A Fast Rescheduling Heuristic of SDF Graphs for HW/SW Partitioning Algorithms

B. Knerr, M. Holzer and M. Rupp

Vienna University of Technology
Institute for Communications and RF Engineering
Gusshausstr. 25/389, 1040 Vienna, Austria
{bknerr, mholzer, mrupp}@nt.tuwien.ac.at

Abstract

HW/SW partitioning of modern heterogeneous systems, which combine signal processing as well as multimedia applications, is usually performed on a task or process graph representation. As this optimisation problem is known to be NP-hard, existing partitioning techniques rely on heuristic methods to traverse the vast search space. Moreover the process scheduling of a synchronous data flow (SDF) graph on distributed resources, which constitutes the evaluation of every single *partitioning solution, is also an NP-hard problem. This paper proposes a fast rescheduling technique of SDF graphs suitable for HW/SW partitioning algorithms that move incrementally through the search space. Its performance is demonstrated by a comparison to classical list scheduling algorithms for distributed resources.*

Keywords: Task Scheduling, HW/SW Partitioning, Synchronous Data Flow Graphs, Multi-Rate Systems, Fast Rescheduling

I. Introduction

Modern system design, especially in the wireless domain, has to face hard challenges with respect to chip area, power consumption, and computation intensive signal processing as well as multimedia applications, while time-to-market is critical. The diversity of the requirements has lead to extremely heterogeneous system architectures that use both application-specific hardware accelerators (HA) and general-purpose, programmable units with the appropriate software [1]. Dedicated hardware units usually have higher throughput and are far more power efficient, but are on the other hand much more expensive to design and inflexible. Contrarily, software provides flexibility and is cheaper to maintain, whereas the required general-purpose processors are power consuming and slower. The optimal trade-off between cost, power and performance has to be identified. The short design cycles in the wireless domain boosted the demand for very early design decisions, such as architecture selection and hardware/software partitioning on the highest abstraction level, i.e. the algorithmic description of the system. This task used to be carried out manually by designers recalling their knowledge from former products and estimating the effects caused by their decision. The rampantly growing complexity made this approach unfeasible and forced research groups to concentrate their efforts on automating the hardware/software partitioning as much as possible.

Hardware/software partitioning can in general be described as the mapping of the interconnected functional objects that constitute the behavioural model of the system onto a chosen architecture model. The task of partitioning has been thoroughly researched and enhanced during the last 15 years and produced a number of feasible solutions, which depend heavily on their prerequisites: the architecture model, the communication model, the granularity of the functional objects, etc. The partitioning problem leads to an exponentionally growing search space, even if one neglects all subtleties brought in by the chosen architecture abstraction. Therefore, the majority of the approaches are based on heuristics. Additionally, even the simplest form of the assumed target platform has at least one component, the general-purpose processor or DSP, which can execute the assigned functional objects *sequentially*. In other words, the partitioning problem embeds in its core another hard problem, resource allocation, also referred to as multi-core scheduling. Powerful approaches for multi-core scheduling do exist, but are far too complex to be applied in the middle of partitioning engines. Therefore, the majority of the partitioning methods that tackle the inherent scheduling issue simultaneously rely on fast list scheduling techniques. In this work we present a scheduling algorithm that is appropriate to be applied at the core of partitioning heuristics,

which perform *one move* steps through the search space, i.e. a new solution is created from the preceding solution by changing the implementation type, from hardware to software or vice versa, of only one functional object at a time. By reusing the old schedule, it is possible to generate a new schedule in linear computation time, that optimises locally the exploitation of parallelism within the interconnected functional objects. The performance of this approach is demonstrated on typical SDF graph structures.

The rest of the paper is organised as follows. The next section sheds some light on related work in the field covering combined partitioning/scheduling techniques. Section III illustrates the system abstraction into multi-rate systems, the target platform model based on a realistic SoC design in the wireless domain, and the problem formulation of hardware/software partitioning in this context. It is followed by a detailed description of the rescheduling algorithm in Section IV. Results of the comparison with other list scheduling techniques are given in Section V. In Section VI the paper concludes and gives perspective to future work.

II. Related Work

Heuristic approaches dominate the field of partitioning algorithms, since partitioning is known to be an NP-hard problem in most formulations [2], [3]. Genetic algorithms have been extensively used [4], [5] as well as simulated annealing [6], [7]. To a smaller degree tabu search [8] and greedy algorithms [9] have also been applied. Other research groups developed custom heuristics such as the GCLP algorithm in [10] or the early work in [11].

Formulations of the inherent multi-processor scheduling problem have been shown to be NP-complete [12], [13]. The vital importance of this task caught permanent attention of research groups over the last 20 years. To name only a few with low complexity: Hu's level based scheduling [14], the dynamic level scheduling approach of Sih [15], the edge-zeroing algorithm [16], which focuses on communication costs, and the mobility directed algorithm [17].

With respect to combined partitioning/scheduling approaches, the work in [18], [19] has to be mentioned. The approaches in [5], [20] also add communication events to links between hardware and software units. The architecture model varies from having a single software and a single hardware unit [6], [9], which might be reconfigurable [18], to a limited set of different hardware units in combination with a general-purpose processor [21].

Approaches showing the *one move* step behaviour of the algorithm through the search space can be found in [6], [8], [21], [22].

III. Prerequisites

A. Signal Processing Systems as SDF Graphs

The graph representation of the system is generally given in the form of a task graph, which is usually assumed to be a directed acyclic graph (DAG) describing the dependencies between the components of the system. We based our work on the a graph representation for multi-rate systems, also known as synchronous data flow (SDF) graphs [23], that was introduced in 1987. This representation established the backbone of renowned signal processing work suites, e.g. Ptolemy [24] or SPW [25]. It captures precisely multiple invocations of processes and their data dependencies and thus is most suitable to serve as a system model in which process computation time, as well as interprocess communication time, is considered.

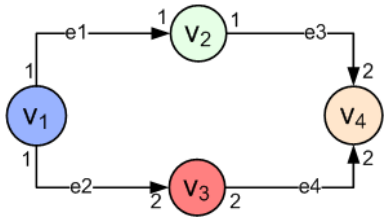

Fig. 1. *Simple example of a synchronous data flow graph.*

In Fig. 1, a simple example of an SDF graph $G = (V, E)$ is depicted, composed of a set of vertices $V = \{v_1, .., v_4\}$ and a set of edges $E = \{e_1, .., e_4\}$. The numbers on the tail of each edge e_i represent the number of samples produced per invocation of the vertex at the edge's tail. The numbers on the head of each edge indicate the number of samples consumed per invocation of the vertex at the edge's head. The decomposition of the system under investigation into an SDF graph depends heavily on the chosen level of granularity. We assume a vertex to be a sequential code block, also referred to as process, like a distinct matched filter function (e.g. DCT, FIR), or a sorting algorithm (e.g. shellsort), as opposed to be a single operational unit (e.g. a MAC or ALU). This granularity enables the identification of whole code blocks that may be implemented as a dedicated hardware accelerator or run as software on a DSP or general-purpose processor.

B. Model of the Hardware Platform

The target architecture model (see Fig. 2) has been chosen to provide a maximum degree of generality, while correspoding to the industry-designed SoCs in use. The platform has been abstracted from a UMTS baseband

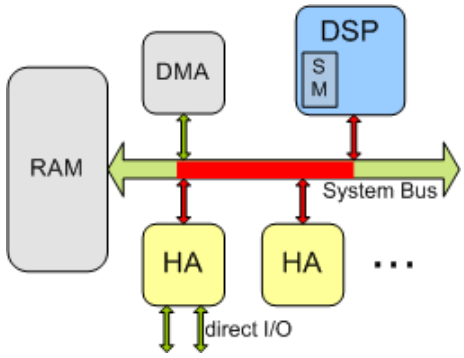

Fig. 2. *General model of the target architecture.*

receiver chip: it consists, in its least complex layout, of one DSP handling the control oriented parts of the design, several hardware accelerator (HA) units for the data oriented and computation intensive signal processing, one system bus, and a system RAM with a DMA. The DSP posesses its own internal shared memory (SM) to enable a fast internal communication between processes. The HAs may have direct I/O ports for off-chip communication, e.g. the antenna subsystem.

C. HW/SW Partitioning Process

Hardware/software partitioning of the graph representation is meant to identify the mapping of the vertices either to a software module running on the DSP or to a dedicated HA interfaced to the system bus, while meeting timing constraints, and minimising chip size and power consumption. The majority of the partitioning approaches assigns a set of characteristic values to every vertex:

$$\forall v_i \in V \; \exists \; S_i^v = \{pct_h, pct_s, gc_h, \dots\} \qquad (1)$$

Here pct_h is the process computation time as a hardware unit, pct_s is the process computation time of the software implementation, gc_h is the gate count for the hardware unit, and others like power consumption etc. From those values, mostly obtained by high level synthesis or estimation techniques like static code analysis or profiling, a cost function is assembled to evaluate every single partitioning solution. Without exception, all of the methods mentioned in Section II integrate the overall system execution time in their cost evaluation of the partitioning. For the following considerations regarding the scheduling only pct_h and pct_s are of interest.

$$\forall e_i \in E \; \exists \; S_i^e = \{ct_{sh}, ct_{bus}\} \qquad (2)$$

Regarding interprocess communication, every edge gets also assigned a set of characteristic values. In our case we distinguish between two types of communication: ct_{sm} is the communication time for sw to sw via shared memory and ct_{bus} is the communication time for sw to hw or hw to hw via the system bus.

Before the mapping takes place, the SDF graph has to be decomposed into its single activation graph (SAG) representation, i.e. the input/output rate dependencies are solved and transformed into another graph representation, in which every vertex is just activated once per system run (see Fig. 3).

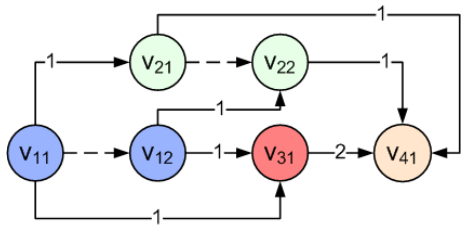

Fig. 3. *Single activation graph of the SDF graph in Fig. 1: every vertex corresponds to a single process invocation.*

The vertices v_1 and v_2 are doubled according to their distinct invocations that result from the data rate analysis. The solid edges indicate precedence as well as data transfers from one vertex to another, whereas the dashed edges indicate precedence only. The data rates at the edges, i.e. interprocess communication, have been replaced by the number of data samples transmitted via each edge per invocation of the source vertex, as they have impact on the communication times ct_{sm} and ct_{bus}, respectively.

A certain partitioning solution is characterised by a unique mapping of the four process vertices to either hardware or software, which results in a search space of $2^{|V|}$, with $|V| = 4$ vertices in Fig. 1. Now assume the vertex v_1 is a software module and $v_{2..4}$ are hardware units. When the partitioning engine tries to move v_2 to the DSP as well, two valid schedules exist, as depicted in Fig. 4. However, for complex graphs the number of valid schedules corresponds to the number of topological sort layouts, which increases geometrically. In this simple case with only two valid schedules, schedule 1 allows for a better exploitation of the inherent parallelism, which results in a shorter overall runtime. The next section explains in detail the proposed algorithm that optimises locally the exploitation of this parallelism. Note that a similar degree of freedom is brought in by the system bus that represents a shared resource for the data transfer edges as the DSP does for those processes in software.

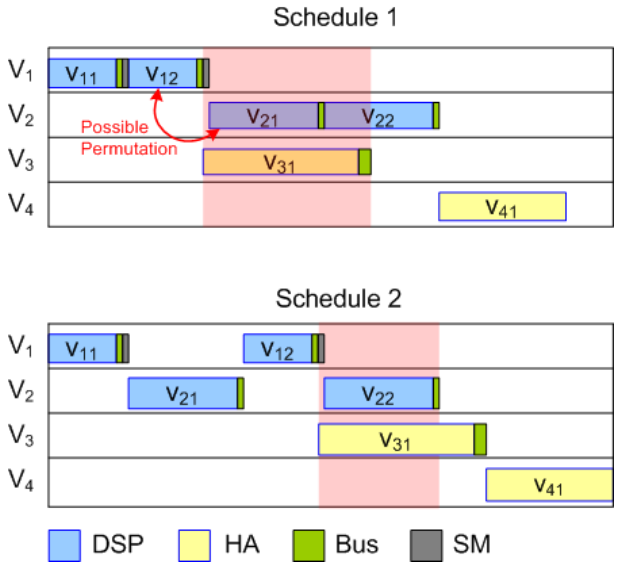

Fig. 4. *Two valid schedules for a single partitioning solution with different runtimes.*

IV. Rescheduling Algorithm for SDF Graphs

Due to the incremental movement of the partitioning algorithm through the search space, a problematic rescheduling scenario can be determined or reduced to be like in the following situation. In Fig. 5a the process x is scheduled on the DSP (blue) and now y should be moved to the DSP. The arrows indicate strict precedence, e.g. p_{x1} is a strict predecessor of x and s_{y1}, s_{y2} are strict successors of y. The remaining processes v_l, v_m are neither predecessors nor successors of x and y.

Now the algorithm detects the overlapping of the two processes x, y and updates the schedule for both possible orderings, depicted in Fig. 5b and 5c. For the following consideration only a local schedule update has to occur, i.e. only in the *region of interest (RoI)*. The RoI is given by the earliest start time of x and y, here 80 in case (b), and the latest end time of x and y, here 140 in case (c). In case (b) the process x is arranged before y and the subgraph of y that lies within the RoI has to be rescheduled, processes that lie further behind are not touched. After this limited and thus very fast rescheduling is completed, the RoI reflects locally the situation caused by the chosen schedule order *x before y*. The same procedure is performed for schedule order *y before x* in case (c). The fundamental idea behind the proposed algorithm becomes clear, when we compare the two possibilities (b) and (c). Intuitively, the best choice is the schedule that allows for 'more work being done' during the ROI. In this example case (c) allows for better exploitation of the inherent parallelism,

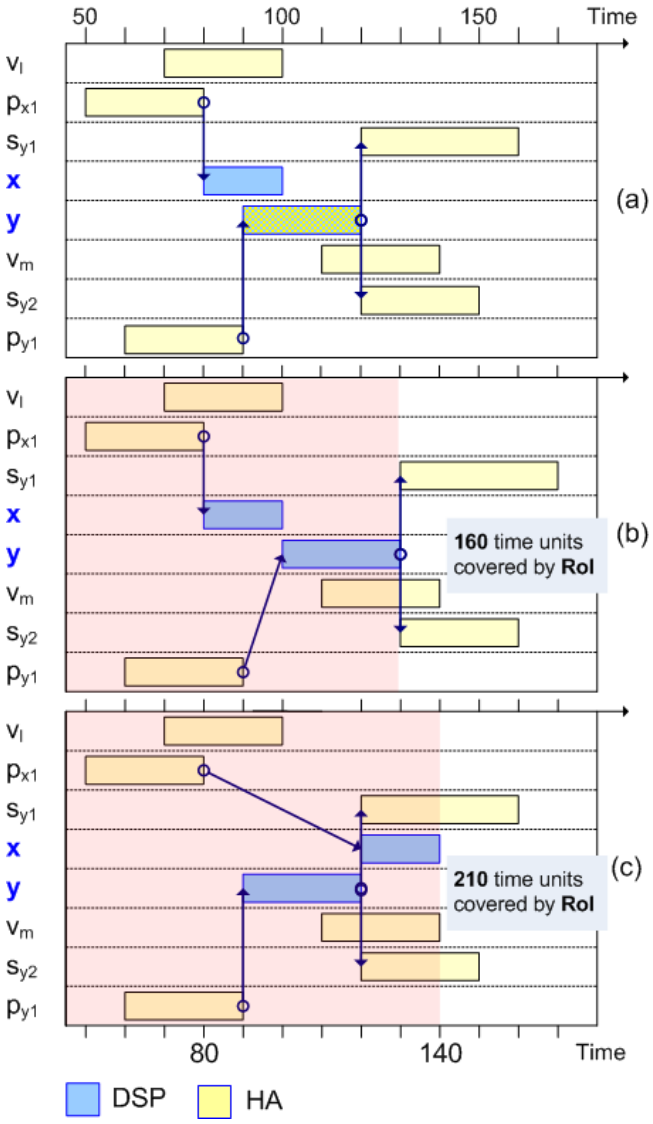

Fig. 5. *Parallelism exploitation and region of interest for the two possible solutions on the DSP: x before y and vice versa.*

which is measured by sum of all process computation times that lie within the RoI. In case (c) the RoI, indicated by the shaded area, covers 150 time units versus 130 in case (b). With respect to the interprocess communication exactly the same consideration is performed for the data transfer edges, which overlap on the system bus, and the same algorithm is used to solve this task. Note that the definition of the RoI also takes into account those effects that are caused by overlapping processes with substantially different start times (and/or end times).

It has to be mentioned that more complex situations occur rarely, where more than two processes overlap on

the DSP and all of their permutations would have to be analysed. To ensure a linear computation time, a fallback to an underlying list scheduling [14] is performed. Hence, it is always possible to reduce these scenarios in constant time to the situation of a choice between two processes as in the example above. In combination with the list scheduling, which also breaks the ties within the proposed algorithm, the linear runtime can be preserved for all cases.

Another important aspect is that typical graph structures for process graphs or multi-rate systems with the assumed granularity reveal an affinity to distinct properties (cp. task graph types in [26], [27]). Firstly, these graphs' density is usually *sparse*, i.e. the number of edges is typically in the same order of magnitude as the number of vertices, i.e. $|E| \approx 1.5 \dots 3|V|$. Secondly, they usually exhibit a strong $locality$ (k-neighbour graphs with $k \approx 2 \dots 4$), in other words a process is predominantly connected to another process on a similar graph level. Thirdly, they have a degree of parallelism of $\gamma = \frac{|V|}{|V_{LP}|} \geq 2$, $|V|$ being the overall number of vertices and $|V_{LP}|$ being the number of vertices in the longest (critical) path [28]. The density property has large impact on the computation complexity as it is shown in the next paragraph.The locality property is of importance as the proposed method naturally exposes its best performance on these graph structures. With decreasing degree of parallelism the graph degenerates towards a chain like structure for which a list scheduling is naturally efficient and much easier to implement.

a) Computation Complexity of the Rescheduling Algorithm: The following short listing illustrates the linear runtime of the proposed algorithm in a simplified form.

The function in line 1 is called when a process v_{org} is moved onto the DSP. As in general a process v_{org} is activated more than once, several process vertices v_{sag} exist in the single activation graph. Each of them has to be moved to the DSP schedule. Therefore, in line 3 it is looped over each v_{sag} originating from one v_{org} and its complexity can be approximated by the constant $c_1 = \frac{|V_{SAG}|}{|V|} \approx 3 \dots 5$. In line 8 it is checked for an overlap situation and just inserts the current vertex v_{sag}, when there is no collision (line 20). This simple insertion is the least complicated part of the algorithm with complexity $O(c_2) = c_2 = 1$.

Listing 1. *Pseudocode for the rescheduling algorithm*

```
 0
 1 MoveProcessToDSP(v_org) {
 2
 3  forall (v_sag of v_org) {
 4  // O(c_1)
 5
 6   // if v_sag overlaps with process x
 7   // on the DSP
 8   if (checkForOverlapOnDSP(v_sag)) {
 9
10    localScheduleUpdate(x before v_sag);
11    // ~= O(U)
12    evaluate();
13    // O(|V|)
14
15    localScheduleUpdate(v_sag before x);
16    // ~= O(U)
17    evaluate();
18    // O(|V|)
19   }
20   else pushToDSP(v_sag);
21   // O(c_2)
22
23   updateSchedule(v_sag);
24   // worst case: O(|V| + |E|)
25  }
26 }
```

If the overlap check in line 8 returns true, the two possible orderings have to be analysed (line 10, 15). The schedule update within the RoI can be reliably approximated by a constant running time $O(U)$, though its worst case running time is $O(|V| + |E|)$. As stated earlier, typical graphs for systems in this field are $sparse$, with a ratio of $\frac{|E|}{|V|} \approx 1.5 \dots 3$ i.e. every vertex v sees in average 2..6 edges, half of them connecting to successors $n_s \lesssim 3$. Additionally it is reasonable to assume that the running times of all vertices do not differ by more than an order of magnitude, which is certainly the case when the chosen granularities of the processes are not very different (i.e. some being whole filter structures and some being single operations). Thus, it can be assumed that the RoI covers only a fraction of the whole graph in an order of $h_l \approx 2$ hierarchy levels below vertex v, approximating $O(U) \lesssim O(\sum_{i=1}^{h_l} n_s^i) = O(12) = 12$. The correctness of this assumption has been empirically asserted. In line 12 and 17 all process invocations are checked for lying within the RoI and hence it is reported on the quality of the chosen ordering in $O(|V|)$.

In line 23 the schedule has to be updated completely according to the ordering decision. The complete schedule update is $O(|V| + |E|)$ only in the worst case, when just one start vertex exists and this vertex is modified. In all other cases shortcuts are possible and exploited.

The complete expression for the core of the rescheduling algorithm in O-notation is then:

$$\begin{aligned} O(c_1) \max(&O(|V| + |E|) + 2O(U) + 2O(|V|), \\ &O(|V| + |E|) + O(c_2)) \\ &= c_1(O(|V| + |E|) + 2U + 2O(|V|)) \end{aligned} \tag{3}$$

The leading term $O(|V| + |E|)$ remains linear.

V. Results

In this section we compare our approach, the algorithm for *local exploitation of parallelism* (LEP), to two classical list scheduling techniques with comparable complexity to demonstrate its performance.

The first list scheduling approach is the popular Highest Levels First (HLF) algorithm [14]. This technique assigns each vertex v a level, or priority. This level is defined as the largest sum of execution times along any directed path from v to an endnode of the graph over all existing endnodes. The classic approach is static, as the priority level is calculated beforehand and doesn't change during the scheduling process. In our case we have to modify this algorithm to be dynamic, since a vertex can have two different execution times and additional varying communication times between vertices, depending on its implementation type, hw or sw. This modification is a level or priority update of all vertices, which are predecessors of the just moved vertex, with a worst case runtime $O(|V| + |E|)$.

The second approach is the Earliest Task First (ETF) algorithm in [29] that calculates the earliest start time for every vertex and schedules the one with lowest start time first. Ties are broken towards those processes with the higher priority. This priority is calculated according to the HLF algorithm above. Thus this algorithm has to be modified accordingly to reflect the priority levels dynamically.

As regards the running time complexity, both of them are linear as well. The code lines 7-11 in Listing 1 are replaced by a constant time priority look up. That corresponds to a replacement of the term $4UO(|V|)$ in Equ. (3) by $O(1)$. The leading term is then still $O(|V|+|E|)$, though the difference in computation time is actually perceivable but not significant.

The random SDF graphs and their SAGs, generated for the algorithm evaluation, have two static properties: their density is always sparse $\frac{|E|}{|V|} \approx 1.5 \ldots 3$, and they all are k-neighbour graphs, with $k = 3$. The variable characteristics of the graphs are their number of vertices in the SDFG $|V|$, their number of vertices in the SAG $|V_{SAG}|$ and their degree of parallelism γ. The properties of an SDF graph are described as a 2-tuple $T_G = (|V|, |V_{SAG}|)$.

To compare the behaviour and performance of the algorithms for one specific graph, we have to simulate the search space traversal of a partitioning algorithm. For a graph with 15 vertices for instance the search space would be 2^{15}. We move one vertex out of 15 at a time and reschedule the SAG according to all three rescheduling algorithms. It has to be mentioned that within such a search space traversal distinct partitioning solutions could occur more than once. For cancellation of this effect up to a reasonable degree, a short tabu list has been implemented to avoid moving the same vertex twice in a row.

Thus we obtain stepwise a new partitioning solution π and three schedule lengths $SL_\pi(\text{HLF})$, $SL_\pi(\text{ETF})$, $SL_\pi(\text{LEP})$. Furthermore, we calculate a lower bound LB_π for the schedule length of each π. This lower bound is determined by the sum of the process computation times pct_h or pct_s and communication times ct_{sm} or ct_{bus} along the critical path. For all considered partitioning solutions Π_G of a graph G, the specific schedule lengths SL_π of one partitioning solution $\pi \in \Pi_G$ are summed up, as well as the specific lower bounds, LB_π. Thus, we obtain for all three algorithms and the lower bound a global sum over all considered schedule lengths, e.g. for LEP:

$$SL_{global}(\text{LEP}, \Pi_G) = \sum_{\pi \in \Pi_G} SL_\pi(\text{LEP}) \qquad (4)$$

Fig. 6 shows a bar chart, in which the x-axis shows different graph sizes. The y-axis shows the global sums over schedule lengths for the three algorithms normalised to the global sum over the lower bound schedule lengths. The results have been averaged over 30 different graphs for each size. It can be seen that the proposed algorithm

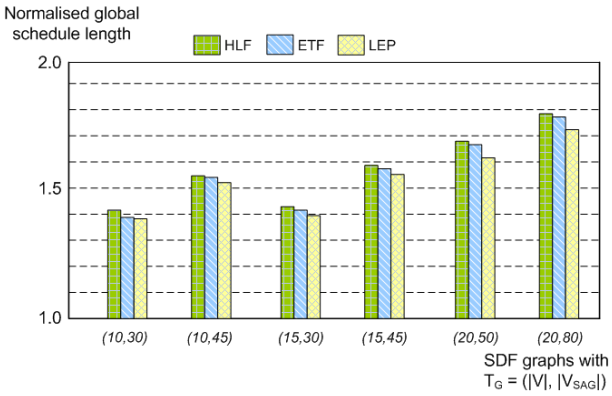

Fig. 6. *Averaged global schedule lengths normalised to the global lower bound schedule lengths over different graph sizes.*

creates better results for all depicted graph sizes with a larger margin going towards larger graphs. Smaller graphs than the depicted did not show in average any remarkable difference for the chosen schedules. That becomes clear when we plot the rescheduling algorithms not over their respective sizes but over their degree of parallelism γ, as shown in Fig. 7. The larger graphs tend to have a higher degree and they certainly offer more opportunities to exploit this parallelism. It can be stated that the proposed algorithm LEP benefits well from a higher degree of parallelism, whereas the HLF algorithm is focused on the current critical path, in other words, its decision is lead

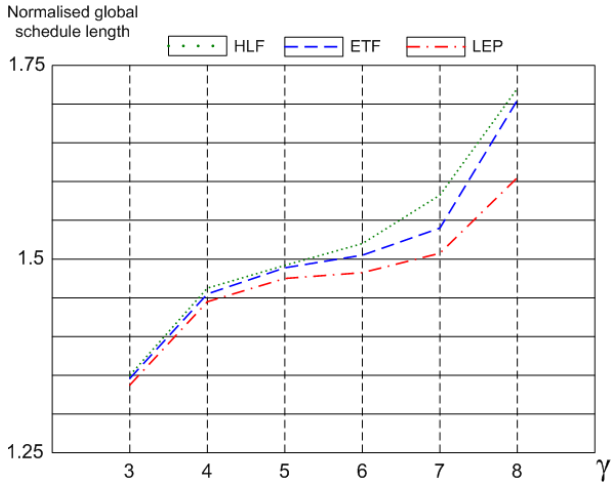

Fig. 7. *Averaged global schedule lengths normalised to the global lower bound schedule length over degree of parallelism* γ.

by a subgraph evaluation of the current vertex. The ETF algorithm obtains its decision from evaluating the precedence graph of the current vertex, with an HLF fallback mechanism that breaks the ties. None of them considers in any way the vertices that lie in parallel to the current vertex. For graphs with $\gamma \leq 3$ the proposed LEP algorithm did not show significant performance improvements.

VI. Conclusions

In this paper a fast rescheduling algorithm for SDF graphs has been proposed that shows better performance on typical task graph structures than two other list scheduling algorithms, which are popular among fast scheduling techniques. The linear runtime of the algorithm makes it a viable choice to be applied within hardware/software partitioning algorithms that have to evaluate a large number of partitioning solutions and move incrementally through the search space. The difference in performance is significant, of up to 10%, when graphs with a degree of parallelism of $\gamma \geq 3$ are under consideration.

Future work will be focused on possible linear time schedule updates for systems including multiple general-purpose processors and multi-bus structures. The extension towards SDF graphs including processes with variable execution times is of special interest in modern heterogeneous systems. As well as the incorporation of variable data rates, which we encounter especially in UMTS systems. Another challenging problem is the design of fast rescheduling algorithms for partitioning engines that do not show the *one move* traversal behaviour.

References

[1] P. Arató, Z. Á. Mann, and A. Orbán, "Hardware-software co-design for Kohonen's self-organizing map," in *Proceedings of the IEEE 7th International Conference on Intelligent Engineering Systems*, 2003.

[2] A. Kalavade and E.A. Lee, "The Extended Partitioning Problem: Hardware/Software Mapping and Implementation-Bin Selection," in *Proc. of Sixth International Workshop on Rapid Systems Prototyping (RSP)*, June 1995, pp. 12–18.

[3] P. Arató, Z. Á. Mann, and A. Orbán, "Algorithmic Aspects of Hardware/Software Partitioning," in *ACM Transactions on Design Automation of Electronic Systems*, January 2005, vol. 10, pp. 136–156.

[4] V. Srinivasan, S. Radhakrishnan, and R. Vemuri, "Hardware software partitioning with integrated hardware design space exploration," in *Proceedings of Design, Automation & Test in Europe (DATE)*, 1998.

[5] Bingfeng Mei, Patrick Schaumont, and Serge Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *Proceedings of ProRISC*, 2000.

[6] J. Henkel and R. Ernst, "An approach to automated hardware/software partitioning driven by high-level estimation techniques," 2001, number 2, pp. 273–289.

[7] M. L. Lopez-Vallejo, J. Grajal, and J. C. Lopez, "Constraint-driven system partitioning," in *Proceedings of Design, Automation & Test in Europe (DATE)*, 2000, pp. 411–416.

[8] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "System level hardware/software partitioning based on simulated annealing and tabu search," *Design Automation for Embedded Systems*, vol. 2, pp. 5–32, 1997.

[9] J. Grode, P. V. Knudsen, and J. Madsen, "Hardware resource allocation for hardware/software partitioning in the lycos system," in *Proceedings of Design, Automation & Test in Europe (DATE)*, Washington, DC, USA, 1998, pp. 22–27, IEEE Computer Society.

[10] A. Kalavade and E. A. Lee, "The extended partitioning problem: hardware/software mapping and implementation-bin selection," in *Proceedings of the Sixth IEEE International Workshop on Rapid System Prototyping (RSP)*, Washington, DC, USA, 1995, p. 12, IEEE Computer Society.

[11] R. K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design and Test of Computers*, 1993.

[12] M.R. Garey and D.S. Johnson, *Computers and intractability: a guide to NP-completeness*, W.H. Freeman, San Francisco, California, 1979.

[13] Walter H. Kohler and Kenneth Steiglitz, "Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems," *J. ACM*, vol. 21, no. 1, pp. 140–156, 1974.

[14] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," Tech. Rep. 6, Operations Research, 1961.

[15] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 175–187, 1993.

[16] Vivek Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA, USA, 1989.

[17] M. Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 3, pp. 330–343, 1990.

[18] Karam S. Chatha and Ranga Vemuri, "Magellan: multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs," in *Proceedings of the ninth international symposium on Hardware/software codesign (CODES)*, New York, NY, USA, 2001, pp. 42–47, ACM Press.

[19] M. Lopez-Vallejo and J.C. Lopez, "On the hardware-software partitioning problem: System modeling and partitioning techniques," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, no. 3, pp. 269–297, 2003.

[20] Robert P. Dick and Niraj K. Jha, "Mogac: a multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems," in *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design (ICCAD)*, Washington, DC, USA, 1997, pp. 522–529, IEEE Computer Society.

[21] B. Knerr, M. Holzer, and M. Rupp, "HW/SW Partitioning Using High Level Metrics," in *International Conference on Computing, Communications and Control Technologies (CCCT)*, June 2004, pp. 33–38.

[22] F. Vahid and T. Dm Le, "Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning," *Design Automation for Embedded Systems*, pp. 237–261, 1997.

[23] E.A. Lee and D.G. Messerschmitt, "Static scheduling of synchronous data-flow programs for digital signal processing," in *IEEE Transactions on Computers*, 1987, vol. 36, pp. 24–35.

[24] E.A. Lee, "Overview of the Ptolemy Project," Tech. Rep., University of Berkeley, March 2001, http://ptolemy.eecs.berkeley.edu.

[25] "CoWare SPW 4," Tech. Rep., CoWare Design Systems, 2004, http://www.coware.com/products/spw4.php.

[26] Minsoo Ryu and Seongsoo Hong, "A Period Assignment Algorithm for Real-Time System Design," in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. 1999, pp. 34–43, Springer-Verlag.

[27] Yu-Kwong Kwok and Ishfaq Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 5, pp. 506–521, 1996.

[28] Y. Le Moullec, N. Ben Amor, J-Ph. Diguet, M. Abid, and J-L. Philippe, "Multi-Granularity Metrics for the Era of Strongly Personalized SOCs," pp. 674–679, March 2003.

[29] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM J. Comput.*, vol. 18, no. 2, pp. 244–257, 1989.

[30] K. S. Chatha and R. Vemuri, "An Iterative Algorithm for Hardware-Software Partitioning, Hardware Design Space Exploration and Scheduling," , no. 5, pp. 281–293, 2000.