Parallelisierung von Embedded Realtime Systemen: Probleme und Lösungsstrategien in Migrationsprojekten

Marwan Abu-Khalil

Siemens AG, Energy Automation
13629 Berlin, Germany
marwan.abu-khalil@siemens.com

Abstract: Dieser Artikel extrahiert Erfahrungen aus einer Reihe erfolgreicher sowie gescheiterter industrieller Parallelisierungsprojekte, bei denen Embedded Realtime Systeme von Single-Core CPUs auf Multi-Core SMP-Plattformen portiert wurden. Die Kernthese des Vortrages lautet, dass die Parallelisierung von Embedded Realtime Systemen spezifischen Herausforderungen gegenübersteht, die bei anderen System-Klassen, wie Server- oder Desktop-Software, nur eine untergeordnete Relevanz haben. Der Artikel analysiert und kategorisiert diese spezifischen Herausforderungen. Als Resultat werden allgemeingültige Herangehensweisen vorgeschlagen, die zu erfolgreicher Parallelisierung im Embedded-Bereich führen.

1 Einleitung

Heutige Embedded Systeme sind überwiegend als Multi-Thread Systeme auf einer Single-Core CPU realisiert. Dies prägt die gesamte Software-Architektur und insbesondere das Nebenläufigkeitskonzept, welches sich daher einer Migration auf eine Multi-Core CPU mit einem SMP Betriebssystem widersetzt. Eine SMP-Parallelisierung wird somit ein riskantes Unterfangen, bei dem das Systemverhalten komplett verändert wird. Aufgrund der begrenzten Taktratensteigerung aktueller CPUs und der Allgegenwart von Multi-Core CPUs ist jedoch die Embedded-Welt gezwungen, diese über Jahre gewachsenen Systeme auf Multi-Core Plattformen zu portieren. Typische Software-Architekturen von Embedded Realtime Systemen basieren auf Annahmen über das Scheduling-Verhalten auf einer Single-Core CPU und verwenden daher oftmals Konzepte wie die implizite Synchronisation durch Interrupt-Locks oder die Steuerung zeitlicher Abläufe durch Realtime-Prioritäten. Diese Konzepte lassen sich jedoch auf SMP Systemen nicht nutzen. Embedded Systeme stehen daher vor besonderen Herausforderungen bei der Multi-Core Portierung. Dieser Artikel analysiert und kategorisiert diese spezifischen Probleme und präsentiert allgemeine Maßgaben für die erfolgreiche SMP-Parallelisierung von Embedded Realtime Systemen.

Der Artikel basiert auf der Analyse beispielhafter erfolgreicher sowie gescheiterter Projekte. Die Auswahl der Systeme spiegelt die typischen und verbreiteten Software-Architekturen im Feld der Embedded Realtime Systeme wider.

Die These über die spezifischen Schwierigkeiten der Parallelisierung von Embedded Systemen wird in Kapitel 2 dargelegt. In Kapitel 3 werden drei Beispielprojekte präsentiert: Die erfolgreiche Parallelisierung eines Systems aus der Energie-Automatisierung, sowie als Gegenpol zwei teilweise gescheiterte Projekte aus dem Energie- bzw. dem Telekommunikationsumfeld. An diesen Beispielen werden Erfolgsrezepte sowie Fallstricke und Risiken solcher Parallelisierungen erkennbar. In Kapitel 4 werden die spezifischen Probleme analysiert und kategorisiert und die jeweiligen Lösungsvorschläge erarbeitet. Kapitel 5 fasst die Lösungsansätze zusammen.

2 These: Probleme der Embedded Realtime Parallelisierung

Die Parallelisierung von Embedded Realtime Systemen im Zuge der Portierung von Single-Core auf Multi-Core Hardware scheitert oftmals an den im Embedded-Bereich etablierten Software-Architekturen. Daher sind spezifische Herangehensweisen erforderlich, um Embedded Systeme erfolgreich zu parallelisieren. Zwei Problembereiche sind dabei ausschlaggebend:

1. Architekturen von Embedded Systemen, die über Jahre auf Single-Core CPUs entwickelt wurden, basieren oft auf Annahmen über das Scheduling-Verhalten, die implizit voraussetzen, dass die Software sequentiell auf einer Single-Core CPU abgearbeitet wird.
2. Im Embedded-Umfeld sind aggressive Optimierungen und eine hardwarenahe Denkweise weit verbreitet. Abstraktionen von Betriebssystemen oder anderen Laufzeitumgebungen werden oftmals aus Performance-Gründen umgangen.

Diese Problembereiche haben folgende technische und nicht-technische Aspekte:

- Embedded Systeme verwenden, anders als Desktop- oder Server-Software, in hohem Maße implizite Synchronisation durch Interrupt-Locks, diese lässt sich jedoch in einem SMP-System nicht nutzen (siehe 4.1).
- In Realtime Systemen wird Synchronisation oft durch Prioritäten von Threads realisiert, dies lässt sich nicht auf SMP-Systeme übertragen (siehe 4.2).
- Ein typischer Embedded-Programmierer möchte zeitliche Abläufe seines Systems in allen Details kontrollieren, oft wird dafür das prioritätsbasierte Scheduling missbraucht. Dies widerspricht jedoch der Philosophie eines SMP Betriebssystems (siehe 4.3).
- Die Denkweise von Embedded-Programmierern ist oftmals sehr hardwarenah. Dies führt zu riskanten Optimierungen und somit zu schwer zu findenden Fehlern bei der SMP-Portierung (siehe 4.4).

3 Beispiel-Projekte

Hier führe ich exemplarisch drei Beispiele erfolgreicher sowie gescheiterter Projekte an, um meine These zu untermauern. Die unterschiedlichen Parallelitätskonzepte, die in diesen Projekten verfolgt wurden, werden jeweils in einem abstrakten Diagramm dargestellt, um sie mit einer „idealen“ SMP-Parallelisierung vergleichen zu können.

3.1 Projekt 1: Hard-Realtime System zum Schutz von Stromverteilnetzen

Dieses auf dem RT-OS (Realtime-Betriebssystem) VxWorks basierende System, das der Kontrolle und dem Schutz von Stromnetzen dient, wurde von einem Single-Core PPC auf einen ARM Dual-Core (Altera Cyclone V SoC) portiert. Ziel der Portierung war eine signifikante Steigerung der Performance. Das System war schon vor der Portierung in viele VxWorks Tasks (vergleichbar mit Threads) strukturiert, die nach strengen Prioritätsvorgaben geschedult wurden. Zwei Herausforderungen standen bei der Migration im Vordergrund:

1. Ein weitgehendes Erhalten des Laufzeitverhaltens des vorhandenen Systems, um das Risiko der Migration zu begrenzen.
2. Das Ersetzen der impliziten Synchronisation durch explizite Synchronisation.

Konzept zu Anforderung 1:

Situation: Das System besteht aus einem IO-Thread („Input-Output“), der zyklisch Daten aus der Außenwelt abholt und im System verteilt, sowie einer Gruppe von BL-Threads („Business-Logik“), in denen Algorithmen ablaufen, die diese Daten verarbeiten. Die Kommunikation zwischen IO-Thread und BL-Threads erfolgt über Ringbuffer. Das Nebenläufigkeitsmodell im Single-Core Fall stellt sicher, dass die Algorithmen in den BL-Threads in einer fest vorgegebenen Reihenfolge nach jedem IO-Zyklus ablaufen. Dieser Ablauf ist auf Basis der Prioritäten des Realtime-OS organisiert.

Problem: Auf einem SMP-System kann durch Prioritäten diese Ablaufreihenfolge nicht mehr sicher gestellt werden, weil Threads unterschiedlicher Priorität gleichzeitig auf verschiedenen CPUs laufen können („CPU“ und „Core“ sind im Folgenden stets gleichbedeutend).

Lösung: Die Anforderung, das Laufzeit- und Scheduling-Verhalten möglichst wenig zu verändern, hat zu einer Software-Architektur geführt, die weite Teile des Systems quasi wie auf einem Single-Core ablaufen lässt. Alle BL-Threads sind an Core-0 gebunden, dadurch verhalten sich diese Threads untereinander so, als liefen sie auf einem Single-Core. Das Laufzeitverhalten der Applikationslogik bleibt damit weitgehend unverändert. Nur der CPU-hungrige IO-Thread wird an den zweiten Core gebunden. Insgesamt führen diese Maßnahmen zu einer „sanften“ Migration, mit relativ geringem Umbau im Code sowie einer beschränkten Veränderung des Laufzeit-Verhaltens, da lediglich der Datenaustausch zwischen dem IO-Thread und den BL-Threads neu organisiert werden muss.

Konzept zu Anforderung 2:

Situation: In Embedded Systemen, die auf Single-Core CPUs laufen, ist das sogenannte implizite Locking durch das Unterdrücken von Interrupts eine weit verbreitete, etablierte und effiziente Synchronisationsstrategie (siehe 4.2). Im Ausgangsystem wurden Critical-Sections an vielen Stellen durch die entsprechenden VxWorks APIs für Interrupt-Locks oder Task-Locks synchronisiert.

Problem: Auf einem SMP-System erzwingen Interrupt-Sperren keinen gegenseitigen Ausschluss (siehe 4.2).

Das Zielsystem verwendet VxWorks 6.9 in einer SMP-Ausprägung. Hier sind diese impliziten Synchronisationsmechanismen auch in der OS-API nicht mehr verfügbar. Die Empfehlung des Betriebssystemherstellers lautet, Interrupt-Locks durch Spinlocks zu ersetzen und Task-Locks durch Semaphore zu ersetzen. Eine durchgängige Befolgung dieser Vorgabe hätte jedoch folgende Konsequenzen: Der massive Einsatz von Semaphoren hätte das System stark verlangsamt, da der Aufruf eines Semaphors ein vergleichsweise teuer Betriebssystem-Aufruf ist. Der massenhafte Einsatz von Spinlocks auf dem Dual-Core hätte das erreichbare Maß der Parallelisierung unnötig stark begrenzt, da ein Spinlock im Wartezustand die CPU nicht freigibt.

Lösung: Es war somit keine schematische Ersetzung von impliziten durch explizite Synchronisationsmechanismen möglich. In diesem Projekt wurde daher für jede einzelne implizite Critical-Section untersucht, wie diese effizient und sicher auf das SMP-System portiert werden kann. Dabei standen die folgenden Optionen zur Verfügung, für die hier jeweils ein Beispiel genannt wird:

1. Semaphor: Die Kommunikation zwischen Threads auf unterschiedlichen Cores ist durch Semaphore synchronisiert.
2. Spinlock: Die Synchronisation zwischen Algorithmen der BL-Threads ist durch Spinlocks realisiert. Dies kann als problematische Low-Level Optimierung gesehen werden, weil hier die Eigenschaft der Spinlock-Implementierung ausgenutzt wurde, sich Core-Lokal wie ein Interrupt-Lock zu verhalten.
3. Lockfreie Datenstruktur: Der Datenaustauch zwischen dem IO-Thread und den BL-Threads wird über eine lockfreie Ringbuffer-Implementierung realisiert, da dies der performancekritischte Teil des Systems ist. Bemerkenswert ist, dass aus Performancegründen selbst die vom CPU-Hersteller geforderten Memory-Barriers teilweise nicht verwendet werden, weil in Tests kein Fehlverhalten nachgewiesen werden konnte. Auch dies ist ein Beispiel für riskante Mikro-Optimierungen in Embedded Systemen.

Bewertung: In einem „idealen“ SMP-Konzept verteilt der Scheduler des Betriebssystems alle Threads frei über alle vorhandenen CPUs, um zu einer optimalen Auslastung zu kommen. Im Vergleich dazu ist das Nebenläufigkeitskonzept dieses Projektes mit wesentlichen Einschränkungen verbunden. Der OS-Scheduler ist bezüglich der Zuteilung der verschiedenen CPUs außer Kraft gesetzt. In der folgenden schematischen Graphik werden die Abweichungen vom Ideal deutlich gemacht:

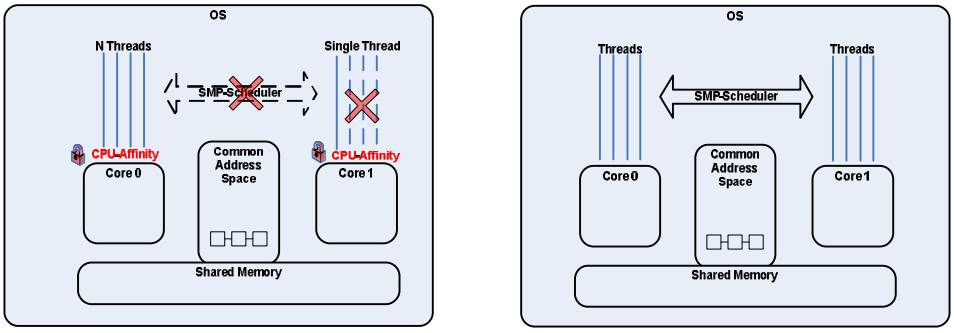

Abb. 1: Projekt 1 (links) im Vergleich zum "idealen SMP" (rechts)

Das Projekt kann bezüglich des Parallelitätskonzeptes daher nur als „teilweise erfolgreich“ bewertet werden. Die einfache Migration wurde mit folgenden gravierenden Defiziten in den nichtfunktionalen Eigenschaften der Architektur erkauft:

1. Die starre Bindung von Threads an CPU-Cores führt zu einer suboptimalen Performance, da Idle-Zeiten auf einem Core nicht von beliebigen Threads genutzt werden können.
2. Eine solche Software-Architektur skaliert nicht, da z.B. ein Vier-Kern Prozessor damit nicht ausgenutzt werden könnte.

3.2 Projekt 2: Parallelisierung einer Soft-Realtime Software für Power-Quality

Das hier geschilderte System dient der Qualitätskontrolle und Fehlererkennung in Stromverteilnetzen.

Die Software dieses Systems lief ursprünglich auf einem Blackfin Single-Core unter dem Betriebssystem µC/OS. Die geplante Dual-Core Migration hatte eine bessere Performance zum Ziel, die neue Features ermöglichen sollte. In diesem Projekt ist mit zwei unterschiedlichen Ansätzen versucht worden, die Portierung auf eine Dual-Core CPU zu realisieren. Der erste Ansatz ist gescheitert.

Im ersten Ansatz war eine Migration auf SMP-Linux und einen ARM Dual-Core mit folgenden Konzepten geplant:

- Ersetzung der µC/OS Tasks durch Linux P-Threads.
- Restrukturierung des Nebenläufigkeitskonzeptes: Ursprünglich hat eine zentrale „schwergewichtige“ Task an einem Synchronisationsprimitiv („Event“) gewartet, das viele unterschiedliche Ereignisse signalisieren konnte (ähnlich einem select Systemcall). Diese Task sollte in viele einzelne Threads aufgebrochen werden, die jeweils eine spezifische Aufgabe haben.
- Ersetzung der µC/OS Events durch P-Thread Condition-Variable.

- Einsatz der Linux Realtime Fähigkeiten, um dem ursprünglichen Realtime-Scheduling in µC/OS nahe zu kommen.
- Ersetzung impliziter Synchronisation durch explizite Synchronisation: Interrupt-Locks und Außerkraftsetzung des OS-Schedulers wurden im Ursprungssystem für die Realisierung von gegenseitigem Ausschluss benutzt. Dies wäre im SMP-Fall nicht mehr möglich gewesen (siehe 4.1).

Dieser Ansatz wurde nach einem Jahr Projektlaufzeit aus folgenden Gründen verworfen:

1. Das Nebenläufigkeitskonzept der ursprünglichen Software basierte wesentlich auf Annahmen über die sequentielle Abarbeitung der Threads auf dem Single-Core, die gewünschte Abfolge wurde durch Prioritäten gewährleistet. Dies ließ sich jedoch im SMP-Fall nur schwer nachbilden und hätte grundlegende Restrukturierungen in der Software-Architektur erfordert.
2. Die ursprüngliche Software war stark an den proprietären Synchronisationsmitteln von µC/OS orientiert (Events, Queues). Die Ersetzung dieser Mittel durch auf P-Threads basierenden Ansätzen (Condition-Variablen, blockierende Queues), veränderte die Semantik des Systems und war daher schwieriger als erwartet.

Man hat sich dann in einem zweiten Migrationsversuch für eine Hardware- und Software-Architektur entschieden, die auf den in der Vergangenheit des Produktes bekannten und etablierten Technologien basierte, um das technische Risiko zu minimieren. Das System wurde auf einen Blackfin Dual-Core Prozessor portiert, als Betriebssystem wurde weiterhin µC/OS genutzt, obwohl es dort keine Unterstützung für SMP auf diesem Prozessor gibt. Daher wurden zwar bestimmte Aufgaben auf den zweiten Core ausgelagert, sie laufen jedoch dort ohne ein Betriebssystem („Bare-Metal“).

Die schematische Architektur zeigt die Abweichungen vom „Ideal“ (vgl. Abb. 1):

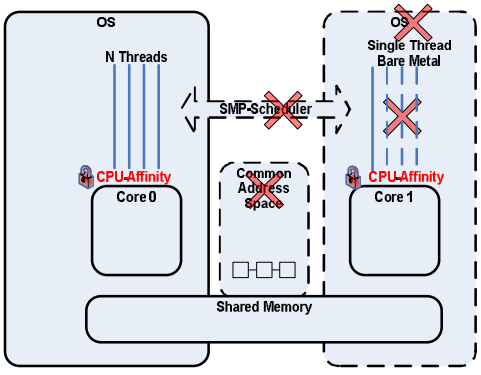

Abb. 2: Projekt 2, zweiter Core ohne OS

Bewertung: Dieses Projekt muss unter dem Gesichtspunkt der Parallelisierung als teilweise gescheitert angesehen werden, weil die gewählte Architektur die Vorteile der Multi-Core CPU nicht effizient ausnutzt. Es wurde weder ein klassischer SMP-Ansatz

noch ein asymmetrischer aber zumindest betriebssystemunterstützter Ansatz realisiert. Stattdessen wurde eine starre Aufgabenaufteilung vorgenommen, ohne OS-Unterstützung auf dem zweiten Core. Dies bringt Performance-Nachteile mit sich, da auf dem zweiten Core keine weiteren Tasks geschedult werden können, und es impliziert ein komplexes Programmiermodell, da die Kommunikation zwischen Software auf unterschiedlichen Cores nicht durch Betriebssystemabstraktionen unterstützt wird.

3.3 Projekt 3: Multi-Core-Portierung einer Telekommunikationssoftware

Dieses Parallelisierungsprojekt bezog sich auf ein Embedded System, das von Telekommunikationsprovidern genutzt wird, um DSL-Kommunikationsstrecken zu Endanwendern zu realisieren. Die Software ist komplett innerhalb eines Linux-Kernels realisiert. Der Grund, die Trennung von Kernel-Address-Space und User-Address-Space, die Linux anbietet, nicht zu nutzen, sind die Performance-Kosten der Systemcalls, die auf diese Weise vermieden werden sollen. Das System wurde von einem Single-Core Power-PC auf eine ARM Dual-Core CPU portiert. Die über Jahre gewachsene Software nutzt viele der Nebenläufigkeitsfeatures und Synchronisationsprimitive, die im Linux-Kernel existieren, was zu einem komplexen Design geführt hat. Der Aufwand dafür, dennoch bei SMP auf dem Dual-Core ein korrektes Verhalten der Anwendung zu erzielen, hat sich im Verlauf der Portierung als so hoch herausgestellt, dass schließlich auf die Nutzung des zweiten Cores komplett verzichtet wurde. Somit läuft die gesamte Software nun trotz vieler Threads auf nur einem Core der Dual-Core CPU.

Schematische Architektur und Abweichung vom idealen SMP (vgl. Abb. 1):

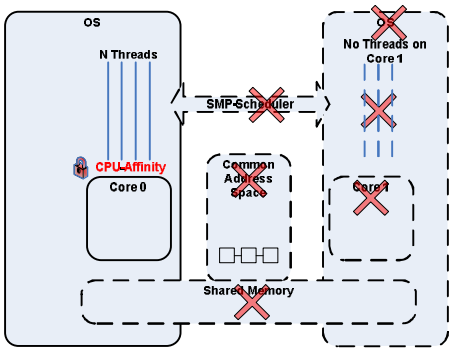

Abb. 3: Projekt 3, Dual-Core ohne Nutzung des zweiten Cores

Bewertung: Dieses Projekt ist aus Perspektive der Parallelisierung als komplett gescheitert anzusehen. Grund für das Scheitern ist eine Software-Architektur, die die Abstraktionen eines Betriebssystems nicht wie vorgesehen nutzt, nur um dadurch kurzfristig einen Performance-Gewinn zu erzielen. Die Architektur war im Single-Core Umfeld möglicherweise effizient, für die Dual-Core Nutzung aber nicht mehr tragfähig, weil sie aufgrund der mangelnden Abstraktion von Hardware und OS zu komplex wurde. Wäre von Anfang an das gesamte Multithreading auf der P-Thread Schnittstelle des OS aufgebaut worden, wäre eine Dual-Core Portierung aussichtsreicher gewesen.

4 Probleme und Lösungen bei der Embedded Parallelisierung

Ausgehend von den oben genannten Beispielen, die charakteristisch für die Situation in der Entwicklung von Embedded Systemen sind, werden nun die wesentlichen Herausforderungen bei der Parallelisierung im Zuge einer Portierung von Single-Core Systemen auf SMP-Hardware im Embedded Systeme Bereich analysiert, und es werden Lösungsstrategien dafür vorgeschlagen. Die Probleme haben neben ihrem technischen Aspekt, der sich mit konkreten Implementierungsstrategien lösen lässt, oft auch einen „Mindset-Anteil“, der sich eher auf Denkweisen oder eine „spezifische Kultur“ im Embedded Umfeld bezieht. Die folgende Tabelle gibt eine Übersicht. Anschließend wird jedes Problem detailliert behandelt.

Ref	***Situation***	***Problem***	***Lösung***	***Beispiel***
4.1	Impl. Synch. Interrupt-Lock	Kein SMP Mutex	Nur Expl. Synch. in Anwend.-SW	Proj. 1: BL-Blöcke Synchronisation
4.2	Impl. Synch. Prioritäten	Kein SMP Mutex	Prioritäten nie für Mutex nutzen	Projekt 2: Startup-Thread
4.3	Zeitl. Ablauf via Prioritäten	Nicht SMP-fähig	Kontrolle an Scheduler abgeben	Projekt 1: BL-Reihenfolge
4.4	Real-Time Optimierungen	Komplexität verhind. Portierg.	Datenkapselung, OS-Abstraktion	Projekt 1: Linux-Kernel Applikation
4.5	Tools f. Server optimiert	Embedded: traditionelle Para.	z.B. EMB² und MTAPI nutzen	Work-Stealing nicht in Embedded OS
4.6	Ausbildung Entwickler	Synchronisation Memory-Modelle	Gezielte Aus- und Weiterbildung	Erfahrungswerte aus Trainertätigkeit
4.6	Management: Naive Sicht	Falsche Planung	Bewusstsein für Komplexität	Projekt 3: Komplett unterschätzt

Tab 1: Probleme der Embedded Parallelisierung

4.1 Implizite Synchronisation durch Abschalten von Interrupts

Situation: Implizite Synchronisation kann definiert werden, als der Schutz einer Critical-Section auf Basis von Annahmen über das Scheduling-Verhalten des Betriebssystems. Durch das Abschalten von Interrupts lassen sich die beiden folgenden Varianten herstellen:

1. Sowohl die Interrupt-Behandlung (ISRs) als auch der Betriebssystem-Scheduler werden unterdrückt („Interrupt-Lock“).
2. Lediglich der Betriebssystem-Scheduler wird unterdrückt, aber die ISRs werden zugelassen. Dies ist eine sanftere Variante des obigen Ansatzes („Task-Lock“).

Diese Art der impliziten Synchronisation ist auf einem Single-Core sehr effizient, weil sie mit dem Abschalten von Interrupts im Wesentlichen auf Hardware-Ebene realisiert werden kann. Weder sind teure System-Calls mit Kontext-Switch in den OS-Kernel erforderlich noch Scheduling-Vorgänge des OS, die beispielsweise beim Akquirieren

eines Semaphors nötig werden können. In Single-Core Embedded Realtime Systemen ist dieses Vorgehen daher gängige und etablierte Praxis.

Problem: Interrupt-Locks lassen sich nicht in ein SMP-System übertragen, da sie auf der Annahme basieren, dass der exklusive Besitz eines CPU-Cores garantiert, dass kein anderer Code gleichzeitig ausgeführt wird. Diese Annahme ist jedoch in einem Multi-Core SMP-System nicht gültig, da zu dem Zeitpunkt an dem die Interrupts unterdrückt werden, auf einem anderen CPU-Core bereits der konkurrierende Code ausgeführt werden kann.

Lösung: Auf implizite Synchronisation durch Interrupt-Locks sollte im Anwendungscode prinzipiell verzichtet werden. Es handelt sich hier um ein Low-Level Synchronisationsmittel (z.B. für Treiber), dessen Verwendung in Anwendungscode auf mangelhaftes Design hindeutet. Da in Zukunft immer mehr Systeme auf Multi-Core-Hardware portiert werden müssen, sollte auch im Single-Core Fall auf Interrupt-Locks verzichtet werden, da sie sich nur schwer wieder ausbauen lassen.

Beispiel: Im Projekt 1 waren weite Teile der Synchronisation durch Interrupt-Locks realisiert. Es war kein schematisches Ersetzen möglich, sondern jede solche Critical-Section hat eine spezifische Umbaumaßnahme erfordert (siehe 3.1).

4.2 Implizite Synchronisation durch prioritätsbasiertes Realtime Scheduling

Situation: Prioritätsbasierte RT-OS stellen sicher, dass ein Thread nur von höher priorisieren Threads unterbrochen wird. Das wird oft dazu genutzt, eine Art gegenseitigen Ausschluss durch Prioritäten zu realisieren, indem ein auszuschließender Thread gleich oder niedriger priorisiert wird.

Problem: Dieser Ansatz ist nicht auf SMP übertragbar, da verschiedene Threads unabhängig von ihrer Priorität gleichzeitig auf unterschiedlichen CPUs laufen können. Die Gefahr bei einer SMP-Portierung ist, dass zunächst kein technisches Problem sichtbar wird, aber das Systemverhalten trotzdem korrumpiert wird. Dadurch ist diese Art der impliziten Synchronisation noch heimtückischer als das Interrupt-Locking.

Lösung: Prioritäten in RT-OS sollten prinzipiell nicht zur Synchronisation verwendet werden. Es sollten immer explizite Synchronisationsmechanismen genutzt werden.

Beispiel: Der Startup-Thread in Projekt 2 startet niedriger priorisierte Threads, die erst laufen dürfen nachdem der Startup-Thread beendet ist. Dieser Ablauf ist im SMP-Fall nicht gewährleistet.

4.3 Organisation zeitlicher Abläufe durch Realtime-Prioritäten

Situation: Ein typischer Embedded Entwickler ist bestrebt, zeitliche Abläufe innerhalb der Applikationslogik, wie Abarbeitungsreihenfolgen, Synchronisation und Datenfluss, explizit zu kontrollieren und vorherzusehen**,** um so die begrenzten Ressourcen optimal auszunutzen. Anders als beispielsweise ein typischer Programmierer eines Programmes,

das in einem Java Application-Server läuft, vermeidet er es, Entscheidungen über Ablaufreihenfolgen einem Scheduler zu überlassen. Oft wird das prioritätsbasierte Scheduling dafür missbraucht, diese zeitlichen Abläufe zu organisieren.

Problem: Dies widerspricht fundamental der Philosophie eines SMP-Betriebssystems oder moderneren Ansätzen feingranularer Parallelisierung z.B. Work-Stealing, dort kann schon aus technischen Gründen durch Prioritäten keine Ablaufreihenfolge gewährleistet werden. Da zudem ein optimaler Scheduling-Algorithmus ein NP-hartes Problem darstellt, ist es im Allgemeinen auch aus Perfomance-Gründen sinnvoller, Scheduling-Entscheidungen einem Scheduler zu überlassen, als sie explizit zu manipulieren, zumal Work-Stealing dem theoretischen Optimum relativ nahe kommt (vgl. [HER] S. 380).

Beispiel: In Projekt 1 realisieren Anwendungsthreads einen Graph von BL-Elementen, durch den Daten in einer bestimmten Reihenfolge fließen. Im Single-Core Fall ist diese Reihenfolge durch Prioritäten garantiert. Im SMP-Fall mussten alle Anwendungsthreads an einen Core gebunden werden, um den Singe-Core Ablauf zu imitieren.

Lösung: Die Embedded-Welt sollte lernen, mehr Kontrolle an Scheduler abzugeben. Software-Architekturen sollten entweder mit unterschiedlichen Reihenfolgen umgehen können, wie z.B. bei einem Server, der eingehende Requests abarbeitet, oder falls Reihenfolgen wichtig sind, diese explizit programmatisch realisieren und nicht auf Basis impliziter (Scheduling-)Annahmen. Prioritäten sollten ausschließlich genutzt werden, um zeitkritische Ereignisse vorrangig zu behandeln.

4.4 Riskante Mikro-Optimierungen zur Erfüllung von Realtime-Anforderungen

Situation: Die Denkweise von Embedded-Programmierern ist oftmals sehr hardwarenah. Dies führt zu Optimierungen, die sich auf eine bestimmte Hardware beziehen und die aus Performancegründen die Abstraktionen eines Betriebssystems umgehen. Es kommt auch vor, dass bestimmte Synchronisationsmittel, die aus Gründen der Datenkonsistenz sinnvoll wären, nicht eingesetzt werden, um die damit verbundenen Performance-Kosten zu vermeiden.

Problem: Diese Optimierungen basieren meist auf impliziten Annahmen über HW oder OS, die im SMP-Fall nicht immer gelten. Dies führt zu schwer zu findenden Problemen.

Beispiele: Die Vermeidung von System-Calls in Projekt 3 hat zu einer Linux-Kernel-Anwendung geführt. Dies hat aufgrund mangelnder Abstraktion die SMP-Portierung verhindert. Im Projekt 1 wurde die im Ringbuffer beim Lesen formal erforderliche Memory-Barrier weggelassen, da im Test kein Fehler nachweisbar war.

Lösung: Low-Level-Optimierungen sollten prinzipiell vermieden werden: OS-Abstraktionen sollten konsequent genutzt werden. Saubere Synchronisation kostet einen gewissen Teil der Performance, die ein SMP-System liefert. Dies muss bei der Dimensionierung der HW berücksichtigt werden. So sollten z.B. Memory-Barriers nicht explizit im Anwendungscode stehen sondern durch Synchronisationsprimitive ersetzt werden, denn die Barrier-Semantik ist sehr subtil und CPU-spezifisch und daher sind Memory-Barriers nicht naiv einsetzbar und nicht risikolos portierbar (vgl. [ARM] A 3.8

„Memory access order“ und [INT] 8.2 „Memory ordering“).

4.5 Tools und Paradigmen sind für Server und Desktop optimiert

Situation: Moderne Paradigmen der Parallelisierung, wie z.B. Work-Stealing, sind nicht im Embedded-Umfeld etabliert. Parallelisierungstools stammen oftmals aus dem Desktop- und Server-Bereich und sind daher nicht direkt für die Entwicklung von Embedded Systemen optimiert.

Problem: Die Embedded-Welt setzt moderne Parallelisierungsparadigmen nur zaghaft ein. Sie verharrt teilweise in traditionellen Denkmustern, dies führt zu den oben genannten Problemen der „schwergewichtigen“ starren Parallelisierung (vgl. 3.1).

Beispiele: Viele Embedded-OS wie z.B. VxWorks liefern keinen Work-Stealing Task-Scheduler mit (iOS hingegen schon, GCD). Es fehlen z.B. bei den general-purpose Work-Stealing Task-Schedulern Prioritäten. Auch klassische Thread-Bibliotheken sind nicht immer ideal für Realtime-Anforderungen, da oftmals Datenstrukturen benutzt werden, die implizit Semaphore verwenden und dynamisch Speicher allokieren (z.B. blockierende Queues auf Basis von C++ STL-Containern). In Realtime-Applikation sind jedoch unter Umständen Lock-Free und Wait-Free Datenstrukturen zu bevorzugen.

Lösung: Spezialisierte Embedded-Parallelisierungsbibliotheken wie z.B. EMB² von Siemens CT (vgl. [EMB]) schließen diese Lücke: Keine dynamische Speicherallokation, Embedded Work-Stealing Task-Scheduler, Lock-Free Datenstrukturen. EMB² entspricht überdies dem MTAPI-Standard für Embedded Task-Scheduling (vgl. [MTA]).

4.6 Denkweise von Entwicklern und Management bzgl. Parallelisierung

Situation: Eine bei Managern und Projektleitern weit verbreitete Vorstellung ist, dass eine Software, die mit vielen Threads auf einer Singe-Core CPU korrekt läuft, ohne wesentliche Umbauten auch auf einer Multi-Core CPU korrekt läuft. Dabei wird außer Acht gelassen, dass bestimmte Nebenläufigkeitsprobleme, die in einem Single-Core nur selten wahrnehmbar sind, im Multi-Core-Fall jedoch leicht zu einem inkonsistenten Systemzustand führen können (z.B. die Implikationen von Instruction-Reordering).

Auf Seiten der Entwickler gibt es oftmals ein Ausbildungsproblem. Aus meiner Erfahrung als Trainer und Berater weiß ich, dass die Bereiche Synchronisation und Memory-Modell vielen Entwicklern nur oberflächlich bekannt sind. Nur wenige der durchschnittlich ausgebildeten Entwickler können das Monitor-Pattern oder die Funktionsweise eines Spinlocks korrekt erklären oder die Semantik eines Relaxed-Consistent Memory-Modells erläutern.

Problem: Aus der vereinfachten Management-Sicht resultieren falsch geplante Projekte. Aus der mangelhaften Entwicklerausbildung resultieren der unsachgemäße Einsatz von Technologien und, was noch schwerer wiegt, nicht tragfähige Software-Architekturen.

Beispiele: Projekt 3 ist aufgrund falscher Technologie-Entscheidungen (Kernel-Applikation) in eine Schieflage geraten. In Projekt 2 wurde der Parallelisierungsaufwand

unterschätzt, was zu dem zweiten Anlauf geführt hat.

Lösung: Die Entwickler-Ausbildung sollte Paradigmen und Theorie der Parallelität anhand moderner Technologien vermitteln. Manager und Entscheider sollten erkennen, dass Parallelisierung ein komplexes Unterfangen ist, das Eingriffe auf allen Ebenen des Technologie-Stacks erfordert, und dass Multi-Core Hardware Herausforderungen an die Software-Architektur stellt.

5 Fazit: Strategien für Parallelisierung von Embedded Systemen

Die folgenden Regeln sind das Kondensat der hier analysierten Erfahrungen. Sie können als Richtschnur für die erfolgreiche Parallelisierung von Embedded Systemen dienen.

1. Performance-Optimierungen in der Mikro-Ebene vermeiden, OS-Abstraktionen nutzen, Applikation von OS-Kernel trennen.
2. Explizite Synchronisation im Realtime-Embedded Bereich etablieren.
3. Parallelisierungsbibliotheken und Tools verwenden, die für Embedded Systeme optimiert sind, z.B. den EMB² Work-Stealing Task-Scheduler.
3. Schrittweise Migration von Single-Core auf SMP, Big-Bang Migration ist riskant.
4. Zeitliche Kontrolle an OS oder Scheduler abgeben, zeitliche Abläufe nicht über Prioritäten sicherstellen.
5. Ausbildung verbessern bzgl. Synchronisation, Memory-Modell, Task-Scheduling.

Literaturverzeichnis

[ARM] ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html

[DUF] Duffy, J.; Sutter, H.: Concurrent Programming on Windows. Addison-Wesley 2008.

[EMB] EMB², https://github.com/siemens/embb

[GLS] Gleim, U.; Schüle, T.: Multi-Core Software, dpunkt 2011.

[HER] Herlihy, M.; Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann 2008.

[INT] Intel IA 32-64 Architecture-Manual Vol. 3, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf

[MTA] MTAPI-Spezifikation, http://www.Multi-Core-association.org/workgroup/mtapi.php

[QUA] Quade, J.; Kunst, E.: Linux-Treiber entwickeln. dpunkt 2011.

[QUM] Quade, J.; Mächtel, M.: Moderne Realzeitsysteme kompakt, dpunkt 2012.

[VXW] VxWorks 6.9 Kernel-Programmers Guide, Wind-River Systems Inc. 2013.