

Erweiterte Konzepte in C++: Multithreading

Tobias Franke
franke_t@rbg.informatik.tu-darmstadt.de

11. August 2005

„Herb also added that I can quote him in telling you that 'The C++ Standardization Committee is insanely interested in this proposal.'“ – Andrei Alexandrescu, cpp-threads Mailingliste

1 Einleitung

Das im C++ ISO Standard definierte Modell zur Ausführung von Programmen ist single-threaded. Im Zeitalter von Multicore Prozessoren und Hyperthreading-Techniken ist es jedoch für viele Entwickler wünschenswert, sichere, korrekte und standard-konforme Programme schreiben zu können, die sich der Multithreading-Technik bedienen. Statt dessen zeichnet sich auf Seite der C++-Nutzer ein durchwachsenes Bild mit vielen unterschiedlichen Threading-Bibliotheken ab, die sich nicht nur allesamt in ihrer Syntax mehr oder minder stark unterscheiden, sondern nicht einmal die von C++ zur Verfügung gestellten Mittel unterstützen bzw. C Interfaces zu Tage legen, und die Möglichkeit für *portablen* Code im Keim ersticken. Die Situation wird deutlich unangenehmer, wenn man den Blick auf die Einzelheiten und Details lenkt, die sich durch Multithreading ergeben: *Speichersichtbarkeit* und *Locking* sind Schlagwörter, die viel Diskussion über Effizienz und Korrektheit auslösen.

Neuere Erkenntnisse auf dem Gebiet zeigen, dass der Ursprung all dieser Probleme nicht auf den Programmierer oder die Bibliothek zurückzuführen ist, sondern sich viel fundamentaler in der Sprachdefinition selbst findet. Kurz gesagt: Multithreading-Unterstützung lässt sich nicht per Bibliothek implementieren, wenn der Compiler sich der Anwesenheit von Threads nicht bewusst ist und den Code dahingehend optimiert. Die Sprache muss, rein semantisch, mit Threads von Grund auf vertraut sein. In dem von der Arbeitsgruppe 21 veröffentlichten Paper *On the Future Evolution of C++* [10] wird daher als erster Punkt besprochen, wie der Thread-Support des neuen Standard C++0x definiert werden soll. Es existieren sowohl Vorschläge zum Speichermodell als auch zu einer möglichen Spracherweiterung. Außerdem werden diverse Bibliotheken wie PThreads oder Boost.Thread auf ihre Nutzbarkeit evaluiert, sollte das Speichermodell angepasst sein.

2 Grundlagen

Dieses Kapitel führt in die Grundelemente von Multithreading ein und bietet gleichzeitig einen Überblick darüber, was eine Multithreading-Bibliothek im größten bieten muss.

2.1 Threads

Die parallele Abarbeitung mehrerer Instruktionen auf einem Rechnersystem lässt sich auf viele Arten bewerkstelligen, die alle unter dem Begriff der Nebenläufigkeit - die kausale Unabhängigkeit mehrerer in Beziehung stehender Ereignisse - zusammengefasst werden. Darunter fallen sowohl asynchrone Kommunikation, Signal-/Interrupt-Handler, Multithreading und Multitasking, die aber allesamt grundlegende Unterschiede aufweisen.

Fängt man auf unterster Ebene an, so sind *Interrupthandler* die hardwarenahe Lösung der Nebenläufigkeit. Durch einen Interrupt oder eine Exception auf Seiten der CPU oder des BIOS wird ein Signal ausgelöst, durch das das Betriebssystem den laufenden Instruktionsfluss unterbricht, um eine Routine zur Bearbeitung des Interrupts zu starten. Beispielsweise ist der vom BIOS zur Verfügung gestellte Interrupt 13h für das Ansprechen von Festplatten zuständig. Andere Interrupts dienen z.B. dem Abfragen der Tastatur.

Auf der Ebene des Betriebssystems finden sich zwei Begriffe wieder, die leicht miteinander gleichgesetzt oder verwechselt werden, nämlich die des *Prozess* und des *Threads*. Beide teilen die Eigenschaft, einzelne, sequentielle Instruktionspfade zu beschreiben, die parallel zu anderen Sequenzen durch so genanntes *Time Slicing* oder über Multiprozessor Systeme ausgeführt werden können. Prozesse unterscheiden sich von Threads in dem Punkt, dass sie einen eigenen Zustand und Adressraum besitzen, und ausschließlich über die vom Betriebssystem bereitgestellten Mittel untereinander kommunizieren können. Prozesse beherbergen üblicherweise Threads, die sich den Heap des Prozess miteinander teilen, und somit über einen gemeinsamen Speicherbereich kommunizieren können. Der Kontextwechsel zweier Prozesse ist demnach aufwendiger, als der zweier Threads eines Prozess (wobei dieser Umstand abhängig von der Implementation eines Betriebssystems ist). Threads werden daher manchmal auch als *leichtgewichtige Prozesse* beschrieben.

Abstrahiert man vollständig vom Betriebssystem sowie der darunter angeordneten Hardware, so ist der Begriff *asynchrone Kommunikation* passend, um Nebenläufigkeit zu beschreiben: Ein Teilstück einer Aktion, das nebenläufig ausgeführt wird, kann Nachrichten an andere Teile absenden oder empfangen, ohne auf die Antwort warten zu müssen. In dieser Zeit wird die Arbeit weiter fortgesetzt, bis das Ergebnis eintrifft.

2.2 Synchronisation

In der Literatur findet sich keine durchgehende Definition für das Wort *Multithreading*, das von der konzeptionellen Sicht bis hin zum Software Engineering

unterschiedlich interpretiert wird. Es finden sich Definitionen wie *etwas, das sich jedes mal total unvorhersehbar verhält wenn es gestartet wird* oder *ein Modell zur nebenläufigen Programmierung*. Sieht man von den Definitionen ab, so bleibt der Indeterminismus die Grundproblematik bei der formalen Analyse nebenläufiger Programme, durch den die Vorhersage eines eindeutigen Programmablaufs unmöglich wird. So kann ein Thread z.B. sowohl in der Zuweisung einer Variable unterbrochen werden, als auch davor oder danach. Die Probleme die sich hierdurch ergeben, wurden erstmals 1965 von Edsger W. Dijkstra in seinem Artikel *Solution of a problem in concurrent programming control* [7] durch den von ihm eingeführten *kritischen Abschnitt* gelöst, in dem durch gegenseitigen Ausschluss ein Codebereich atomar gegenüber anderen Threads ausgeführt wird. Die später folgende ausführliche Abhandlung [8] erklärt mit dem Mechanismus der *Semaphore* ein Konzept, das bis heute bei der Implementierung von Multithreading Bibliotheken weitestgehend unverändert ist.

Durch die Parallelisierung diverser Programmteile können Fehler zu Tage treten, die sich im Code und per Debugger nur schwer auffinden lassen. Da das Schreiben korrekter Multithreading-Programme nur durch die richtige Anwendung von Synchronisation an kritischen Stellen gewährleistet ist, kann ein Fehler leicht entstehen, wenn diese Abschnitte nicht richtig gesichert wurden. Die zwei großen Vertreter dieser Fehlerklassen sind so genannte *Race Conditions* und *Live-/Deadlocks*.

Eine detaillierte Einführung in die Problematik findet sich in [13].

2.3 Speichermodell

Sicherlich der wichtigste Teil einer Multithreadingimplementation ist das Speichermodell. Bevor eine Entwicklung an den eigentlichen Multithreading-Primitiven beginnen kann, muss Klarheit darüber herrschen, wie sich der Code unter den neuen Bedingungen zu verhalten hat. Darunter fallen folgende Definitionen:

- **Atomare Operationen:** Operationen, welche garantiert unterbrechungsfrei ausgeführt werden können. Ohne solche Operationen ist die Umsetzung von z.B. Synchronisationsmechanismen praktisch unmöglich.
- **Partielle Ordnungen von Operationen:** Unter bestimmten Umständen spielt die Reihenfolge von Operationen eine wichtige Rolle, die der Compiler nicht verändern darf. Durch ein Ordnungsprimitiv wie *happens-before* kann definiert werden, welche Operationen wie zu ordnen sind (besonders bei Synchronisation zu beachten).
- **Speichersichtbarkeit:** Der Zeitpunkt, ab dem der gemeinsame Speicher für alle Threads den gleichen Inhalt besitzt. Dies ist sowohl aus softwaretechnischer Sicht als auch auf Seite der Hardware zu beachten.
- **Data Race Semantik:** Die Umstände, unter denen es zu einem Data Race kommen kann, und wie diese Operation endet (z.B. undefiniert oder via Exception). Viel Arbeit zu diesem Thema wurde in das *Java Speichermodell* [12] investiert, um den Schaden zu minimieren.

Das Speichermodell ist maßgebend für den Compiler, um Code zu transformieren, ohne dass dieser seine ursprüngliche Bedeutung verliert. Ein entsprechendes Modell für Multithreading gibt daher vor, welche Optimierungen in Anwesenheit von Threads erlaubt sind, und welche die korrekte Abarbeitung mit Threads stören. Allerdings greift das Speichermodell dabei über die Grenzen des Compilers hinweg auch die Hardware auf. Beispielhaft dafür steht der noch in der Entwicklung befindliche Cell-Prozessor [9], der mit mehreren Recheneinheiten (SPE: *Synergetic Processing Element*), die jeweils einen eigenen Cache besitzen, Multithreading effizienter gestalten soll. Hier sticht die Bedeutung der *Speichersichtbarkeit* besonders hervor, da der lokal gehaltene Cache jedes SPE nicht unbedingt sofort mit allen anderen Recheneinheiten abgeglichen ist. Gerade bei der Initialisierung großer Objekte muss daher im Speichermodell festgelegt sein, wann der Zugriff einzelner Threads auf ein neu angelegtes Objekt letzten Endes erfolgen kann und darf.

2.4 Thread Safety

Die oben erwähnten Races und Live-/Deadlocks sind Merkmale von Programmen, die nicht thread-safe sind. Eine der vielen Definitionen von *Thread-Safety* ist, dass ein Programm genau dann *sicher* ist, wenn es durch die Verwendung mehrerer Threads im Programmablauf nicht zu unerwartetem oder instabilem Verhalten kommen kann. Synchronisation ist dabei ein Hilfsmittel, um diese Sicherheit zu erreichen. Da es jedoch einfach ist, gerade bei der Synchronisation Fehler zu machen, ist es durch reines Lesen des Source Code oft nicht möglich festzustellen, ob ein Programm thread-safe ist oder nicht. Es gibt nur eine Reihe Indikatoren, die darauf hinweisen, dass das Gegenteil der Fall ist, wie z.B. Zugriff auf globale Variablen. Ein verwandtes Problem unter C++ ist Exception-Safety, bei dem ähnliche Prinzipien verfolgt werden. Generell kann man sagen, dass es schwierig ist, ein Programm mit sicherer Verwendung von Threads zu schreiben.

2.5 Weiteres

Die meisten Multithreading-Bibliotheken unterstützen, neben der reinen Erzeugung von Threads und deren Synchronisation durch Semaphoren, Condition Variables oder die Verwaltung mehrerer Attribute eines Threads wie z.B. Scheduling-Parameter oder Stackgröße. Eine weitere Besprechung dieser Mechanismen entfällt, um den Rahmen der Ausarbeitung nicht zu sprengen.

3 Multithreading und C++

3.1 Compileroptimierungen

Die bisherige Erfahrung hat gezeigt, dass es trotz der fehlenden Multithreading-Spezifikation im C++ Standard möglich ist, entsprechende Funktionalität über Bibliotheken zu nutzen. Nicht ganz evident hingegen blieb vorerst die Frage, ob der vom Compiler generierte Code auch korrekt gegenüber der eigentlichen

Bedeutung des Programms ist. In [6] wird erstmals gezeigt, dass ein Compiler in Anwesenheit von Multithreading ein Programm, das semantisch als *thread safe* gilt, fehlerhaft übersetzen kann, sobald der entsprechende Code optimiert wird. Drei Fehlerklassen werden im Folgenden vorgestellt.

```
flag1 = flag2 = 0;
```

Thread 1

```
while(flag1 == 0);
flag2 = 1;
```

Thread 2

```
flag1 = 1;
while(flag2 == 0);
```

Abbildung 1: Cacheing (aus [6])

Das in Abbildung 1 gezeigte Programm beendet, rein intuitiv, nach der Ausführung von Thread 2 den restlichen Ablauf. Compiler greifen jedoch gerne auf ein Hilfsmittel zurück, sobald auf eine Variable mehrfach in einem Abschnitt zugegriffen wird (siehe `/Og` Option in Visual Studio .NET 2003). Statt den Inhalt der beiden von den Threads geteilten Variablen `flag1` und `flag2` direkt abzufragen, wird dieser vorher aus Performancegründen in ein Register geladen. Dies führt dazu, dass beide Threads außerhalb des Kontext betrachtet unverändert bleiben, allerdings das Endresultat ein ganz anderes ist: `flag1` wird trotz der Modifikation durch Thread 2 nicht mehr ausgelesen, wodurch das Programm in einem Deadlock endet. Auch die Absicherung durch einen Mutex beim Zugriff auf die Variablen kann hier nicht viel bewirken, wenn deren Inhalt zum schnelleren Bearbeiten außerhalb des Mutex in ein Register geladen wird.

Eine Lösung für diese Problemklasse bietet in einigen Situationen das `volatile` Schlüsselwort, durch das signalisiert wird, dass ein Variableninhalt sich jeder Zeit auf für den Compiler unvorhersehbare Weise ändern kann. Mit diesem Mittel können Variablen wie z.B. `flag1/flag2` gegen Caching geschützt werden, allerdings zu einem hohen Preis: Sämtliche Optimierungen entfallen zum momentanen Zeitpunkt für den Zugriff auf `volatile` Variablen.

```
struct { int a:17; int b:15; } x;
x.a = 42;
```

```
0041122E mov     eax,dword ptr [x (416564h)]
00411233 and     eax,0FFFE0000h
00411238 or     eax,2Ah
0041123B mov     dword ptr [x (416564h)],eax
```

Abbildung 2: Codesubstitution bei Bitfeldern (VS.NET 2003 Disassemblierung)

Eine weitere Problemklasse fällt speziell in den Bereich der Behandlung von Bitfeldern. Beim Laden und Schreiben von Feldern, deren Breite keine Potenz von 2 oder weniger als 8bit ist, wird der Code (abhängig von dem darunter liegenden Maschinencode) an die Situation angepasst. In Abbildung 2 wird der

Zugriff auf das 17bit Feld ersetzt, indem ein 32bit breites Feld den Gesamthalt der Struktur übernimmt, verändert und zurückgeschrieben wird. Die neue Situation führt allerdings potentiell zu einem Data Race: Die gesamte Struktur wird zu Anfang der Operation in einem gleichgroßen Temporärspeicher abgelegt, und wird am Ende mit der veränderten Kopie überschrieben. Ein in der Zwischenzeit gesetzter Wert in x.b geht dadurch verloren.

```
p(lock);  
... kritischer Abschnitt ...  
v(lock);
```

Abbildung 3: Sequenzänderung

Das letzte Beispiel behandelt Sequenzänderung, die vom Compiler durchgeführt werden, um einen möglichst schnellen Bearbeitungszyklus zu erzwingen (siehe `man gcc` zu der Option `-fschedule-insns`). Dazu können beispielsweise Speicherzugriffe zusammengefasst oder umgeschichtet werden, solange diese die Abhängigkeiten innerhalb des Programmablaufs unverändert lassen (dies gilt ebenfalls für optimierende Hardware). Vor allem Code, der per `inline` ersetzt wird, kann dieser Gefahr unterliegen: In Abbildung 3 kann der kritische Abschnitt mit den Operationen `p()` oder `v()` vermischt werden, sollte deren Code via `inline` an dieser Stelle eingefügt werden. Sollte sich für den Compiler ergeben, dass weder `p()` noch `v()` Einfluss auf den Zugriff einer Variable innerhalb des kritischen Abschnitts haben, so kann dieser im schlimmsten Fall den Zugriff aus dem gesicherten Bereich hinaus bewegen.

Gerade das letzte Problem wurde von Compilern, die explizit PThreads unterstützen, durch folgende Mechanismen gelöst (siehe [5]): Zum einen enthalten Befehle wie `pthread_mutex_lock()` so genannte *Hardware Barrieren*, die die Hardware am umordnen von Speicheroperationen hindern sollen, zum anderen werden sie vom Compiler als *undurchsichtige* Funktionen, also Funktionen, die der Compiler nicht weiter zerlegen und parsen kann, betrachtet. In diesem Fall muss der Compiler davon ausgehen, dass dieser Befehl praktisch jede globale Variable ändern kann, und somit seine Position im Code eine wichtige Bedeutung hat. Allerdings helfen diese Mechanismen nur, das Problem einzudämmen, denn die vorangegangenen Fälle aus Abbildung 1 und 2 werden dadurch nicht gelöst.

Das in [6] behandelte Problem wurde in [5] weiter ausgearbeitet, mit dem Schluss, dass eine Implementation von Multithreading als reine Bibliothek nicht möglich ist. Wie Boehm korrekt erwähnt, kommt die Unterstützung von PThreads auf Seiten des Compilers einem ersten Schritt zur Spracherweiterung gleich.

3.2 Function-Local Statics

Weiterhin problematisch gestalten sich lokale `static` Variablen von Funktionen. Ist einer Menge von Threads die gleiche Funktion zugeordnet, so bleibt zu klären, wie und wann die entsprechende Variable initialisiert wird, um die u.U. teuren Initialisierungskosten nicht mehrfach zu tragen (siehe `/GT` Option in Visual Studio .NET 2003). Dazu ist ein Flag in der Art `is_initialized` nötig, das allerdings ohne entsprechende atomare Eigenschaften bisher zu einem potentiellen Race führt: Durch den vom Compiler neu generierten Code, der bisher ohne Lock-Mechanismen vollkommen unsynchronisiert arbeitet, kann unter ungünstigen Umständen ein Thread ein `static` Feld als bereits initialisiert erachten, während ein anderer Thread an der lang andauernden Initialisierung selbst arbeitet. In diesem Fall kommt es zum Zugriff auf ein uninitialisiertes Feld, und das weitere Verhalten des Programms ist undefiniert.

4 Aktuelle Diskussion

4.1 Spracherweiterung

Wie im vorherigen Abschnitt erarbeitet wurde, ist das C++ Single-Thread Modell nicht für Multithread-Code geeignet und muss an die neue Situation angepasst werden. In einem *Strawman Proposal* [4] werden erste Modifikationen an der Sprache vorgeschlagen, die sowohl die Ordnung von Befehlen, Bitfelder, Synchronisation, Function-Local-Statics, das `volatile` Schlüsselwort als auch Thread-Local Variablen ansprechen.

In [2] wird die Semantik von Data Races angesprochen: Der momentane Konsens der Gruppe ist, die Semantik weiterhin undefiniert zu lassen, um Compilern Optimierungsmöglichkeiten nicht zu untersagen. Allerdings bleiben dadurch einige der besprochenen Probleme bestehen. Ob sich diese Meinung weiterhin durchsetzt ist abhängig von der Entscheidung des Komitees. Im Strawman Proposal wird die Thematik weiter klassifiziert: Mit neuen Relationen wie *synchronized-with* und *happens-before* soll die Ordnung von Operationen, die für Multithreading kritisch sind, gesichert werden. Innerhalb dieser Bereiche ist klar definiert, das ein Data Race nur dann auftritt, wenn das beschriebene Regelwerk verletzt wird (z.B. durch einen Befehl, der keiner Ordnung unterliegt, aber auf dem selben Speicherbereich arbeitet).

Das `volatile` Schlüsselwort soll weiter gestärkt werden, um für gemeinsame Variablen in Multithreadapplikationen besser und kosteneffizienter genutzt zu werden, statt, wie bisher, nur für IO Operationen. Um hier eine Unterscheidung zu treffen, wann es sich um eine *normale* Anwendung von `volatile`, und wann um eine Verwendung für Multithreading handelt, wird die Schreibweise `__async volatile` für Multithread-Code vorgeschlagen. Gleichzeitig handelt es sich bei dieser Qualifikation um eine im neuen Standard definierte *Synchronisations-Operation*, in der garantiert wird, das die Ordnung des Befehls nicht verloren geht. Damit der Compiler allerdings korrekten Code mit `__async volatile` produzieren kann, ist vorher zu klären, wie die qualifizierten Datensätze atomar

geschrieben werden können, ohne teure *Speicherbarrieren* zu verwenden. Dazu sind vor allem für größere Datenstrukturen, die nicht mit einem einzigen Store Befehl geschrieben werden können, u.U. mehrere wieder-ausführbare atomare Operationen nötig, um die Einheit zu gewährleisten. Prozessoren ohne jegliche atomare Lese-/Schreiboperationen müssen in diesem Fall auf eine Emulation zurückgreifen, oder können die Unterstützung nicht anbieten. Weitere atomare Primitive wie CAS (siehe Abschnitt 4.2) benötigen ebenfalls Prozessorunterstützung.

Function-Local Statics sind ein umstrittenes Thema. Es existieren verschiedene Vorschläge von Boehm, Lea und Alexandrescu zur Syntaxerweiterung bzw. Anpassung, um `static` Variablen korrekt zu initialisieren, wie z.B. `static (synchronized) var` oder `protected static var`. Allgemein wird die Lösung, den Compiler entsprechenden Synchronisationscode hinzuzufügen zu lassen, als beste Wahl angesehen, wobei die Gruppe zu der drastischen Maßnahme tendiert, das `static` Schlüsselwort vollständig zu entfernen. Hierzu wird noch auf mehr Informationen der einzelnen Compilerhersteller gewartet.

Um einen Großteil der Optimierung durch Absicherung gegen Races nicht zu verlieren, wird für ein neues Schlüsselwort `__thread`, das seit einiger Zeit als Extension bekannt ist, geworben. In diesem Fall lassen sich Speicherzugriffe, die nur innerhalb eines Threads vorkommen (*Thread-Local*), weiterhin optimieren. Beispielsweise kann somit eine Unterscheidung zwischen einem Smartpointer, der über eine Thread-Grenze hinweg genutzt wird, und einem *lokalen* Smartpointer, für den keine Synchronisation erforderlich ist, getroffen werden.

Ein noch scheinbar unbehandeltes Thema ist das Verhalten von Exceptions in Multithreadanwendungen, und wie diese sich durch den Code propagieren. Ohne ein Speichermodell wird die Behandlung dieser Thematik allerdings nicht möglich sein.

Über die Threading API, die getrennt vom Speichermodell behandelt werden soll, herrscht noch Unklarheit. Auf der einen Seite ist es wünschenswert, einen gemeinsamen großen Standard zu haben, z.B. in ähnlicher Form wie Boost.Threads. Auf der anderen Seite sind gerade Bibliotheken wie PThreads weit verbreitet. Sollte das Speichermodell tatsächlich getrennt von der Threading API umgesetzt werden, dann ist die Behandlung einer Standardbibliothek vorerst nicht erforderlich, da sich alle bereits bestehenden Bibliotheken verwenden lassen. Auch hierzu muss sich das Komitee noch äußern.

4.2 Lock-freie Datenstrukturen

Um Multithreading in C++ korrekt implementieren zu können, sind, wie in den vorangegangenen Abschnitten demonstriert, atomare Locking-Mechanismen erforderlich. Allerdings sind gerade diese anfällig für eine Reihe von Problemen, wie die in Abschnitt 2.2 angesprochenen Live-/Deadlocks. Ohne Synchronisation entstehen allerdings auf Dauer Data Races, die den korrekten Ablauf des Programms mehr oder minder dem Zufall überlassen.

In [1] stellt der Autor eine Methode zur Synchronisation vor, die nicht auf Locking-Mechanismen zurückgreift. Er nutzt stattdessen ein Primitiv, das

```

template <class T>
bool CAS(T* addr, T expected, T fresh) {
    if (*addr != expected)
        return false;
    *addr = fresh;
    return true;
}

```

Abbildung 4: CAS Beispiel (aus [1])

nach [11] ausreichend ist, um jede Datenstruktur thread-safe zu implementieren. Die CAS-Methode (*Compare-And-Swap*) ist dabei aus einigen anderen Anwendungsfällen lange bekannt: Source-Control-Management Systeme wie Subversion legen, statt für die Dauer der Bearbeitung einer Datei diese für alle anderen Teilnehmer zu sperren, eine lokale Kopie an. Der Benutzer verändert diese und schreibt sie zu einem anderen Zeitpunkt zurück. Hat sich der Zustand des Source-Archivs in dieser Zeit verändert, so muss der Nutzer ggf. Modifikationen an seinem eigenen Datensatz vornehmen, um das Endergebnis synchron zu halten. In Abbildung 4 ist eine Beispielimplementierung dieses Mechanismus zu sehen. Viele modernen Prozessoren haben mittlerweile für einige Datentypen diesen Mechanismus als atomare Einheit implementiert (z.B. `CMPXCHG`).

Die Motivation für CAS ist, dass gemeinsame Daten weitaus öfter nur zum lesen genutzt werden, als zum beschreiben, wodurch die hohen Kosten der lokalen Kopie relativiert werden. Abgesehen davon werden implizit alle vorherigen Problematiken wie z.B. Deadlocks umgangen. Gegen CAS sprechen unkontrollierbare Prioritäten und die Schwierigkeit, Datenstrukturen an das neue Prinzip anzupassen. Daher ist der allgemeine Konsens, dass beide Verfahren gebraucht werden um den Code an passenden Stellen in Bezug auf die Performance zu verbessern, und gleichzeitig die Übersicht zu wahren.

Ein Umstand von CAS ist die Frage, wann der gemeinsam genutzte Speicher gelöscht werden soll. Sollte beim Löschen des Speichers noch ein Thread den Speicher in einer lokalen Kopie bearbeiten und anschliessend versuchen, zurückzuschreiben, so käme es zu einem undefinierten Verhalten. Um den Rahmen dieser Ausarbeitung nicht zu sprengen sei auf die Lösung in [3] verwiesen.

5 Fazit

C++ ist die dominierende Sprache in der Softwareentwicklung. Mit den in den kommenden Jahren weiter zunehmenden, parallelisierten Architekturen ist ein Weg zu Optimierungen verfügbar, der jedoch die Anpassung des C++ Standards erfordert um sichere und korrekte Implementierung von Multithreadanwendungen zu erlauben. Dazu sind semantische Korrekturen an der Sprache, sowie eine Standardbibliothek für Multithreading nötig, um gleichzeitig Portabilität zu gewährleisten.

Literatur

- [1] Andrei Alexandrescu. Lock-free data structures. *C/C++ User Journal*, October 2004.
- [2] Andrei Alexandrescu, Hans Boehm, Kevlin Henney, Ben Hutchings, Doug Lea, and Bill Pugh. Memory model for multithreaded c++: Issues. Technical Report WG21/N1777=J16/05-0037, ISO/IEC Information Technology Task Force, March 2005.
- [3] Andrei Alexandrescu and Maged Michael. Lock-free data structures with hazard pointers. *C/C++ User Journal*, December 2004.
- [4] Boehm. A memory model for c++: Strawman proposal, 2005. <http://www.hpl.hp.com/personal/Hans.Boehm/c++mm/mm.html>.
- [5] Hans Boehm. Threads cannot be implemented as a library. Technical Report HPL-2004-209, HP Laboratories Palo Alto, December 2004.
- [6] Peter A. Buhr. Are safe concurrency libraries possible? *Commun. ACM*, 38(2):117–120, 1995.
- [7] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965. <http://doi.acm.org/10.1145/365559.365617>.
- [8] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.
- [9] D. Pham et al. The design and implementation of a first-generation cell processor. In *ISSCC 2005 IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, pages 184–185. 2005.
- [10] Lois Goldthwaite. On the future evolution of c++. Technical Report JTC1/SC22/WG21 N1774=05-0034, ISO/IEC Information Technology Task Force, March 2005.
- [11] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. <http://doi.acm.org/10.1145/114005.102808>.
- [12] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM Press. <http://doi.acm.org/10.1145/1040305.1040336>.
- [13] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.