Jens von Aspern

Turbo für die Ablaufsprache

Ablaufsprachen wie SFC wird gelegentlich eine schlechte Performance nachgesagt, was die Ausführung von Steuerungsprogrammen betrifft. Deutlich steigern lässt sich die Geschwindigkeit mit Verfahren, die ursprünglich zur Optimierung von Petrinetzen entwickelt wurden.

Jens von Aspern ist Fachbuchautor und beschäftigt sich seit 10 Jahren mit der Anwendung von Petrinetzen in der Steuerungstechnik

Ablauf- und zustandsorientierte Methoden haben sich in der Automation mittlerweile etabliert und treten zukünftig noch stärker in den Vordergrund. Beispiele hierfür sind die IEC 61131-Sprache SFC (Sequential Function Chart), Schrittketten, Graph 7, High Graf, ECC (IEC 61499) oder auch Zustandsmaschinen der Motion Control- Spezifikation der PLCOpen - um nur einige zu nennen. Ein weiterer Trend lautet: weg von zentralen hin zu verteilten Steuerungen mit dem Ziel, die Steuerung näher an den Prozess heranzuführen. Dies erfordert hochintegrierte, kommunikative Steuerungen, die aus Kostengründen jede Ressource optimal ausnutzen müssen. Auch wenn die Leistungsfähigkeit der Hardware bei sinkenden Preisen ständig steigt, sollte immer wieder überprüft werden, ob nicht auch eine verbesserte Auslegung der Software zu einem Performance-Gewinn beitragen kann - insbesondere bei Software von der Stange wie beispielsweise den IEC 61131-Compilern, denn sie erzeugen letztlich die Anwendungen.

Gerade für kleine und dezentrale Steuerungen, sowie Steuerungen in Serienmaschinen bietet eine Hochgeschwindigkeits-Codierung interessante Möglichkeiten, nicht nur die Hardwarekosten zu minimieren sondern zudem zusätzliche funktionale Eigenschaften in die Steuerung zu integrieren, die ein Projekt unter Umständen überhaupt erst realisierbar machen. Auf welcher Software-Ebene ist aber mit der Optimierung im Idealfall anzusetzen?

Die Tatsache, dass die Zukunft den offenen Systemen gehört, setzt zunächst voraus, dass eine Portierung der IEC 61131-Sourcen auf Systeme verschiedener Hersteller relativ einfach möglich sein muss. Sind die Optimierungen bereits in der Source enthalten, anstatt in der SPS-Runtime, ist nach einer Portierung ein ähnliches Verhalten zu erwarten. Zwar bietet eine Implementierung der Optimierungsstrategien innerhalb der Laufzeitumgebung einen kleinen Geschwindigkeitsvorteil gegenüber der Source- Optimierung; es ist jedoch mit erheblichen Unterschieden bei Portierungen auf Systemen zu rechnen, die nicht über eine derartige Runtime verfügen. Abwärtskompatibilität ist damit jedenfalls nicht erreichbar. Ferner ist ein Update des Runtimesystems häufig aufwendiger, als den Quellcode mit einem upgedateten Compiler zu übersetzen. Um ein geeignetes Verfahren zur Optimierung auf Source-Code-Ebene entwickeln und umsetzen zu können, ist zunächst zu identifizieren, woraus die Verluste der Bearbeitungsgeschwindigkeit resultieren.

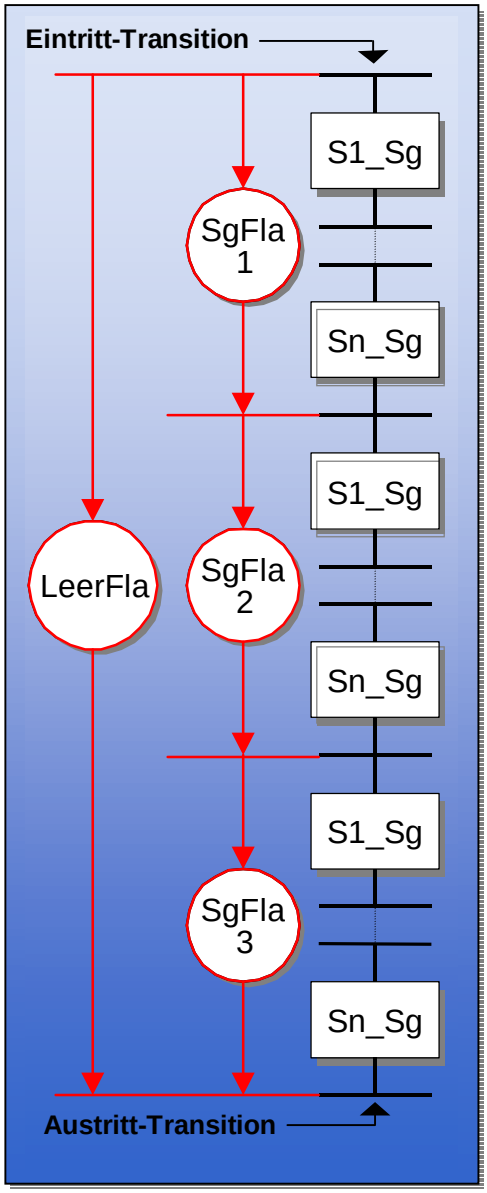
Die Geschwindigkeits-Bremsen

Verknüpfungsorientierte Lösungen – hierzu zählt die boolesche Algebra - müssen von der SPS in jedem Zyklus im allgemeinen Zeile für Zeile bearbeitet werden, damit jeder Signalwechsel an den Eingängen eines Netzwerkes auch zum gewünschten Verhalten der Ausgänge führt. Ablauforientierte Lösungen hingegen bilden Kausalstrukturen ab, die dafür sorgen, dass eine Signal- beziehungsweise Zustandsänderung nur in Zusammenhang mit einem aktiven Schritt zu Veränderungen der Ausgänge (boolesche Aktion) führen kann. Ausgenommen hiervon sind Signale, die in aktiven Aktionen unmittelbar gebraucht werden und so direkt auf die Ausgänge wirken. Lässt ein Compiler bei der Code-Erzeugung jedoch die Kausalstrukturen unberücksichtigt, ist das Programm wie eine verknüpfungsorientierte Lösungen zu bearbeiten. Im folgenden wird ausschließlich der Zustandswechsel von Schritten betrachtet.

Aus den genannten Zusammenhängen lässt sich folgern, dass eine Schrittkette - wie in *Bild 1* schwarz dargestellt - nur bearbeitet werden muss, wenn sie über mindestens einen aktiven Schritt verfügt, da nur in diesem Fall ein Zustandswechsel möglich ist. Das heißt: Bleibt die Bearbeitung aus, weil es keinen aktiven Schritt gibt, lässt sich die Verarbeitungszeit der gesamten Kette einsparen. Das Potential einer möglichen Performancesteigerung hängt dabei unmittelbar von der „Länge“ der Kette ab, sprich von der Summe der Befehlszeiten, die zur Kettencodierung sowie zur Koordinierung der Ausführung nötig sind. Grundsätzlich gilt: je länger eine unverzweigte Kette ist, desto größer der potentielle Leistungszuwachs.

Soviel zur Theorie: Gängige Softwaretools bilden ablauf-orientierte Sprachen allerdings häufig so ab, als handle es sich um ein verknüpfungs-orientiertes Modell und verschwenden somit einen Teil der verfügbaren Leistung. Mit anderen Worten: Die Applikation ermittelt zur Laufzeit aufwendig ein Ergebnis, welches bereits von vornherein bekannt ist, nämlich dass keine Zustandsänderung erfolgen kann.

Bild 1 Ausschnitt aus einer unverzweigten AS-Kette, deren Bearbeitung durch Kettenflags koordiniert und damit optimiert wird.



Aber auch wenn es einen aktiven Zustand innerhalb einer Schrittkette gibt, stellt sich die Frage, ob die Kette über die gesamte Länge zu bearbeiten ist, um das gewünschte Verhalten zu erzeugen. Zustandsänderung sind nur durch schaltfähige Transitionen

möglich. Diese schalten nur, wenn die gesamte Weiterschaltbedingung erfüllt ist. Dazu gehört, dass alle ihre Vorgängerschritte aktiv sind. Ergo: Es sind nur die Transitionen zu bearbeiten, die möglicherweise schalten könnten. Die Anzahl nicht schaltfähiger Transitionen überwiegt jedoch im allgemeinen bei weitem. Auf der Grundlage dieser Erkenntnis lässt sich die Kette in Teilsegmente gliedern, die wiederum nur dann zu bearbeiten sind, wenn sich aktive Schritte innerhalb eines Segmentes befinden.

Optimierte Kettenbearbeitung

Es liegt nahe, eine Optimierung eines Ablaufmodells auch mit den Mitteln des Modells zu implementieren. Dadurch lassen sich bereits implementierte Funktionen des Compilers nutzen, eine manuelle Realisierung ist gegebenenfalls möglich und vor allem lässt sich so Normkonformität sowie Auf- und Abwärtskompatibilität herstellen. Zur Realisierung der Optimierung ist folgendes Problem zu lösen: Wie lässt sich mit einem geringen Overhead feststellen, welche Transitionen in einem Netz schalten könnten beziehungsweise welche Schritte aktiv sind?
Das Bild zeigt, wie das Verhalten einer Kettenoptimierung mittels Petrinetzen (rot gekennzeichnet) mit einfachen Mitteln gestaltet werden kann: Sobald die Aktivierung des ersten Schritts einer Kette oder eines Segmentes (zum Beispiel S1_Sg1) durch die Transition (Eintritttransition) erfolgt, werden zusätzlich ein oder mehrere Ketten-Flags gesetzt (LeerFlag, SgFlag1). Die Rücksetzung des Flags erfolgt sobald der letzte Schritt der Kette oder des Segmentes durch die Austritttransition wieder deaktiviert wird. Dabei unterliegt die Kette beziehungsweise das Segment nur der Bearbeitung, sofern das entsprechende Flag gesetzt ist.
Im dargestellten Beispiel wird mit zwei Flags gearbeitet: einem LeerFlag, das angibt, ob es in der gesamten Kette aktive Schritte gibt, und jeweils einem Flag (SgFlag1...n), das angibt, ob ein Segment über einen aktiven Zustand verfügt. Auf das LeerFlag lässt sich gegebenenfalls verzichten; es dient dazu, dass im Falle einer komplett deaktivierten Kette nur eine Abfrage anstelle von dreien (SgFlag1...n) erfolgen muss. In diesem Fall lässt sich fast die gesamte Bearbeitungszeit der Kette einsparen. Unter der Annahme, dass alle Segmente gleiche Bearbeitungszeit in Anspruch nehmen, verkürzt sich die Bearbeitungszeit bei Vorhandensein eines aktiven Schrittes innerhalb der Kette nahezu auf ein Drittel.
Eine Erweiterung dieses Prinzips führt zu einem Optimierungsnetz, das die Ausführungsverwaltung des eigentlichen Steuerungsnetzes übernimmt. Dieses einfache und zugleich effiziente Verfahren lässt sich mit Mitteln realisieren, welche die Anweisungsliste (AWL) zur Verfügung stellt.

Bis zum Faktor 20 schneller

Codeoptimierung wird künftig auch bei hohen verfügbaren Prozessorleistungen nicht zuletzt wegen des ständig steigenden Umfanges der Applikationen an Bedeutung gewinnen. Bis dato wurden insgesamt neun Verfahren entwickelt und patentiert. Diese lassen sich kombinieren und ermöglichen so die Erstellung von Programmen, die bis zum Faktor 20 schneller sind als noch vor der Optimierung. Die Methoden der Codierung nutzen dabei sowohl topologische als auch dynamische Netzeigenschaften für die Erzeugung von Hochgeschwindigkeitsnetzen. Entscheidender Vorteil dieser Verfahren ist, dass Transitionen, abhängig von der Netzdynamik, die Abarbeitung entsprechender Codesegmente selbständig koordinieren und so auf das Notwendigste beschränken. Eine nachträglich implementierte Kettenoptimierung wirkt sich generell nicht negativ auf ältere Applikationen aus und kann durch einen neueren Compilerlauf übernommen werden. Somit ist Auf- und Abwärtskompatibilität gegeben. Wer viel mit ablauforientierten oder zu-standsorientierten Sprachen arbeitet, sollte seinen Hersteller in jedem Fall nach der exakten Arbeitsweise seines Systems fragen.

Nähere Informationen: JvAspern@gmx.de

Die Implementierung

Zur Integration der Verfahren sind die existierenden Compiler von den Herstellern zu erweitern. Auch eine Implementierung in das Runtime- System ist denkbar. Eine Alternative wäre, basierend auf der Textform der SFC eine Transformation in AWL (oder ggf. auch in ST (Strukturierter Text)) durch ein herstellerneutrales Tool. Compiler, die ein Zertifikat der PLCOpen für SFC und AWL bzw. ST besitzen, eignen sich hierfür besonders gut.
Die Auswahl geeigneter Verfahren für ein Netz kann weitgehend automatisch geschehen. Dies gilt für die topologischen Netzeigenschaften. Die Feinoptimierung nutzt die dynamischen Eigenschaften und muss im Dialog erfolgen.
Es existieren zwei Gruppen von Verfahren. Die Netzoptimierung unterliegt der ganzheitlichen Betrachtung eines (Teil-)Netzes, während die Elementoptimierung sich den einzelnen Netzelementen (Transition, Aktion und Platz) widmet. Die Komposition der Verfahren, die jeweils spezielle topologische und dynamische Netzeigenschaften nutzen, führt zu einer bedarfsgerechten Optimierungsstrategie.

Verfahren der Netzoptimierung

» Kettenflag

Verwaltung der Bearbeitung von (Teil-) Netzen erfolgt mittels Flags, deren Kontrolle durch Ein- und Austrittstransitionen erfolgt.

» Segmentierung

Strukturelle Verfeinerung des Kettenflagverfahrens. Die Komposition beider führt zu einer mehrstufigen Ausführungsstruktur.

» Dispatcher

Zur Ausführung gelangen nur Transitionen, die in einer Liste als potentiell schaltfähig verwaltet sind. Die Transitionen sind für die Aktualisierung der Liste verantwortlich.

» Einmarkendispatcher

Reduziert die Liste des Dispatchers im wesentlichen auf eine Variable. Eignung für (Teil-)Netze mit maximal einer Marke.

» Dynamischer Sprungverteiler

Umsetzung des Einmarkendispatcher mittels Anweisungen, die dynamische Sprungziele unterstützen (nicht IEC 61131-konform).

Verfahren der Elementoptimierung

» Linear

Klassische Codierung; eignet sich für sehr einfache Transitionen.

» Splitting

Aufsplittung der Weiterschaltbedingung in mehrere Teilbedingungen. Nach Bearbeitung jeder Teilbedingung wird fest-gestellt, ob die Transition weiterhin bearbeitet werden muss (potentiell schaltfähig).

» Sprungverteiler

Jede Transition wird abhängig von einem Flag bearbeitet.

» Einmarkensprungverteiler

Wie Sprungverteiler, jedoch unter dem Aspekt, dass nur eine Marke im (Teil-)Netz auftreten kann.

» Feinoptimierung

Abhängig von dynamischen Eigenschaften erfolgt die Neuordnung der Teilbedingungen des Splittings oder