4. Yu-Kwong, K., Ishfaq, A.: Benchmarking and comparison of the task graph scheduling algorithms. J. Parallel Distrib. Comput. **59**, 381–422 (1999)
5. Sinnen, O.: Task Scheduling for Parallel Systems. Wiley, England (2007)

6. Błażewicz, J., Ecker, K.H., Pesch, E., Schmidt, G., Weglarz, J.: Handbook on Scheduling. International Handbooks on Information Systems. Springer, Heidelberg (2007)
7. Topcuoglu, H., Hariri, S., Min-You, W.: Performance-effective, low-complexity task scheduling for heterogeneous computing. IEEE Trans. Parallel, Distrib. Syst. 13(3), 260–274
8. Masko, Ł., Dutot, P.F., Mounié, G., Trystram, D., Tudruj, M.: Scheduling moldable tasks for dynamic SMP clusters in SoC technology. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 879–887. Springer, Heidelberg (2006)
9. Maśko, Ł., Tudruj, M.: Task scheduling for SoC-Based dynamic SMP clusters with communication on the fly. In: 7th International Symposium on Parallel and Distributed Computing, ISPDC, pp. 99–10. IEEE CS (2008)
10. Tudruj, M., Maśko, Ł.: Scheduling parallel programs based on architecture–supported regions. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part II. LNCS, vol. 7204, pp. 51–60. Springer, Heidelberg (2012)