computing@computingonline.net
www.computingonline.net

ISSN 1727-6209
International Journal of Computing

VECTOR CLOCK TRACING AND MODEL BASED PARTITIONING FOR DISTRIBUTED EMBEDDED SYSTEMS

Robert Hoettger, Burkhard Igel, Erik Kamsties

University of Applied Sciences and Arts
Pimes Research Institute
Fachhochschule Dortmund,
Sonnenstr. 96, 44139 Dortmund, Germany
robert.hoettger@fh-dortmund.de, igel@fh-dortmund.de, erik.kamsties@fh-dortmund.de

Abstract: Tracking, partitioning and tracing in modern dynamic high performance computing systems are three of the most innovative and important development aspects for performance optimization purposes and state-of-the-art advanced quality. This paper discusses these three aspects with respect to distributed systems and proposes new mechanisms for an advanced utilization of software in this domain.

We present a specific tracking mechanism via vector clocks for model and code partitioning purposes and the determination of causality relations. Further, a tracing approach for an effective analysis and thereby utilization of code and the corresponding architecture is introduced. The combination of both approaches leads to a high degree of parallelism and a fine-grained structure of execution units, that further traced, supports a precise analysis of synchronous and asynchronous system's behavior as well as an optimal load balancing. The mechanisms are introduced with respect to a model based control engineering tool and event diagrams.

Keywords: partitioning; event tracing; vector clocks; control engineering; distributed systems; virtual time.

1. INTRODUCTION

The modern digital computing era involves increasing amounts and relations of stored data as well as more complex computation platforms, architectures, tools and frameworks. A common, mandatory and important aspect is the prevention of computation- and storage overheads i.e. the efficient use of soft- and hardware. Especially the modern distributed system domain reveals a more complex determination of causality relations due to the use of replicas, unreliable hardware, large scales of data and commoditized machines. The increasing number of requirements, functions, safety issues or assistance demands call for a significant increase of computing power accompanied by the request for reduction of energy and costs. To handle these requirements the multicore processor technology starts to permeate electronic control units (ECUs) in cars for example. Existing applications cannot realize immediate benefit from these multicore ECUs, because they are not designed to run on such architectures.

The Itea 2 project 09013 *AMALTHEA*[1] is a state of the art research project in the automotive industry that addresses building a model-driven platform for this new generation of development environments, which supports the development of multicore systems, takes product line engineering into account and produces AUTOSAR [1] compatible software. Tracing and partitioning are two of the challenges to be met with respect to timing constraints. This paper presents a novel approach for both partitioning and tracing and further supports the determination of causality relations among events in a distributed system. Multicore systems in this case are one example for distributed systems.

Partitioning in context directed acyclic graphs that occur in most computing applications, influence system performance. The more efficient the partitioning process forms computation sets distributed among computation units i.e. processors, the more the systems benefits from time issues, energy demands or high performance real time applications. These aspects are common topics of interest in almost all areas of science and technology.

Forming computation sets mostly concerns the division of processes into subprocesses whereas each subprocess consists of computational load [2]. In terms of graph theory these subprocesses are

[1] Itea 2 09013 AMALTHEA, BMBF funded.

denoted as nodes. A node often reveals uni directed communication with one or multiple other nodes, such that a directed transition between them denotes dependency as shown in Fig. 1.

Fig. 1 – Node dependency.

Node B depends on a result of Node A and thereby depends on Node A. In case Node B is assigned to a different computation unit i.e. processor, the system must preserve the given ordering of both nodes. Otherwise Node B may be started without Node A being finished resulting in Node B termination violation. Such order may be preserved via inter process activation, client / server calls, OS-events (Node A (set) and Node B (wait)) or Semaphores.

The paper is organized as follows. The next section introduces related work on tracking, partitioning and tracing. Afterwards the concrete usage of vector clocks is described with respect to a model driven control engineering example, that is adapted to several different partitions and load balance approaches. The following section introduces tracing as a more comprehensive approach for performance and partitioning optimization purposes. Finally, the novel approach is analyzed according to benefits, ease of use and industrial relevance. Corresponding contents are published with respect to [3].

2. RELATED WORK

Innovation according to *virtual time*, *tracing* and *partitioning* stretches over years of development and huge amounts of different mechanisms and algorithms addressing the increasing number of requirements and constraints emerging from all kinds of political, entertainment, safety or energy demands.

Tracking, initially used in technical and theoretical computing, considers causality relations and the determination of event orderings in distributed systems with the help of logical clocks [4]. Extending this mechanism in order to gain information about the program's global state and possible concurrency, vector clocks can be used [5, 6]. Newer approaches focus on applying the exposed mechanisms to models, transferring models to mathematical equations [7] or introduce graphical editors, model checkers, code generators, simulators or dynamic systems and algorithms [8]. All these mechanisms basically address the derivation of timing characteristics for causality relations and thereby ensure logical and temporal correctness within communication, synchronization and computational flows actively by applying the certain mechanisms to a system.

Partitioning is a significant approach for an efficient assignment of runnables to tasks in order to utilize parallel computing. The generic PCAM (Partitioning, Communication, Agglomeration and Mapping) approach by Foster et al. [9] forms the basis for most common partition approaches. It focuses on providing benefits like improving cost-performance ratio, availability via avoiding redundancy, computing power and understanding of a program's behavior due to more detailed information about the problem structure. Partitioning is a division of independent parts in order to solve them in parallel. Therefore, small tasks must be defined, that utilize processors in an optimal way and avoid duplicate data and calculation. The smaller the partitions get, the more flexible and potential the parallelism is. Foster [9] further introduces domain decomposition and functional decomposition. In domain decomposition, data associated with a problem is divided into small parts with approximately equal size. Afterwards, computation is partitioned by associating the operations with the data on which it operates. The focus within functional decomposition lies on the computation that is to be performed instead of the data that is manipulated by the computation. The computation is divided into disjoint tasks, with a subsequent data requirement analysis. In case the data requirements are disjoint, the partition is complete, otherwise considerable communication is required to avoid data replication.

Tracing addresses revealing a program's execution according to more complex problems, errors, ineffective patterns and a lot more issues by considering way more parameters like architecture properties, scheduling paradigms, signals, runnables, processes, or threads and corresponding timing properties depending on the used trace format. Though, tracing only passively applies optimization and efficiency on a system, as specific trace format's APIs are used to generate trace files that can be read by specific tools. These tools mostly reveal the system's behavior in a timeline diagram and users are supposed to react and improve systems according to conflicts and ineffective patterns.

3. PARTITIONING

The following sections propose a novel approach for distributing execution units, emerging from both model elements i.e. data flow systems and vector clock traces as a result of specific code extensions.

The mechanism is based upon a transformation to a data flow graph (*directed acyclic graph*) and a subsequent partitioning for either a dynamic number or a fixed number of processes. The final result leads to an optimal utilization of parallel resources.

3.1. DATA FLOW PARALLELISM

Typical data flow systems provide a fine grained early degree of parallelism. Having a data flow system like Fig. 2, delay blocks encapsulate calculation dependencies providing distributed calculations due to their output being not directly dependent of their input.

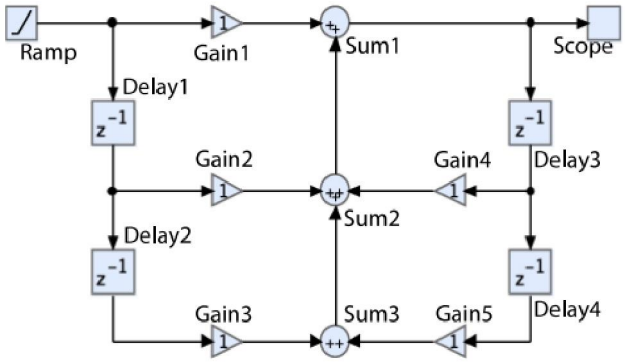

Fig. 2 – Data flow diagram.

The data flow diagram shown in Fig. 2 is derived via the frequency response:

$$H(z) = \frac{Gain1 + Gain2 \cdot z^{-1} + Gain3 \cdot z^{-2}}{1 + Gain4 \cdot z^{-1} + Gain5 \cdot z^{-2}} \quad (1)$$

In the first step i.e. calculation cycle, *Ramp* and *Delay1* to *Delay4* can be calculated in parallel by either different runnables, tasks or cores. The second step contains all blocks connected to the encapsulated blocks of the first step i.e. *Gain1* to *Gain5*. The fact that the subsequent components hold more than one input i.e. dependencies of previous components, *Sum3*, *Sum2*, *Sum1* and *Scope* must be executed subsequently after the first two calculation cycles (see *data flow graph* Fig. 3). Such calculation cycles may be also known as sequential code segments (SCS) [10].

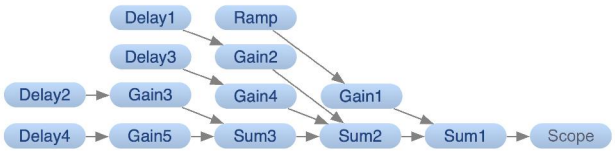

Fig. 3 – Typical DFG with nodes (blocks) and transitions.

3.2. DATA FLOW GRAPH

A data flow graph (DFG) displays nodes (execution units, algorithms, calculations, functions, events, blocks, etc.) connected to other nodes via transitions or edges sequentially (often from left to the right) and is convenient for the exploration of parallelism due to its asynchronous nature. It can be both created via modeled block diagrams (control engineering) or via vector clocks augmented code (described in Section III-C) and can be transformed to a table based structure revealing mandatory sequential orderings, dependencies (horizontally, indicated by arrows) and concurrency (vertically). Any DFG usually exposes a directed acyclic graph structure such that DFG(N,T,R,S) is defined by N nodes (execution units), T transitions, R root nodes and S sink nodes with no directed cycles. Any node $n \varepsilon N$ lies at least within one path from a root node $r \varepsilon R$ to a sink node $s \varepsilon S$. Any transition $t \varepsilon T$ between two nodes n_1 and n_2 represents data dependency between the two nodes and implies mandatory sequential ordering such that the execution of n_1 precedes n_2 in time. Hence n_1 and n_2 shall not be mapped to different processes or processing units as they can not be calculated in parallel.

A node n may possess multiple in- and out-transitions and transitions must always cross one or more sequential code segments (SCS). A low level DFG can be seen in Fig. 3 exposing the data flow model of Fig. 2, featuring R = (Delay1, Delay2, Delay3, Delay4, Ramp), S = (Scope), $\#T = 13$ and $\#N = 14$. Such DFGs can be automatically created from any block diagrams such as in the *Damos* environment [11]. A critical path leading from a root r to a sink s provides the maximal number of sequential nodes and represents the minimal runtime of a program. There may exist several critical paths in a DFG. One possible critical path in Fig. 3 starts at *Delay4* and ends at *Scope* (the other critical path in this example starts at *Delay2* and ends with *Scope*). Any usual control engineering based blockdiagram can be transformed into a DFG via forming SCSs with the help of delay calculation encapsulation and depth first search calculations. A DFG is mandatory for an optimal partitioning, respectively efficient utilization of distributed resources as described in Section III-D. The process of finding the critical path starts with identifying the root and checking all dependent nodes, whether one or more nodes provide the distance of the root 1 to the farthest sink. Afterwards for all selected nodes that provide that distance, the process is repeated regarding the selected node's sink distance 1 until the sink is found. This methodology identifies at least one

critical path via a helping function, that calculates a node's distance to the farthest sink.

3.3. VECTOR CLOCK AUGMENTED EXECUTABLE CODE

Any program code can be partitioned to one or more processes or initially execution units featuring dependencies. Most common programs use message passing techniques and data transfer between functions, objects and similar execution units for communication. In case such a program is not related to a modeled system (like described in Section 3.1), one can extend any program's code, adding vector clock API calls at specific points for both creating vector clock traces and use validation mechanisms for tracking data updates and determine causal dependency relations among transactions. Data updates thereby support synchronizing events in a totally decentralized way. Especially modern transactional systems using partial replication and scalable distributed multiversioning such as NoSQL data grids like BigTable, Amazon Dynamo or Cassandra require multiversion based update mechanisms [12]. A simple vector clock trace in combination with the executable code can be used in order to create an unpartitioned DFG on the one hand as well as a partitioned message passing based event diagram on the other hand. Such an event diagram is shown in Fig. 4 as an example, featuring three processes and several communicating events.

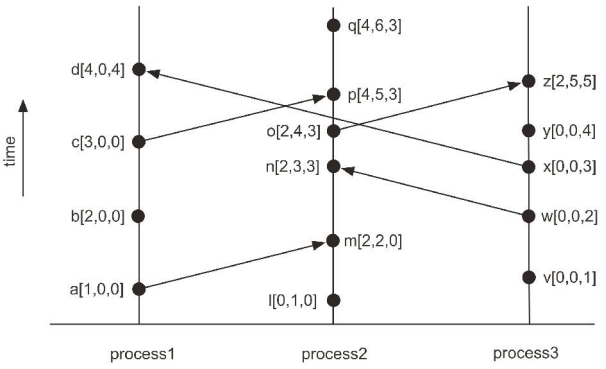

Fig. 4 – Typical event diagram with three processes.

It is assumed, that processes communicate through message passing in a classic asynchronous way such that messages consume a specific delay. The three processes show parallelism, whereas the determination of causality relations (provided by the vector clocks) respectively the knowledge of the precise time related occurrence of events and their communication is mandatory to avoid conflicts and preserve the program's semantics. The mentioned mechanism for causality relation determination of events was introduced by [5] and [6] simultaneously via comparing vector clocks using the following rules:

$$e_1 \rightarrow e_2 \; means \; C_{e1}[p] < C_{e2}[q] \tag{2}$$

and vice versa:

$$C_{e1}[p] < C_{e2}[q] \; means \; e_1 \rightarrow e_2 \tag{3}$$

Here, e_1 and e_2 define two different events with corresponding vector clock arrays C_{e1} and C_{e2}, $\rightarrow$ defines the "happened before" relation and p, q define two transactional processes.

A generated vector clock trace already references a specific number of processes and can be used in order to assign the code segments to different processes i.e. to perform the partitioning. A vector clock trace can be extended as described in Section 4. Without a vector clock trace, the described DFG is a central activity for partitioning and load balancing. A DFG can be created via identifying nodes (execution units) and transitions (dependencies).

In order to gain a partitioned event diagram (see Fig. 4) from an unpartitioned DFG (see Fig. 3), all execution units need to be assigned to processes. This can be dynamically performed assuming a static predefined number of processes. Each API call then assigns the execution units chronologically to a process with respect to their communication. Assuming execution unit a with transactions to *b* and *m* (Fig. 4), *m* could be assigned to process2 or process3. Process2 is chosen if execution unit *l* at process2 finished. If *l* is not finished, process3 is chosen if execution unit *v* finished. If both processes are performing calculations (execution units *l* and *v*) at execution unit *m* assignment time, the event will be assigned to the process, that notifies its availability first. The vector clock mechanism thereby ensures the correct replication of the actual behavior i.e. the call sequence in a distributed system. This mechanism is important especially for distributed systems consisting of multiple commoditized systems, meeting the necessity of a global time for causal ordering determination e.g. managing consistency in the Amazon Dynamo architecture [13]. The actual DFG for the event diagram in Fig. 4 is shown in Fig. 5. The DFG reveals an optimal parallelism of three processes, providing a complete system calculation consisting of 15 nodes in six SCSs (steps) and five cross process communications indicated by dashed arrows.

The proposed executable code trace generation extension provides causality relation determination as well as the DFG- and event diagram partitioning

approaches for an efficient parallelism support via trace analysis.

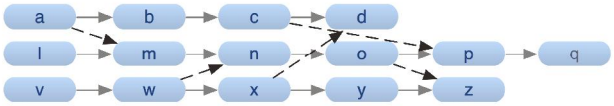

Fig. 5 – Event diagram correspondent DFG.

3.4. DFG PARTITIONING

Besides the use of the vector clock API call augmented executable code for program partitioning and distribution, data flow systems can utilize similar mechanisms in order take advantage of parallelism. Data flow diagrams (Fig. 2) can be transformed to data flow graphs (DFG, Fig. 3) as described in Section 3.2 in order to apply a specific partitioning mechanism for assigning nodes to runnables or processes. The DFG's number of SCSs defines the minimal number of steps (sequential executions) and the number of rows defines the maximal number of processes, whereas the number of occupied rows varies from SCS to SCS and the maximal process number refers to the SCS with the maximal row count. Fig. 6 displays the event diagram with regard to the data flow example shown in Section 3.1 i.e. Fig. 2 and Fig. 3 partitioned to four processes. Here, the maximum number of sequential nodes is bound to process1 by six nodes.

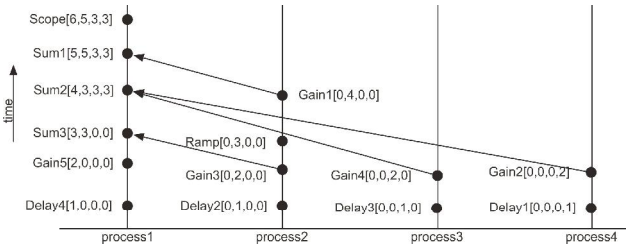

Fig. 6 – Data flow correspondent event diagram with four processes.

Fig. 7 displays the same program, but mapped into three processes. Here, the maximum number of SCS is increased to seven due to the fact that process3 can not calculate *Delay1* at the third SCS because of *Delay1*'s adjacent nodes to the sink. The partitioning algorithm always assigns nodes to a process according to the SCS and the node's adjacent nodes to the sink. Assigning nodes to process3, the algorithm only detects *Gain2* for SCS three (fourth last SCS to sink), due to *Delay1* (being the only unassigned node besides *Gain2*) revealing four adjacent nodes to sink and only nodes with maximal three adjacent nodes to the sink are considered. In this case the partitioning algorithm stretches processes in order to assign the unassigned nodes, i.e. inserting a new SCS at the corresponding SCS step beginning with with first process that is a new SCS preliminary to Sum3 in the example shown in 7.

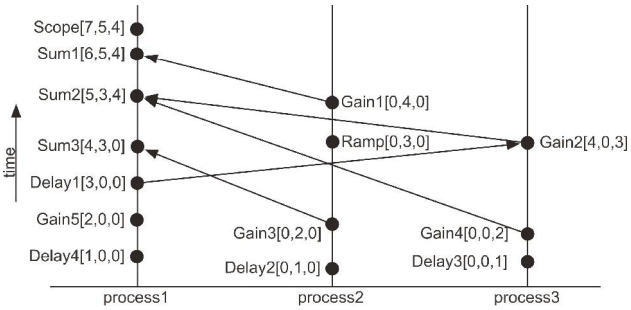

Fig. 7 – Data flow correspondent event diagram with three processes.

Having the nodes assigned to two processes, the result looks like Fig. 8. Here, the maximum number of SCS rises to eight due to two stretch operations caused by *Delay1* and *Gain2*.

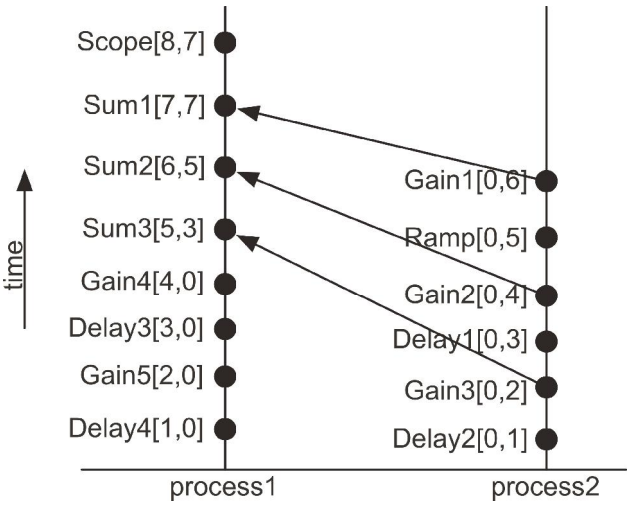

Fig. 8 – Data flow correspondent event diagram with two processes.

The example shows, that a critical path described in 3.2 with exactly one node from each SCS connected via transitions initially forms the first process. The partitioning algorithm is supposed to determine the critical path, that provides the least cost instensive calculation, for forming the first process. Hence it uses multiple optimization criteria i.e. minimal runtime, minimal cross process communication and the precise parallelism degree (number of processes constraint). In order to form additional processes, already assigned events are ignored and the farthest node from a sink is identified and assigned to another process via a depth first search (DFS). According to the following node assignments to each SCS at a process, firstly adjacent nodes (that provide a transition to the

preceding assigned node) and secondly any other nodes according to the specific SCS are considered. In case neither a adjacent nor a node for the specific SCS can be found, nodes from subsequent SCSs are taken into account. In case the number of processes is restricted by the user and the mechanism obtains unassigned nodes, the partitioning algorithm is able to stretch processes and insert unassigned nodes at specific SCS steps in order to finally assign all nodes. These assignment are processed with regard to the node's dependencies and execution cycles, such that no order constraint is violated.

The previously described methodology has been implemented in an approach called local graph partitioning (LGP). The basic idea of LGP is the assumption of at least one critical path within a directed acyclic graph. This path represents a sequential ordering that does not benefit from being distributed or calculated in parallel due to each node depending on previously calculated results. Mapping such a critical path to different calculation units would result in an increased calculation time due to overheads produced by synchronization and communication between the calculation units. Consequently the critical path is assigned to the first ProcessPrototype and all side paths, branches, sources and sinks of the graph are calculated parallel to that critical path in other ProcessPrototypes. The amount of ProcessPrototypes, respectively the number of actual parallel calculations, can either be maximized automatically by the implementation or specifically defined by the user. The partitioning is able to identify the maximal number of nodes to be calculated in parallel and creates ProcessPrototypes correspondingly. Furthermore, in case the user defines a specific number of threads, the partitioning is able to stretch threads by inserting nodes at a specific time slice between already assigned nodes according to their distances to the farthest sink in order to meet the user's thread constraint. The LGP mechanism shall be outlined by the following pseudo code.

Initially, in lines 1 and 2, two sets are built, containing unassigned nodes and all tasks. Afterwards the critical path is determined, assigned to the first task and all critical path nodes are removed from the list containing the unassigned nodes (*U*). The subsequent for loop (line 4) performs the node to ProcessPrototype (task) assignment such that other ProcessPrototypes contain graph branches beginning with the runnable (node) that provides the greatest distance to the critical path's sink. In case the number of ProcessPrototypes has been automatically calculated, this assignment will cover all occurring runnables. The second for loop (line 8) assigns a node from the list that contains the unassigned nodes (*U*) to each time slice parallel to the critical path with respect to not violating any order constraint. The subsequent while loop (line 24) performs the node insertion process, that is activated in case the user restricted the number of tasks to a smaller value compared with the automatically generated value. In other words, the loop will only be executed in case there remain unassigned nodes after the prior node to task assignment. The user's task number restriction causes each task to execute more nodes such that the overall execution time will be greater than the critical path's execution time. This *stretching* (execution time increase) is defined by the *stepincrease* value (see Listing 1 lines 25-28). The *stepincrease* value is calculated by the number of unassigned nodes divided by the number of tasks and incremented in case the division did not result in an integer value. This ensures that all unassigned nodes can be evenly distributed among the tasks. E.g. if there are three tasks and five unassigned nodes, the tasks one and two will be extended by two nodes (5=3 = 1+1 = 2) and task three by one node.

```
1  Let U denote the set of all unassigned nodes
2  Let T denote the set of tasks
3  Determine the critical path CP and its length CPL (number of
       nodes) and remove the CP nodes from U
4  FOR each task t in T
5    Determine the farthest node to the next sink fn in U
6    Let NTSi denote fn's distance to the farest sink (critical
         path sink)
7    Let NTSo denote fn's distance to the next source (critical
         path source)
8    FOR each time slice s of CP in t beginning with s=CPL
9      IF NTSi == s
10       select fn
11     ELSE IF there are multiple farthest nodes
12       Let fns denote all nodes in U providing the farthest
             distance to sink
13       select fn of fns with highest NTSo
14     ELSE IF there is exactly one farthest node && NTSo <=
           CPL-s+1
15       select fn
16     ELSE no node fits into current time slice --> empty slot
17     ENDIF
18     Assign selected node to t
19     remove selected node from U
20     set fn to either a connected node providing a distance to
           sink = fnd-1 or to the node providing the farthest
           sink distance
21   ENDFOR
22 ENDFOR

24 WHILE U is not empty
25   set stepIncrease to U.size / T.size
26   IF U.size modulo T.size not equals 0
27     increase stepIncrease by one
28   ENDIF
29   FOR each stepIncrease
30     FOR each task t
31       IF U is not empty
32         Determine the farthest node to the next sink fn in U
33         Let fnd denote the distance to the next sink of fn
34         assign fn to t after fnd
35         remove fn from U
36       ENDIF
37     ENDFOR
38   ENDFOR
39 ENDWHILE
```

Listing 1. Pseudocode for partitioning algorithm

A feature that is not mentioned in the pseudo code is the recognition of CPC (Cross Process Communication). For instance if a runnable A provides a RunnablePrecedence to a runnable *B* and the runnables are assigned to different ProcessPrototypes, specific model elements have to be created as described in the introduction (section I) i.e. synchronization events and sequencing constraints.

Finally the LGP aprroach provides all system's runnables distributed among several ProcessPrototypes (user defined or automatically generated) as well as explicit CPC model elements, that can be combined and transmitted to a mapping plugin [14] for further information augmentation and finally to a code generator in order to apply the partitioning to the software and run it in parallel on a multicore system. The partitioning mechanism processes runnables (nodes) with respect to their dependencies (orderings) and execution cycles and utilizes multicore architectures by efficient parallelism and load balancing such that execution times and energy consumption can be lowered and high performance application development can be facilitated.

3.5. PARTITIONING EVALUATION

In [10] two several similar approaches to DFG partitioning are introduced with respect to node's earliest and latest initial times respectively node's runtime or calculation cycles. However the particular number of processes constraint is not considered i.e. merging untreated nodes into existing processes. Due to strictly forming and not changing the critical path, the partitioning in [10] is ineffective according to calculation intense multiple propagated nodes in combination with a process amount constraint. Hence the proposed DFG partitioning in this paper benefits from parallel constraint consideration. Fig. 9 shows *a)* a DFG and correspondingly in *b)* a pipeline partitioning, in *c)* the partitioning from [10] and in *d)* the proposed partitioning of this paper for two processes (indicated by the lower row as process1 and the upper row as process2). The dashed arrows indicate process wide communication. The pipeline partitioning features most cross process communication due to not considering any dependencies. The c) partitioning features the critical path in process1 but increases the overall SCSs due to not being capable of stretching a process. The presented partitioning approach of this paper is shown in d) and provides both a low SCSs amount as well as low cross process communication. The presented partitioning reveals a simple structure whereas industrial applications feature much bigger DFGs such that the partitioning provides more significant benefits for parallelism.

Several literature emphasizes on reducing communication overhead like the region partitioning approach [15] or min-cut partitions [16] or distinguishing between control-, data- and dependence transitions [17] via specific complex mechanisms whereas the presented approach of this paper focuses on simplicity and an efficient automatic load balancing for practical issues in early development phases.

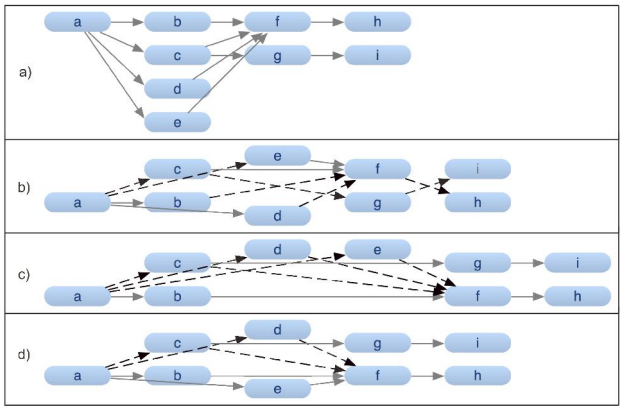

Fig. 9 – Comparison of DFG partitioning.

4. PERFORMANCE OPTIMIZATION THROUGH TRACING

The next step after the efficient and causal correct partitioning, is addressing and revealing more convoluted problems like race conditions, errors, ineffective patterns and dynamic behaviors by considering system parameters like architecture properties, scheduling paradigms, signals, runnables, processes or threads and corresponding timing properties. This can be handled by storing relevant system activities during execution of a physical or simulated system by using a extended trace API, that specifies the trace format. An additional program is supposed to read and analyze the trace data, that has been recorded during the system's execution.

The *AMALTHEA* project takes advantage of two major types of evaluation applied to the trace data, the metric calculation and the Gantt visualization. The metric calculation determines response times of a task or the duration of event chains for instance. All these metrics have in common that certain actions of specific entities are required for calculation. The response time of a task for example requires the actions activated and terminated from the related task. Another example is an event chain which consists of a write access of a task *A* and a read access of task *B*. This case requires the collection of both tasks' related events.

The evaluation types require both dynamic system behavior exploration and dynamic system behavior comparison. The dynamic system behavior exploration allows the determination of system characteristics during execution of the system by tracing system environment or system parts interaction. It provides system behavior, resource consumption, safety related activities and the generation of the system's model. The dynamic system behavior comparison provides quantifying

differences between modeled and physical systems at different development phases (architectural design, functional design, implementation, verification) in order to improve the abstract modeled system.

A vector clock extended trace intensely improves handling simulation, inferential performance and error analysis due to abstraction via virtual time and no need of global clocks respectively the timestamp. Furthermore, tracing execution time for software elements facilitates the partitioning activity by detecting relevant execution time differences for the same SCS. Knowledge about execution times for all nodes improves the load balancing via assigning appropriate nodes to empty time slots at different SCSs (before a cross process synchronization or data exchange for instance). Besides vector clocks the new trace approach features various information like the timestamp, its resolution, a configuration section for comments, optional parameters, creator and format version, the event type, a trace merge field, a memory access field, a memory protection usage field, the recording's precision, the unique event identifier, an instance field and provides pre-defining the data set to be logged.

Having all this information, one can gain absolute knowledge about a system's execution respectively use key information in order to evaluate, improve, and optimize a system. This especially concerns performance analysis according to preserving the temporal and spacial relationships of events, gaining information about using limited resources more efficiently or increasing scalability for bigger simulations. Vector clocks in this context facilitate causality relation determination and constitute a way of replacing expensive timer modules as well as combined with the described trace data, provide the detection of inadequate states during runtime in contrast to debugging, that stops the system at specific breakpoints.

5. CONCLUSION

The proposed new partitioning mechanism combined with both the tracking and the tracing approach, provide a fine grained parallelism bound to a causal correct and optimized software development, focusing on efficiency and optimized performance for modern distributed systems.

The novel tracing approach helps users to reveal errors, problems and conflicts, improve system's performance, utilize limited resources more efficiently and facilitate development processes in a wide field of software domains. Compared to most commonly used trace formats, the described approach meets modern demands, constraints and requirements of distributed systems according to hard- and software issues such as memory accesses, cores, frequency or semaphores and timing metrics.

The comparison of various partitioning approaches reveals, that the proposed mechanism is capable of constraints and still preserves minimal runtime (number of SCSs) and minimal cross process communication in order to utilize parallel resources optimally. Various adaptions such as communication and computation cycle handling influences the mechanism and enables minimizing synchronization costs or waiting periods.

Applying the promising concepts to a program, the user benefits from an automatic partitioning and the assignment of execution units to any number of processes resulting in an optimal load balancing across processes and a trace, providing all necessary information for commonly used analysis or evaluation tools.

ACKNOWLEDGEMENT

The authors would like to thank Robert Preis for fruitful discussions according to the partitioning mechanism as well as the Amalthea consortium for a great exchange of different experience, knowledge and expertise focus.

6. REFERENCES

[1] Autosar – automotive open system architecture, January 2013, http://www.autosar.org.

[2] R. Preis, *Analysis and Design of Efficient Graph Partitioning Methods*, Ph.D. dissertation, University Paderborn, 2000.

[3] R. Hoettger, B. Igel, and E. Kamsties, A novel partitioning and tracing approach for distributed systems based on vector clocks, *Proceedings of the IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems, IDAACS'2013*, Berlin, 12-14 September 2013, pp. 670–675.

[4] L. Lamport, Time, clocks and the ordering of events in a distributed system, *Communications of the ACM*, (21) 7 (1978), pp. 558–565.

[5] F. Mattern, *Virtual time and global states of distributed systems, Parallel and Distributed Algorithms*, M. Cosnard et al. (editors), North-Holland, 1989, pp. 215–226.

[6] C. J. Fidge, Timestamps in message-passing systems that preserve the partial ordering, *Proceedings of the 11th Australian Computer Science Conference*, 1988, Vol. 10, pp. 56–66.

[7] A. Benveniste and G. Berry, The synchronous approach to reactive and real-time systems, *Proceedings of the IEEE*, (79) 9 (1991), pp. 1270-1282.

[8] Paulo Sérgio Almeida, Carlos Baquero, Victor Fonte, Interval tree clocks: A logical clock for

dynamic systems, *Proceedings of the 12th International Conference on Principles of Distributed Systems OPODIS'08*, Luxor, Egypt, *Lecture Notes in Computer Science*, Vol. 5401, 2008, pp. 259-274.

[9] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison Wesley, 1995.

[10] A. Nadgir and H. Haridas, Data flow partitioning schemes, 2003.

[11] Dataflow-oriented modeling with damos, March 2013, http://www.eclipse.org/proposals/tools.damos/.

[12] S. Peluso, P. Romano, F. Quagila, and L. Rodrigues, When scalability meets consistency: Genuine update-serializable partial replication, *Proceedings of the IEEE International Conference on Distributed Computing Systems*, 2012, pp. 455–464.

[13] G. DeCandia, D. Hastorun, M. Jampani, et al., Dynamo: Amazon's highly available key-value store, *Proceedings of twenty-first ACM SIGOPS symposium on Operating Systems Principles*, 2007, pp. 205–220.

[14] L. Krawczyk and E. Kamsties, Hardware models for automated partitioning and mapping in multi-core systems, *Proceedings of the IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems, IDAACS'2013*, Berlin, 12-14 September 2013, Vol. 2, pp. 721–726.

[15] Y. Fong Lee, B. G. Ryder, and M. E. Fiuczynski, Region analysis: A parallel elimination method for data flow analysis, *IEEE Transactions on Software Engineering*, (21) 11 (1995), pp. 913–926.

[16] V. Elling and K. Schwan, Min-cut methods for mapping dataflow graphs, *Proceedings of the 5th International Euro-Par Conference on Parallel Processing, ser. Euro-Par'99*. Springer-Verlag, London, UK, 1999, pp. 203–212, http://dl.acm.org/citation.cfm?id=646664.701045.

[17] K. E. Schauser, D. E. Culler, and T. V. Eicken, Compiler controlled multithreading for lenient parallel languages, in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge*, MA, USA, August 26–30, 1991, *Lecture Notes in Computer Science*, Vol. 523, 1991, pp. 50–72.

Robert Hoettger *received his B. Eng degree in 2011 in information technology and electrical engineering from University of Applied Sciences and Arts, Dortmund in Germany. He is a Master student and is going to start his Ph D studies in 2014. His research area covers distributed and parallel computing and model based software engineering with regard to automotive applications.*

Burkhard Igel *after study in electrical engineering and computer science received his doctoral degree (Ph.D.) in computer science from University of Dortmund. After more than 15 years working for Siemens Corporation now he is Professor at University of Applied Science and Arts in Dortmund and chairman of the supervisory board of item is AG. His research area covers distributed and parallel computing as well as requirements engineering and model based design.*

Erik Kamsties *is a professor for software engineering and embedded systems at the Dortmund University of Applied Science and Arts. He received a Diploma (M.S.) from the Technical University of Berlin and a doctoral degree (Ph.D.) in computer science from the University of Kaiserslautern (Germany). His current research focuses on requirements engineering, model-driven development, and embedded multi-core systems.*