Self-timed Scheduling and Execution of Nonlinear Pipelines with Parallel Stages

Lars Lucas[1], Tobias Schuele[2], and Wolfgang Schwitzer[1]

[1] Technische Universität München, Fakultät für Informatik
Boltzmannstr. 3, 85748 Garching, Germany
{lucas,schwitze}@in.tum.de
[2] Siemens AG, Corporate Technology
Otto-Hahn-Ring 6, 81739 München, Germany
tobias.schuele@siemens.com

Abstract. Applications that process continuous streams of data, e.g., sensor signals, video images, network packets, etc. are well-suited for pipelined execution on multicore processors. In many cases, however, the applications are subject to real-time constraints, especially in embedded systems. Besides maximizing the throughput, it is therefore important to minimize deviations in the timing. To solve this problem, we propose a method for self-timed scheduling and parallel execution of stream-based applications in soft real-time environments. Our experimental results show significantly lower latencies compared to state-of-the-art approaches, while achieving high throughput.

1 Introduction

Multicore processors, which are prevalent in laptops, desktop computers, and servers, increasingly find their way into embedded systems [1]. For instance, many smartphones already contain processors with two or more cores. Unlike personal computers, however, embedded systems are often subject to real-time constraints. Typical examples are vehicle controllers, medical devices, and machines for industrial automation. In such systems, predictability of the timing plays an important role to ensure correct interaction with the environment. In general, one distinguishes between soft and hard real-time systems. For the former it must be guaranteed that the given deadlines are met in all cases. Hard real-time systems are typically found in safety-critical areas like aviation, where high demands are put on reliability. In soft real-time systems, a deadline may be missed without causing harm. However, to maintain the quality of service, a major goal in the design of such systems is to reduce deviations from the desired timing.

Consider, for example, a sorting machine that discards broken objects (Fig. 1). The machine consists of a conveyor belt, which transports the objects to be sorted at a fixed speed, a camera, which produces a video stream of the objects on the conveyor belt, and a pusher, which pushes broken objects into a box. The camera and the pusher are connected to a computer, which analyzes the incoming video stream and triggers the pusher when a broken object is detected or the considered object could not be classified within the given time span. In order to run the conveyor belt at a high speed, the image processing application must have a high throughput. Additionally, the latency should be

J.M. Lourenço and E. Farchi (Eds.): MUSEPAT 2013, LNCS 8063, pp. 1–12, 2013.

as low as possible to keep the conveyor belt as short as possible. Increasing the distance between the camera and the pusher relaxes the timing constraints, but increases the costs and requires additional space. As usual in soft real-time systems, the deadline may be missed without causing harm. Nevertheless, this should only happen in rare cases to reduce the number of intact objects that are thrown away.

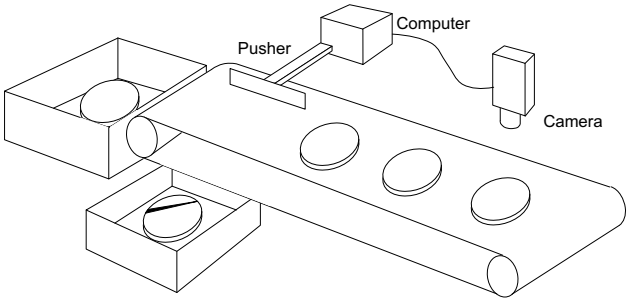


Fig. 1. Example for a soft real-time system (sorting machine)

In the past decades, a plethora of techniques for scheduling tasks on parallel systems have been developed, ranging from completely static to fully dynamic approaches [2]. Static scheduling algorithms are used to compute schedules before an application is executed [3,4,5]. Since the points of time at which the tasks are executed are known a priori, static scheduling allows a maximum degree of predictability. However, static scheduling algorithms require precise information on the worst-case execution times of the tasks, which are hard to obtain for modern multicore processors with shared caches. In fully dynamic approaches, all scheduling decisions are performed at run-time. A prominent representative of this class is work stealing, where idle processor cores steal tasks from other (busy) cores to balance the load [6]. A major advantage of dynamic scheduling is the ability to deal with varying execution times that may arise when shared resources such as the main memory are accessed. The downside is limited predictability, since there is usually little a priori information on the timing.

Self-timed scheduling approaches combine the advantages of both worlds. This is achieved by statically assigning the tasks to the processor cores using (not necessarily safe) estimates of the tasks' worst-case execution times. The result is a list for each processor core specifying the sequential ordering of the tasks. As in static scheduling, the lists are processed in an infinite loop, but the points of time at which the tasks are executed are determined dynamically. To this end, the runtime systems keeps track of the available data and ensures that a task is not executed until all input data is available, i.e., synchronization is performed during run-time. This makes self-timed scheduling robust with respect to changes in the execution times of the tasks.

To utilize the power of multicore processors, computations have to be split into tasks that can be executed in parallel. For that purpose, applications that process continuous streams of data, such as the sorting machine described previously, may be executed in a pipelined fashion [7,8]. This is one of the reasons why stream processing, which has a long history in computer science [9], has gained resurgent interest in recent years

(see, e.g., [10,11,12]). In its basic form, however, the throughput of a pipeline is limited by the slowest stage. To solve this problem, multiple invocations of a stage may be executed in parallel [13]. Hence, a new item, subsequently also called token, may enter a parallel stage before the previous one has left it, provided that the invocations do not interfere. For stages with side effects it is usually required that at most one invocation is active at a time. Such stages are said to be serial.

Chain-like (linear) pipelines increase the throughput, but do not decrease the latency, the time required to process an element from a stream. A reduction of the latency can be achieved using nonlinear pipelines, which may exhibit an arbitrary structure. In addition to temporal parallelism, such pipelines exploit spatial parallelism by processing independent substreams in parallel. In digital signal processing, for example, streams are frequently fed into multiple filter chains that may operate simultaneously.

In this paper, we present a method for self-timed scheduling of nonlinear pipelines with parallel stages. Our approach is well-suited for stream processing applications in soft real-time systems, especially when latency is a crucial concern. It allows developers of embedded systems to leverage from multicore processors without the need for complex dynamic schedulers, which often exhibit unpredictable runtime behavior.

An efficient algorithm for scheduling pipelines on multiprocessor systems was presented in [14]. In contrast to our approach, however, it cannot deal with nonlinear pipelines. The scheduling algorithm described in [15] solves this problem, but is optimized for throughput and does not aim to minimize the latency. The work of Banerjee et al. [16] has the same objective, but also supports heterogeneous systems. All three approaches assume a fixed assignment of pipeline stages to processors (cores), which may significantly hurt cache locality, since data flowing between successive pipeline stages must be physically transferred between the corresponding cores. Our approach avoids this problem in that successive stages are preferably scheduled on the same core.

The remainder of this paper is organized as follows: In the next section, we describe the foundations of our work and introduce the notion of extended task graphs used to model stream processing applications. In Sect. 3, we present our scheduling algorithm, which is based on the modified critical path method [17]. After that, we sketch the implementation (Sect. 4) and discuss experimental results (Sect. 5). Finally, we conclude with a summary and directions for future work in Sect. 6.

2 Foundations

An extended task graph (ETG) consists of a set of nodes that represent the computations of an application and a set of edges that represent the data flow. ETGs are essentially ordinary task graphs [4] except that each node has an additional attribute that specifies whether it is serial or parallel.

Definition 1. *An extended task graph is a directed acyclic graph* $G = (V, E, r, p, c)$*, where*

- V *is the set of nodes,*
- $E \subset V \times V$ *is the set of edges,*
- $r : V \to \mathbb{N}^+$ *is a function that associates with each node its computations costs,*

- $p : V \to \{0, 1\}$ *is a function that returns* 1 *if the node is parallel, and*
- $c : E \to \mathbb{N}^+$ *is a function that associates with each edge* $(v, w) \in E$ *the communication costs between nodes* v *and* w.

A node without predecessors is a *source* and a node without successors is a *sink* (for simplicity, we restrict ourselves to ETGs with a single source and a single sink). We write $v \rightsquigarrow w$ iff there is a path from v to w with $v, w \in V$ ($v \rightsquigarrow v$ holds for all $v \in V$). Additionally, we need the following definitions (see [4]):

- The *length* of a path $p = [v_i, v_{i+1}, \ldots, v_j]$ is the sum of the computation and communication costs, i.e., $\mathrm{len}(p) = \sum_{i \leq k \leq j} r(v_k) + \sum_{i \leq k < j} c(v_k, v_{k+1})$.
- The *bottom level* $\mathrm{bl}(v)$ of a node v is the length of the longest path from v to the sink v_{sink}, i.e., $\mathrm{bl}(v) = \max\{len(p) \mid p = [v, \ldots, v_{sink}]\}$.[1]
- A *critical path* cp is a longest path, i.e., $\mathrm{len}(\mathrm{cp}) = \mathrm{bl}(v_{source})$.
- The *as-late-as-possible* (ALAP) time of a node v is defined as $\mathrm{alap}(v) = \mathrm{len}(\mathrm{cp}) - \mathrm{bl}(v)$.

Intuitively, the ALAP time is the latest point of time a node must be scheduled without unnecessarily increasing the span of the resulting schedule. Figure 2 shows a sample ETG consisting of five nodes, where multiple instances of nodes C and D may be executed in parallel. The length of the critical path $[A, C, D, E]$ is 14. The bottom levels and ALAP times are given next to each node.

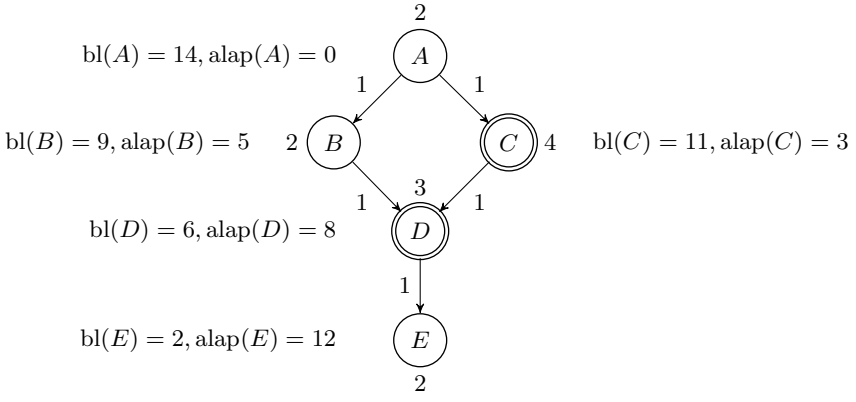


Fig. 2. Sample ETG consisting of five nodes (nodes C and D are parallel)

In order to exploit pipeline parallelism, we unfold ETGs. For that purpose, we create a fixed number of copies of an ETG, where each copy represents a run through the pipeline. Recall, however, that serial stages must be executed one after the other. This can be accomplished by introducing additional edges in the unfolded ETG such that the copies of a serial stage depend on each other. More formally, we denote by $G_i = (V_i, E_i, w_i, p_i, c_i)$ the ETG obtained from an ETG $G = (V, E, r, p, c)$ by labeling every

[1] An algorithm for computing bottom levels can be found in [4].

node $v \in V$ with index i such that G and G_i are isomorphic. The union of two ETGs G_i and G_j, written $G_i \cup G_j$, is defined elementwise.

Definition 2. *Let $(V, E, w, p, c) = \bigcup_{i=1}^{n} G_i$ be the ETG consisting of n copies of an ETG G. Then, $G' = (V, E \cup E', r, p, c')$ is the ETG obtained by unfolding G n times, where $E' \subset V \times V$ and $c' : E \cup E' \to \mathbb{N}$ are defined as follows:*

$$E' = \{(v_i, v_{i+1}) \mid \neg p(v_i) \wedge 1 \leq i < n\}$$

$$c'(e) = \begin{cases} 0 & \text{if } e \in E' \\ c(e) & \text{otherwise} \end{cases}$$

The above definition ensures that for every serial node there is an edge between successive runs. This way, parallel execution of different instantiations of a serial node is avoided. Since these additional edges are used for synchronization only and not for exchanging data, we set their communication costs to zero. Figure 3 shows the ETG of Fig. 2 unfolded three times (the computation and communications costs of the original ETG are omitted for the sake of clarity).

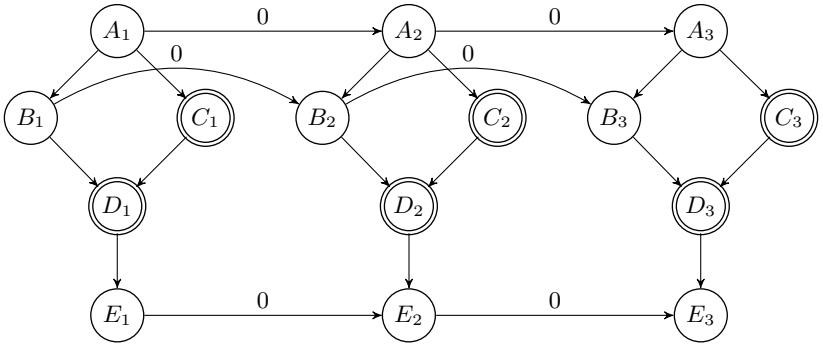


Fig. 3. ETG of Fig. 2 unfolded three times

3 Scheduling Algorithm

Having unfolded an ETG, we can, at least in principle, employ any static scheduling algorithm to obtain a schedule. It is not even necessary to distinguish between serial and parallel stages, as this information is implicitly encoded in the synchronization edges. Since we consider self-timed scheduling, the temporal assignment of tasks to processors can be ignored. In fact, we are only interested in the processor allocation, i.e., the sequence of tasks executed by a processor. As mentioned in the introduction, synchronization is performed on-the-fly to cope with varying runtimes. Note, however, that the resulting schedules only comprise finitely many runs. In order to execute nonterminating applications, the schedules are repeated in an infinite loop. This is referred to as blocked scheduling, where the number of unfoldings is called the blocking factor [1].

Most scheduling algorithms aim to minimize the span of the resulting scheduling. Regarding nonterminating systems this means that the average throughput is maximized. While this is reasonable in high performance computing, it is usually not the primary goal in embedded systems that are subject to real-time constraints. Our scheduling algorithm solves this problem in that it not only maximizes the throughput, but also minimizes the latency. It is based on the modified critical path (MCP) algorithm developed by Wu and Gajski [17], which is widely used in practice and performs very well compared to other scheduling algorithms [18]. The basic idea of the MCP algorithm is to schedule nodes with the least ALAP times first, where ties are broken dependent on the ALAP times of the nodes' descendants. The algorithm comprises two steps[2]: In the first step, a list is created for each node containing the ALAP times of the node itself and all its descendants in ascending order. In the second step, the nodes are scheduled one after the other according to the lexicographical order of the obtained lists.

Given a set of m processors, we represent a schedule by two functions, the processor allocation function $P : V \to \{1 \dots m\}$ and the start time function $S : V \to \mathbb{N}$. These two functions specify for each node on which processor and at what time it shall be executed. As usual in static scheduling, we neglect communication costs if two adjacent nodes are scheduled on the same processor. Thus, the communication costs $c_p(u, v)$ when scheduling node v on processor p are zero if $P(u) = p$ and $c(u, v)$ otherwise. The earliest point of time v can be executed is then defined as follows:

$$\text{earliest}_p(v) = \max\{S(u) + r(u) + c_p(u, v) \mid (u, v) \in E\}$$

However, executing v at a given point of time t is only possible if it does not overlap with any other node scheduled on processor p:

$$\text{feasible}_p(v, t) = \forall w \in V.\, (P(w) = p) \to (t + r(v) \leq S(w)) \vee (S(w) + r(w) \leq t)$$

Consequently, the start time of v is the earliest feasible time:

$$\text{start}_p(v) = \min\{t \in \mathbb{N} \mid t \geq \text{earliest}_p(v) \wedge \text{feasible}_p(v, t)\}$$

Figure 4 shows our algorithm.[3] In contrast to MCP, it takes into account the unfoldings of the original ETG, which is the key to minimizing the latency. For that purpose, the subgraphs $V_1 \cup V_2 \cup \dots \cup V_n$ are processed in ascending order (lines 3–11). This way, it is guaranteed that no nodes are scheduled before all nodes of the previous unfolding have been scheduled. For each node v inside a subgraph, the algorithm creates a list l_v (lines 5–8) containing v and its successors sorted by their ALAP times. Once the nodes of a subgraph have been processed, these lists are sorted lexicographically and appended to the global list L (line 10). Finally, the nodes are scheduled in a greedy fashion (lines 12–23) w.r.t. their start times (lines 16–21).

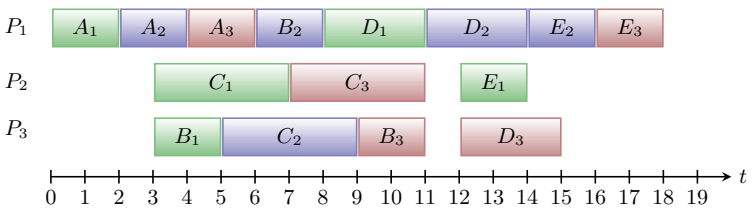
Figure 5 depicts two schedules for the ETG of Fig. 3, one computed with the classical MCP algorithm and the other one with our algorithm (LMCP). In both cases, we assumed a system with three processors. It should be emphasized that nodes belonging

[2] See *MCP Revisited* by M.-Y. Wu (`http://www.ece.unm.edu/~wu/mcp/mcp.pdf`).

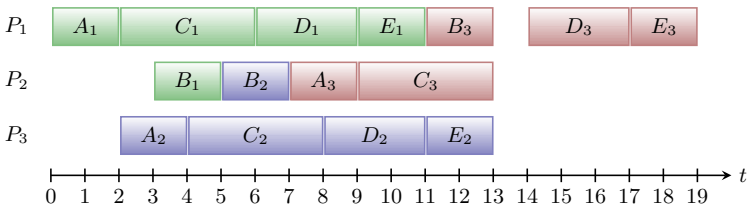
[3] An expression $[\, x \mid p(x) \,]$ denotes a list of elements each satisfying the predicate p. The second, optional parameter of the function sort is a binary predicate specifying the sorting order.

```
1  function LMCP(V_1 ∪ V_2 ∪ ··· ∪ V_n, E, r, p, c)
2    L := [];
3    // process subgraphs in ascending order (n = number of unfoldings)
4    for i := 1 to n do
5      for each v ∈ V_i do
6        // sort node and its descendants according to their ALAP times
7        l_v := sort([ w | v ⇝ w ], (a, b) → alap(a) < alap(b));
8      end for
9      // sort lists lexicographically and append them to L
10     L := append(L, sort([ l_v | v ∈ V_i ]));
11   end for
12   (P, S) := (∅, ∅);
13   while L ≠ [] do
14     v := head(head(L));
15     (P(v), S(v)) := (0, ∞);
16     // determine processor with least start time (m = number of processors)
17     for p := 1 to m do
18       if start_p(v) < S(v) then
19         (P(v), S(v)) := (p, start_p(v));
20       end if
21     end for
22     L := tail(L);
23   end while
24   return (P, S);
25 end function
```

Fig. 4. Levelized modified critical path algorithm (LMCP)



(a) Modified critical path algorithm (MCP)



(b) Levelized modified critical path algorithm (LMCP)

Fig. 5. Schedules for the unfolded ETG of Fig. 3 using three processors

to different unfoldings such as A_1 and A_2 might be executed on different processors, since they represent different tasks. Moreover, note that execution of B_1, C_1, and C_2 might overlap, since B and C are independent of each other and C is a parallel node. The maximum latency for the schedule in Fig. 5 (a) is 14 time units, whereas processing an element according to Fig. 5 (b) requires at most 12 time units.

4 Implementation

We have implemented our approach on top of a framework called FluenC [19,20], which allows developers to easily model parallel stream processing applications in C++. Typical fields of application are digital signal and image processing. As a major advantage, FluenC is based on a deterministic model of computation (dataflow process networks [21]), which simplifies testing and debugging. Determinism is particularly important in safety-critical systems, where high demands are put on correctness and reliability. For efficiency reasons, most of the classes and functions are implemented using templates. This also facilitates type safety: connecting the output of a pipeline stage to an input of a different type results in a compile-time error. Moreover, FluenC is completely lock-free, and hence does not suffer from costly synchronization operations. Figure 6 sketches the implementation of our running example.

```
...
// Serial node with input of type S and output of type T
class B: public network_t::serial<in<S>, out<T>> {
public:
  void operator()(const S& input, T& output) const {
    // computations
  }
} myB;
...
// Connect output of B with first input of D (port 0)
myB >> myD.port<0>();
```

Fig. 6. Excerpt from the implementation of the ETG shown in Fig. 2

In order to determine the runtimes of the nodes, we execute the resulting graph several times on the target processor and use the measured information for scheduling. In a second step, we create a list for each processor (core) containing the nodes to be executed. These lists are then repeatedly processed in parallel (a barrier at the end of the schedules synchronizes successive runs). Data dependencies are resolved dynamically, i.e., a node is not executed until all input data is available. For that purpose, each node has an associated table in which the incoming data is stored depending on the number of the unfolding.[4] If a processor (core) is ready, but some data is still missing, it waits until the next table entry is complete.

[4] Simply speaking, the columns represent the inputs and the rows the unfoldings.

5 Experimental Results

To evaluate our approach, we parallelized three applications, a sorting network (SORT), a fast Fourier transform (FFT) for digital signal processing, and an image recognition application (IMGR) developed at Siemens for industrial automation. The latter is used to identify moving objects in a stream of video images (cf. Sect. 1). The image processing steps range from simple local filters to complex operations, e.g. the computation of connected components and image compression. All applications exhibit a nonlinear structure and contain parallel stages.

We executed the benchmarks using self-timed as well as dynamic scheduling in order to assess the performance characteristics of both approaches. For dynamic scheduling we employed the scheduler of Intel's Threading Building Blocks[5], a state-of-the-art C++ library for parallel programming. The scheduler is based on the work stealing paradigm [6], which has proven to be very efficient in practice. To obtain comparable results, we used the same platform in both scenarios and only exchanged the underlying scheduler. All experiments were performed on a system with two quad-core Xeon E5440 processors with 2.83 GHz and 6 GB RAM running Linux.

For each benchmark, we varied the number of processor cores from one to eight and the number of unfoldings from one to ten. Moreover, we executed each benchmark 5000 times, resulting in a total number of approx. 2.4 million measured latency values. Table 1 shows the results, where L_{seq} denotes the latency (in milliseconds) and T_{seq} the throughput (elements per second) for sequential execution. For the parallel implementations, m specifies the number of processor cores and n the number of unfoldings. Besides the minimum, average, and maximum latency, we also list the standard deviation L_{dev}, which serves as a measure for the degree of timing predictability.

Table 1. Experimental results

Bench.	L_{seq}	T_{seq}	Scheduling	Objective	m	n	L_{min}	L_{avg}	L_{max}	L_{dev}	T
SORT	44	23	self-timed	min. L_{max}	2	1	33	39	67	2.8	25
				sat. T	6	9	47	94	180	17.0	38
			dynamic				57	222	421	49.6	40
				max. T	5	10	43	231	619	58.2	43
FFT	11	93	self-timed	min. L_{max}	6	1	6	6	10	0.2	167
				sat. T	8	4	10	11	26	0.6	343
			dynamic				10	15	28	1.1	271
				max. T	6	8	8	26	52	7.6	306
IMGR	84	12	self-timed	min. L_{max}	2	5	53	54	55	0.2	22
				sat. T	8	4	74	78	95	1.3	45
			dynamic				91	105	152	6.8	38
				max. T	7	10	104	192	340	34.3	52

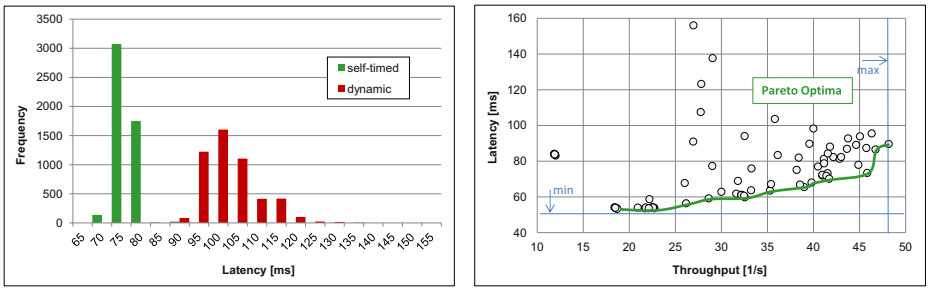
[5] `http://threadingbuildingblocks.org/`

We considered three different objectives in Table 1 regarding latency and throughput in order to summarize the results. For self-timed scheduling, we determined a combination of m and n such that the maximum latency L_{max} is minimal. Additionally, we determined when the throughput saturates, i.e., higher values for m or n do not yield significantly higher values T. To provide a direct comparison, we also list the results for dynamic scheduling with the same values for m and n. Finally, we determined a combination of m and n such that the throughput using dynamic scheduling is maximal.

SORT. Using self-timed scheduling, the average latency L_{avg} of 39 ms is slightly less than in the sequential case (89%) and the throughput of 25 is slightly higher (109%). However, it should be noted that the maximum latency is 67 ms (only 44 ms in the sequential case). Nevertheless, the standard deviation L_{dev} of 2.8 ms is comparatively low. The remaining results of SORT indicate that this benchmark does not respond well to pipelined execution, no matter which scheduling variant is employed. Even with throughput optimized dynamic scheduling, the maximum throughput is only 43 (187%) on five cores, which is relatively low compared to the other benchmarks.

FFT. If latency reduction is the primary objective, self-timed scheduling yields good results: L_{avg} is with 6 ms significantly lower than in the sequential case (55%), while still a high throughput of 167 compared to 93 (180%) is achieved. The standard deviation of 0.2 ms is remarkably low. An interesting result is the saturated throughput of 343 elements per seconds using self-timed scheduling compared to the maximum throughput achievable with dynamic scheduling of 306 elements per second. This is presumably due to the fact that the data chunks to be processed are rather small, which results in a high scheduling overhead. To sum up, self-timed scheduling pays off in significantly reduced, highly predictable latencies combined with increased throughput.

IMGR. The IMGR benchmark responds well to self-timed scheduling with respect to latency minimization: All latency values are clearly below the sequential case (55 ms in the worst case instead of 84 ms). This is merely 1 ms more than L_{avg} and 2 ms more than L_{min}. Together with the very low standard deviation 0.2 ms, these results indicate that self-timed scheduling offers highly predictable latencies for this benchmark and almost twice the throughput (183%) using two processor cores. The saturated throughput is with 45 elements per second slightly below what is achievable using dynamic scheduling (52 elements per second), whereas L_{max} = 95 ms is only a fraction of the 340 ms required to achieve such a throughput. Again, the standard deviation is very low (L_{dev} = 1.3 ms). Figure 7 (a) gives a more detailed picture of the latency distribution for both scheduling variants ($m = 8$ and $n = 4$). The measured values for self-timed scheduling span a range of 21 ms (from 74 to 95 ms), where the majority of the runs completed within 75 and 85 ms, which is a time window of only 10 ms. The latencies in dynamic scheduling are distributed over a much wider range of 61 ms (from 91 to 152 ms). Figure 7 (b) illustrates the relationship between throughput and average latency for IMGR (all combinations of m and n). As indicated by the Pareto frontier, the throughput can be increased from approx. 18 to 45 elements per second at the cost of 20 ms longer processing times (L_{avg} increases from 53 to 73 ms). The main reason for this is that our algorithm tries to fill holes in the schedules in order to reduce idle times. Consequently, increasing processor utilization may lead to fragmented schedules which in turn result in increased latencies.



(a) (b)

Fig. 7. Latency distribution for IMGR (eight cores, four unfoldings) and Pareto frontier (throughput vs. average latency) using self-timed scheduling

6 Summary and Conclusion

Besides high performance, multicore processors offer several advantages for embedded systems, in particular reduced power consumption and heat dissipation. However, the migration of sequential software to multicore processors poses serious challenges to the developers. From an industrial perspective, one of the major challenges is the protection of investment, i.e., the reuse of legacy code. Our experience has shown that stream-based applications can be parallelized relatively easy by means of pipelining, whereas the exploitation of data parallelism often requires significant refactoring.

Another challenge concerns the implementation of real-time systems. Most frameworks for parallel programming are optimized for throughput and disregard latency. In fact, the employed schedulers are usually not fair so that some tasks may even starve. To solve this problem, we proposed a polynomial-time algorithm for self-timed scheduling of nonlinear pipelines with parallel stages. It is based on the modified critical path algorithm, but takes into account the latency. For the largest of our benchmarks, an image recognition application from industrial automation, we achieved a significant reduction of the latency with only a slight drop of throughput.

It should be emphasized that our approach is not intended for hard real-time systems. Guaranteeing safe upper bounds on the execution times of parallel applications requires special hardware support and is an active area of research [22]. Another issue is the determination of an optimal number of unfoldings. Auto-tuning [23] can help to explore the design space and to find a configuration that fits the application's requirements regarding latency and throughput.

References

1. Sriram, S., Bhattacharyya, S.: Embedded Multiprocessors: Scheduling and Synchronization, 2nd edn. CRC Press (2009)
2. Lee, E., Ha, S.: Scheduling strategies for multiprocessor DSP. In: Global Telecommunications Conference, Dallas, TX, USA. IEEE (1989)
3. Davis, R., Burns, A.: A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. Technical Report YCS-2009-443, University of York, Department of Computer Science (2009)

4. Sinnen, O.: Task Scheduling for Parallel Systems. Wiley (2007)
5. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys 31, 406–471 (1999)
6. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. In: Annual Symposium on Foundations of Computer Science (FOCS), Santa Fe, NM, USA, pp. 356–368. IEEE (1994)
7. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison Wesley (2005)
8. Ortega-Arjona, J.: Patterns for Parallel Software Design. Wiley (2010)
9. Stephens, R.: A survey of stream processing. Acta Informatica 34(7), 491–541 (1997)
10. Aldinucci, M., Torquati, M., Meneghin, M.: FastFlow: Efficient parallel streaming applications on multi-core. Technical Report TR-09-12, Università di Pisa, Dipartimento di Informatica, Italy (September 2009)
11. Otto, F., Pankratius, V., Tichy, W.F.: XJava: Exploiting parallelism with object-oriented stream programming. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 875–886. Springer, Heidelberg (2009)
12. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)
13. MacDonald, S., Szafron, D., Schaeffer, J.: Rethinking the pipeline as object-oriented states with transformations. In: High-Level Parallel Programming Models and Supportive Environments (HIPS), Santa Fe, NM, USA, pp. 12–21. IEEE Computer Society (2004)
14. Bokhari, S.: Partitioning problems in parallel, pipelined, and distributed computing. IEEE Transactions on Computers 37(1), 48–57 (1988)
15. Hoang, P., Rabaey, J.: Scheduling of DSP programs onto multiprocessors for maximum throughput. IEEE Transactions on Signal Processing 31(6), 2225–2235 (1993)
16. Banerjee, S., Hamada, T., Chau, P., Fellman, R.: Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems. IEEE Transactions on Signal Processing 43(6), 1468–1484 (1995)
17. Wu, M.Y., Gajski, D.: Hypertool: A programming aid for message-passing systems. IEEE Transactions on Parallel and Distributed Systems 1, 330–343 (1990)
18. Kwok, Y.K., Ahmad, I.: Benchmarking and comparison of the task graph scheduling algorithms. Journal of Parallel and Distributed Computing 59(3), 381–422 (1999)
19. Schuele, T.: A coordination language for programming embedded multi-core systems. In: International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), Hiroshima, Japan. IEEE (2009)
20. Schuele, T.: Efficient parallel execution of streaming applications on multi-core processors. In: International Conference on Parallel, Distributed and Network-Based Processing (PDP), Ayia Napa, Cyprus. IEEE (2011)
21. Lee, E., Parks, T.: Dataflow process networks. Proceedings of the IEEE 83(5), 773–801 (1995)
22. Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., Cassé, H., Rochange, C., Quinones, E., Uhrig, S., Gerdes, M., Guliashvili, I., Houston, M., Kluge, F., Metzlaff, S., Mische, J., Paolieri, M., Wolf, J.: MERASA: Multi-core execution of hard real-time applications supporting analysability. IEEE Micro 30(5), 66–75 (2010)
23. Karcher, T., Schaefer, C., Pankratius, V.: Auto-tuning support for manycore applications: perspectives for operating systems and compilers. ACM SIGOPS Operating Systems Review Archive 43(2), 96–97 (2009)