Available online at www.sciencedirect.com

ScienceDirect

Procedia CIRP 41 (2016) 864 – 869

www.elsevier.com/locate/procedia

48th CIRP Conference on MANUFACTURING SYSTEMS - CIRP CMS 2015

Analysis and design of computerized numerical controls for execution on multi-core systems

José María Vivanco[a], Matthias Keinert[a,*], Armin Lechler[a], Alexander Verl[b]

[a]*Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW), Seidenstr. 36, Stuttgart, 70174, Germany*
[b]*Fraunhofer-Gesellschaft, Hansastr. 27c, Munich, 80686, Germany*

* Corresponding author. Tel.: +49-711-685-84625; fax: +49-711-685-82808. *E-mail address:* matthias.keinert@isw.uni-stuttgart.de

Abstract

Multi-core processors offer a performance increment compared to single-core platforms. This leap in performance is desired to be ported to computerized numerical controls. However, in order to profit of the benefits multi-core processors can bring, the software has to be tailored to real parallel execution. In this paper a concept is proposed for partitioning numerical control software functions for being concurrently executed on multi-core systems. Specifically, the interpreter and the cutter radius compensation modules have been analyzed for devising a feasible parallel architecture. The parser algorithm has been implemented following the proposed scheme in a thread-based approach. Experiments were conducted under a real time Linux kernel extension utilizing the PREEMPT_RT patch. The results were compared against its serial version in terms of execution times to validate the concept.

Peer-review under responsibility of the scientific committee of 48th CIRP Conference on MANUFACTURING SYSTEMS - CIRP CMS 2015

*Keywords: Computerized numerical control; M*ulti-core; Parallel computing

1. Introduction

Since its advent Computerized Numerical Controls (CNC) have advanced the manufacturing industry. They came to substitute manual machining when high accuracy and complexity is required and contributed simultaneously to reduce lead time. But the manufacturing needs became increasingly challenging and still are, asking for efficient solutions from the research community. These necessities involve productivity and quality enhancements, researches have been directed to find good levels in both. Methods such as adaptive control, high level interpolators, high-speed contouring control, and feed-rate optimization, among others, have been proposed [1-3]. Most of these methods increase the complexity of the algorithms and in turn demand better computing performance. Similarly, the functionality of CNC systems has been extended considerably. The most modern systems are capable to perform monitoring for diagnosis, collision detection, and data recording for energy considerations [4-6]. These features demand for high capability of the computing platform to process large amounts of data in short time [7]. While the complexity in CNC software is growing, single-core processors performance is not getting considerable increases any more [8]. Multi-core processors offer a possibility to keep increasing performance by adding more budgets of transistors on a single device [8]. Furthermore this technology outperforms its single-core counterpart in the way of dealing with thermal considerations and power consumption [9]. This condition is suggesting the migration of CNC systems to multi-core platforms. A parallel software concept for allowing concurrent execution on multi-core processors of CNC functions is proposed in this paper. Specifically, the suggested design consists in partitioning the Interpreter (IPR) module into threads for dynamically distributing its execution among system cores. The presented software mechanism solves efficiently sequential data dependencies and permits the parallel execution of the cutter radius compensation function. Tests following the proposed software architecture show the benefits in performance of the parallel algorithm against its serial variant under several scenarios.

Peer-review under responsibility of the scientific committee of 48th CIRP Conference on MANUFACTURING SYSTEMS - CIRP CMS 2015
doi:10.1016/j.procir.2015.12.021

This paper is organized as follows: In Section 2 a discussion is presented regarding works considered relevant to this research. This is followed by presenting the state-of-the-art concept of CNC systems in Section 3. Further, the designed parallel concept of an IPR module is described in Section 4. In Section 5 a case study of a software implementation is presented following the proposed parallel concept, a benchmarking analysis among different scenarios results comprising serial and parallel executions is reported. A discussion about the developed work is presented in Section 6. Finally, in Section 7, the paper closes with a summary and an outlook about this research.

2. Related Work

Few approaches address migration from single-core to multi-core platforms in automation applications whereas none of them focuses on CNC software partitioning at a lower level than tasks segmentation. All current available methods rely on static and semi-static tasks allocation on cores. One of the main downsides of this is the lack of flexibility whenever a different architecture is present. For instance, Intel Corporation and Beckhoff GmbH [10] propose a solution for tasks migration to parallel platforms based on static allocation from tasks to system cores. The designer has to define the arrangement manually and the software can be restructured when using a different platform. Likewise, Bregenzer and Hartmann [11] propose a method to generate a graph-based model of a CNC firmware looking at its runtime characteristics. Based on developer's expertise, optimal parallel designs can be proposed to meet a set of objectives. Although this approach provides a deep study of the firmware tasks interaction, it strongly depends on the computing platform utilized. In like manner, IBM Corporation [12] presents a model driven tool in order to ease the arbitrary assignation of tasks to different cores and to virtually execute the system to achieve the predicted performance gains. Changes on the parallelization strategy require little effort compared to other static methods. Another work using static parallelization is reported by Dadji et al. [13], they presented an approach for concurrently scheduling modules in a PC-based control system for parallel kinematic robots. This method is based on segmentation of tasks into modules. Tasks within different modules have no shared memory dependencies for allowing parallel execution on different system cores. A different approach based on dynamic parallelization at the algorithmic level is presented by Keinert et al. [14]. It has shown already the benefits of parallelization using a more flexible strategy than those using static allocation of tasks onto cores. Specifically the look-ahead module has been partitioned for concurrent execution on multi-core systems. Regarding other modules in a CNC system such as the IPR, Hong et al. [15] proposed a method to leverage its execution in order to be able to provide enough data to the next module in the data processing flow. The IPR is run in a separate thread independently from other modules in the system. Although this research is not focused on parallel computing, it shows the seriousness for the IPR to decode data opportunely.

3. Concept of CNC Systems

The architecture design of CNC systems is vendor specific. However, the concept does not differ substantially from one another. In this paper the CNC system components and their interaction are considered as presented by Suh et al. [16] (see Fig. 1).

A CNC system is mainly composed by a Man Machine Interface (MMI), a Programmable Logic Controller (PLC), and a Numerical Control Kernel (NCK). The MMI enables the interaction between the user and the CNC system. This interaction comprises machine status displaying, program edition, process parameter settings, among others. The PLC processes the system IO signals, tool changes, and in general it controls the machine behavior not related to the servo drives. The NCK realizes the functions related to the interpretation of the NC-program, trajectory generation, and position control.

Since the MMI and the PLC analysis is out of the scope of this study, these elements are not further considered. The NCK is composed by several modules which process data following a sequential flow. They are scheduled according to their priority and depending its nature their iteration times are defined. High priority functions present the shortest iteration times, followed by the middle priority tasks having larger cycle times than the aforementioned functions, and finally the low priority modules are executed in the remaining computation time.

The IPR module that is in the focus of this paper, decodes sequentially data blocks (hereinafter referred as blocks) specified in the NC-program. Depending on this information, process parameters, and machine specific data, it builds data structures for commanding system peripherals and for defining the tool path. Further modules of a CNC system are:

- Acceleration/Deceleration (Acc/Dec) module
- Rough Interpolation (IPO) module
- Mapping or transformation module
- Fine interpolation module (Fine IPO)
- Position Control module

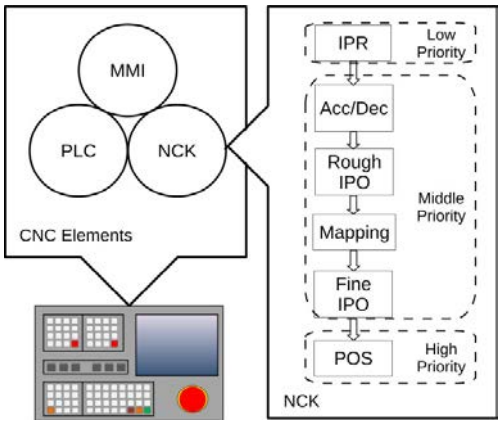

Fig. 1. CNC system components

4. Interpreter Parallel Concept

The proposed parallel architecture of an IPR (see Fig. 2) consists in several modules concurrently executed following a thread based approach.

The *Feeder* module runs on its own thread. It sequentially fetches blocks defined in the NC-program and stores them in the *stringBuffer*. Its execution starts before all the other modules in order to fill up the *stringBuffer* for initialization purposes. The *stringBuffer* itself is managed as a circular queue. It stores strings corresponding to the blocks defined in the NC-program. It can be thought as a 2-Dimensional array consisting of *g* rows and *n* columns, being *n* the same number of *Lexical Analzer* (LEX) threads. The sequence of blocks b_{read} retrieved by the *Feeder* in terms of the column *i* where the data is stored in the *stringBuffer* is given as follows:

$$b_{read} = i + k_s \cdot n; \qquad 0 \le k_s \le \frac{N - i - 1}{n} \tag{1}$$

- *i*: Column in the *stringBuffer* where the data is placed.
- *N*: Total number of blocks in a NC-program.
- *n*: Total number of LEX threads.

The term k_s is synchronized among all columns, it is incremented when a row in the *stringBuffer* is filled. Data contained in each column corresponds specifically to a certain LEX thread according to a unique ID number assigned to each LEX thread. The LEX module is split in n threads running concurrently. Each LEX thread parses data from its specific column in the *stringBuffer*, transforms it in a readable data structure to the system named *parsedStructure*, and stores the deciphered data in the so called *parsedBuffer*. The *parsedBuffer* holds the data after being decoded by the LEX module and also after being processed by the *Data Analyzer* (DA) module. This buffer consists in only one column and h rows. Given that the LEX threads run independent from each other, their indexes where to store data in the *parsedBuffer* are updated separately. They are incremented once a block is parsed and stored. These indexes must be restarted after reaching the last memory space in the *parsedBuffer* corresponding to each LEX thread called: idx_{ie}. Where sub-index *i* stands for the LEX thread ID, and sub-index *e* denotes the last index reachable by the LEX thread *i*. The sequence for updating these indexes is defined as follows:

$$idx_{i0} \mapsto idx_{i1} \rightarrow ... \mapsto idx_{ie} \mapsto idx_{i0} \mapsto ... \tag{2}$$

$$idx_{ik} = i + k_{pi} \cdot n; \qquad 0 \le k_{pi} \le \frac{k - i - 1}{n} \tag{3}$$

- *i*: LEX thread ID.
- *h*: Rows in parsedBuffer.
- *n*: Total number of LEX threads.

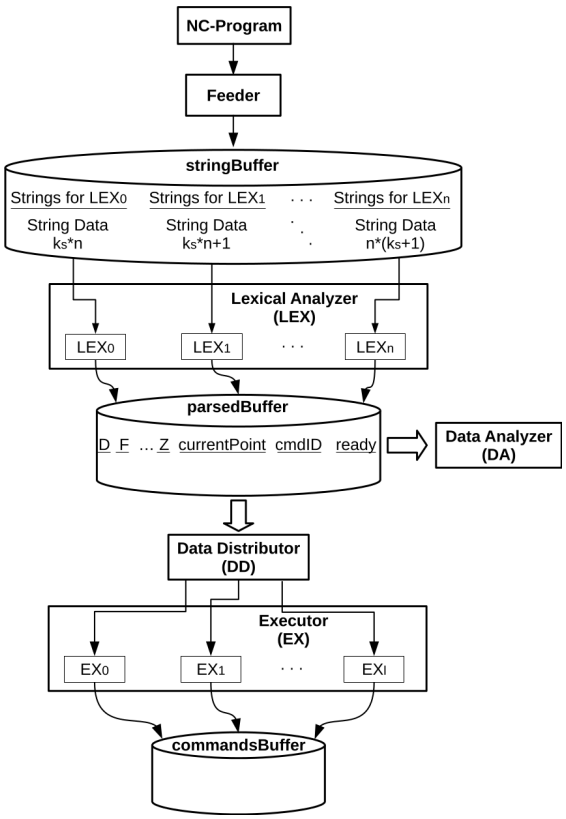

Fig. 2. Interpreter (IPR) parallel concept.

The DA module is executed on its own thread. It sequentially processes data from the *parsedBuffer*. This software component solves most of the sequential data dependencies within the IPR. Instructions which take effect throughout the whole process until another similar instruction replaces this setting are called modal instructions. The values defined by these instructions are tracked and modified by this module. Likewise, when increments in machine axes positions are specified in the NC-program instead of tool coordinates for defining the tool path, the DA module updates the current tool location. This value is held by a variable in the *parsedStructure* called *currentPoint*. Equally important, when the programmed tool path must be suited for matching the cutter radius dimensions (cutter radius compensation mode on), an extra segment might be added to the original block depending on the angle between the current and its previous block. The DA module realizes calculations in order to evaluate whether to add or not an extra segment and according to the result it assigns to each block a sequential number called *cmdID*. This number defines the index where data is going to be stored in the so named *commandsBuffer*.

The *Data Distributor* (DD) module shifts the already analyzed data to the *Executor* (EX) module. The amount of data passed remains static but it is adjusted in order to optimize the whole process execution speed. The EX module computes the

destination point of the tool and realizes the cutter compensation algorithm when it is active. It is executed on 1 threads running concurrently. Certain amount of data is passed by the DD module to each EX thread to be processed. Each EX thread stores immediately its computed result in the so called *commandsBuffer*. This buffer holds data delivered by all LEX threads. The information contained in this buffer serves for further processing such as velocity profile, and interpolation. The next module taking data from this buffer is specifically the *Acc/Dec*.

Following the proposed concept, an application comprising the Feeder, *stringBuffer*, LEX, and *parsedBuffer* has been devised. The thread-based parallel software has been developed using Pthreads Application Programming Interface (API). It specifies a Portable Operating System Interface (POSIX) providing a set of features to create, terminate, and synchronize threads [17]. Pthreads provides a mechanism for creating threads in a joinable state. This attribute lets thread resources, such as thread specific data, to be maintained in the system after the thread is terminated. These resources are reclaimed by the Operating System (OS) until the thread is joined to another thread. All threads in the developed application are created as joinable in order to facilitate code extendibility.

The software defines the parallelization strategy without setting any affinities from threads to system cores (unless specified). The OS allocates threads to the available cores and manages migration among them. This makes impossible to have always the same computing scenario. Hence, it is not possible to predict which thread is going to finish first its execution. In order to ensure the correctness of the algorithm the approach shown on Fig. 3 has been implemented.

The main thread is created when the process is started. All the other threads on the application are spawned from it. The *Feeder* thread is the first created from the main, it starts fetching blocks from the NC-program and storing them in the *stringBuffer*. After the *stringBuffer* is full or an initialization time has passed, the LEX threads are created from the main. The *Feeder* thread is joined to the main after all LEX threads have finished processing their corresponding data, and are joined to the main thread.

A thread in this application can be in three different states: *Active*, *Sleeping*, and *Zombie*. The first, is the state when all thread resources are utilized, and thread execution is taking place. The second, happens when a thread is suspended waiting for other threads to join it, this applies only to the main thread. The last, is reached when a thread finishes its execution but it is still not joined to any other thread.

5. Case Study

In this section the effectiveness of the proposed algorithm is investigated through comparison among several execution scenarios, this analysis includes a benchmarking between parallel versions against the serial variant. The application was tested on an Intel Core i3-2370M processor with 2.40GHz and

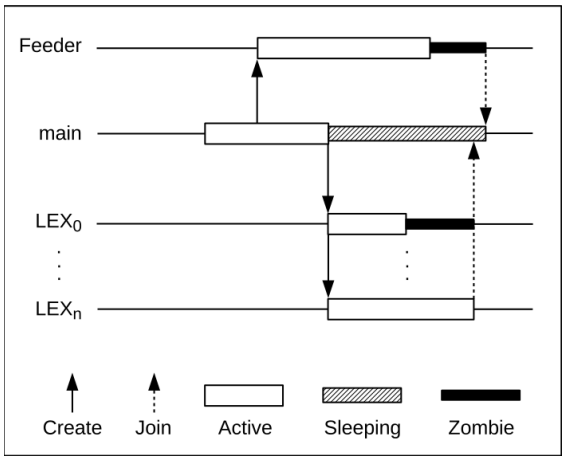

Fig. 3. Thread execution scheme.

4 logical processors. The Linux RT extension with the PREEMPT_RT patch was utilized in order to explore the system characteristics under an OS which reinforces determinism. As input to the software a NC-program was selected including 998 blocks.

Since this research strives to propose a dynamical parallelization strategy in which threads can run on any core and be migrated according to the need by the OS, the behavior of threads migration is explored (see Fig. 4). The cores on which threads were allocated for execution at each function specific iteration were tracked. A function iteration for the *Feeder* thread consists in the operations needed for fetching a block from the NC-program and placing it in the *stringBuffer*. For LEX threads, a function iteration comprises the steps for retrieving a block data from the *stringBuffer*, analyze its content, parse it, and store the result in the *parsedBuffer*. From Fig. 4 it may be seen that minimum migration of threads execution among system cores is present on a 4 cores system when 5 active threads are scheduled.

In order to deeply analyze the system behavior, 500 consecutive executions were performed at each test scenario. The time it takes for the LEX module to process the whole file is recorded at each program run. The realized experiments differ in the following parameters: Priority: Scheduling policy, parallelization strategy and number of threads.

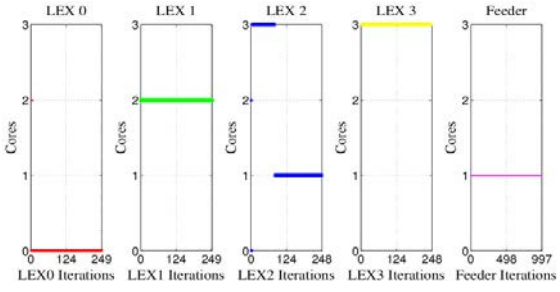

Fig. 4. Thread execution over system cores.

The priority defines all application threads priority in a range of 0-99. System processes compete also for resources within this priorities span. The software was tested with priority 0 (Non-RT constraints. It can be preempted by system processes. Several user space processes run at the same level.) and priority 90 (Considerable high RT constraints. Can only be preempted by few equal or higher priority system processes.).

The scheduling policy determines the algorithm utilized for dictating threads execution. In the performed experiments the Linux-provided default policies were used, namely SCHED_OTHER (It applies when no RT behavior is desired. It is based on equal timing execution for all threads.) and SCHED_FIFO (It implements a FIFO scheduling policy. It preempts any thread running under SCHED_OTHER policy. A thread under this policy runs until it completes its execution or it is preempted. These threads can be preempted by another higher priority thread or a new one introduced under the same priority (if it is runnable). In the realized tests this policy is set for threads running at priorities above 0.).

The parallelization strategy defines how threads are allocated to system cores for execution. For benchmarking purposes not only dynamic parallelization is used but also static threads allocation onto system cores. Using static parallelization each of the spawned threads from the main is executed for all the long the process lasts on a specific processor and is never migrated. Hereafter this method is named as *ATT n Threads*, where *n* denotes the total number of active threads. Using dynamic parallelization the OS determines the system core where to execute each thread and the migrations required. From now onwards this method is referred as *DET n Threads*, where *n* has the same meaning as above.

The thread number refers to the total number of active threads. When referring to active threads, the main thread is not counted because it is most of the time in a Sleeping mode. For the serial version only 1 LEX and 1 *Feeder* thread are created and are both allocated to run on CPU0. For the parallel executions 4 and 8 active threads algorithms were tested. When testing *DET n* versions the OS allocates threads onto execution whereas for *ATT n* algorithms the arrangements in Tables 1 and Table 2 were tried.

Table 1. *Att 4 Threads* configuration.

CPU0	CPU1	CPU2	CPU3
Feeder	LEX0	LEX1	LEX2

Table 2. *Att 8 Threads* configuration.

CPU0	CPU1	CPU2	CPU3
Feeder	LEX0	LEX1	LEX2
LEX3	LEX4	LEX5	LEX6

Statistical data obtained by the various experiments is shown on Table 3, a graphical version in Fig. 5 and Fig. 6. In each test scenario is considered worthwhile to know the mean, minimum, and maximum execution times. Furthermore the standard deviation σ gives us a way to measure the dispersion among the run times at each scenario. In order to quantify the benefits of the proposed solution a performance gain factor is determined for all parallel versions against its serial variant with the same test parameters, this gain is calculated in terms of average execution times (from now onwards it is referred as performance gain).

The gathered results supply valuable information from which we can comment the following: The performance gain obtained by the parallelized algorithm depends mainly on 2 factors: priority assigned to the application threads, and ratio between number of threads and system cores. The *Det n Threads* versions show very close results to their *Att n Threads* counterparts when setting high priorities and equal number of threads as cores on the computing platform. Under these conditions the resulted performance gains were 4.205, and 4.225 for the *Det 4 Threads* and *Att 4 Threads* executions respectively. Without reinforcing determinism the timing results homogenity of the *Det n Threads* algorithms is seriously degraded. When setting priority 90 to the software threads, both parallel versions *Det 8 Threads* and *Att 8 Threads* presented slower execution times than their 4 threaded variants, especially the performance gain for the *Det 8 Threads* algorithm was considerably decremented getting only a gain factor of 2.621. In general the application behavior is better when splitting up the execution in the same number of threads as available processing cores than when running the software over 8 threads.

Table 3. Recorded experiment results.

	Mean (ms)	Min (ms)	Max (ms)	σ (ms)	Gain
Non RT. Serial	9.265	9.144	13.172	0.262	-
Non RT. Det 4 Threads	4.841	2.429	10.871	1.366	1.914
Non RT. Att 4 Threads	2.629	2.486	8.708	0.455	3.524
Non RT. Det 8 Threads	9.314	4.436	29.802	1.521	0.995
Non RT. Att 8 Threads	6.995	5.116	19.705	2.7	1.3245
Priority 90. Serial	10.643	10.105	11.383	0.176	-
Prio 90. Det 4 Threads	2.531	2.37	4.47	0.111	4.205
Prio 90. Att 4 Threads	2.519	2.437	4.96	0.15	4.225
Prio 90. Det 8 Threads	4.061	3.937	5.785	0.112	2.621
Prio 90. Att 8 Threads	2.328	2.535	4.52	0.13	4.0498

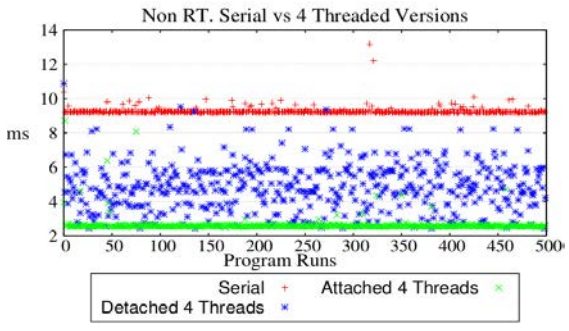

Fig. 5. Experiments execution time results part 1.

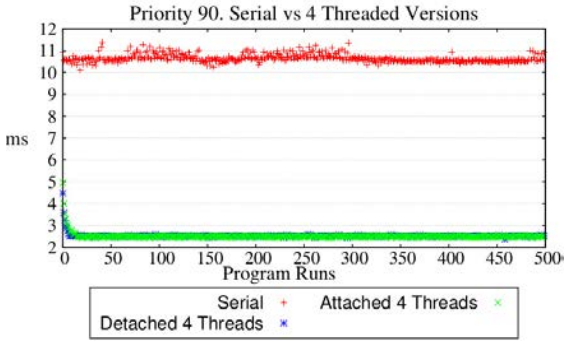

Fig. 6. Experiments execution results part 2.

6. Discussion

The reported results show the possibility of getting advantage of multi-core systems by partitioning CNC software modules following a dynamic parallelization strategy. The developed functions were tested isolated from the complete NCK. Admittedly the integration of the devised software in a complete NCK is desired in order to test the effect of threads competing for system resources, interaction among software components, and heavier computational load. Nevertheless, the work presented in this paper is a step forward towards the goal of conceiving a parallel NCK architecture. Furthermore, the obtained results bring us an approximation to the real execution scenario, and let the door open for further research on this lead.

7. Summary and Outlook

Parallel programming has been addressed towards CNC software partitioning. The devised concept does not rely on static parallelization, it can be split on any number of threads letting the OS to allocate and migrate threads among cores for execution. The proposed design is based on the study of state of the art NCK modules allowing concurrent execution of an IPR and considering also the cutter radius compensation function. Following the proposed design the Feeder and the LEX modules were implemented, being the *latter* partitioned into 4 and 8 threads for benchmarking purposes. Recorded timing results on a 4 processing cores platform showed considerable performance gains of the suggested approach against its serial variants. It turned out that priority at which the application threads are run, and ratio between number of active threads and available system cores influence on the performance enhancement.

This research could be directed towards several paths. The complete implementation of a parallel NCK following a dynamic parallelization strategy will bring the benefits of multi-core processors to CNC taking advantage also of the rapid computing technology evolution. So far the presented hypothesis has been proved on the IPR module, and the look ahead function [14]; interpolation algorithms can also benefit of this strategy resulting in higher machining accuracy.

Acknowledgements

The work presented in this paper was funded by the DFG in the project PANAMA.

References

[1] Koren Y. Control of machine tools. Journal of Manufacturing Science and Engineering, 119(4B):749{755, 1997.
[2] Erkorkmaz K and Altintas Y. High speed cnc system design. part iii: high speed tracking and contouring control of feed drives. International Journal of Machine Tools and Manufacture, 41(11):1637{1658, 2001.
[3] Sencer B, Altintas Y, and Croft E. Feed optimization for five-axis cnc machine tools with drive constraints. International Journal of Machine Tools and Manufacture, 48(7):733{745, 2008.
[4] M. Keinert, A. Huf, and A. Verl. Continuous parameter calculation as control internal task (kontinuierliche kennwertberechnung als steuerungsinterne task). Proceedings of SPS/IPC/DRIVES 2010, Nuremberg, Germany, pages 361{369, 2010.
[5] D. Scheifele and G. Pritschow. Methods for the avoidance of surface defects and collisions during milling operation for the lot size one. In Proceedings of CIRP 2nd International Conference on Process Machine Interactions, Vancouver, Canada, 2010.
[6] A. Verl, E. Westkaemper, E. Abele, A. Dietmair, J. Schlechtendahl, J. Friedrich, H. Haag, and S. Schrems. Architecture for multilevel monitoring and control of energy consumption. In Glocalized Solutions for Sustainability in Manufacturing, pages 347{352. Springer, 2011.
[7] M. Keinert and A. Verl. System platform requirements for high-performance cncs. In Proceedings of FAIM 2012 22nd international conference on fexible automation and intelligent manufacturing, Helsinki, Finland, 2012.
[8] S. A. Guccione. Hardware/software trade offss in multicore architectures. 2008.
[9] P. Gepner and M. F. Kowalik. Multi-core processors: New way to achieve high system performance. In Parallel Computing in Electrical Engineering, 2006. PARELEC 2006. International Symposium on, pages 9-13. IEEE, 2006.
[10] N.N. Simplifying multi-core migration in automation applications. White Paper, Intel/Beckhoff 2008.
[11] J. Bregenzer and J. Hartmann. An approach towards automation firmware modeling for an exploration and evaluation of efficient parallelization alternatives. In Parallel Computing in Electrical Engineering (PARELEC), 2011 6th International Symposium on, pages 13{18. IEEE, 2011.
[12] N.N. Model driven development - simplifying multicore systems deployment. White Paper, IBM Corporation, 2009.
[13] Y. Dadji, J. Maass, and H. Michalik. Parallel task processing on a multicore platform in a pc-based control system for parallel kinematics. In proceedings of the 6th International Conference on Computing, Communications and Control Technologies (CCCT), Orlando (FL), USA, 2008.
[14] M. Keinert, B. Kaiser, A. Lechler, and A. Verl. Analysis of cnc software modules regarding parallelization capability. Proceedings of 24th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM), San Antonio, Texas., 2014.
[15] H. Hong, D. Yu, X. Zhang, and L. Chen. Research on the data hungry problem in cnc system based on the architecture of real-time multitask. In Computer Research and Development (ICCRD), 2011 3rd International Conference on, volume 2, pages 103{108. IEEE, 2011.
[16] S.-H. Suh, S.-K. Kang, D.-H. Chung, and I. Stroud. Theory and design of CNC systems. Springer, 2008.
[17] B. Chapman, G. Jost, and R. Van Der Pas. Using OpenMP: portable shared memory parallel programming, volume 10. MIT press, 2008.