Großer Beleg
Tracing unter L4/Fiasco

Andreas Weigand

11. April 2003

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Dresden Real Time Operating System, kurz Drops [3], ist ein Forschungsprojekt der Betriebssystemegruppe an der TU-Dresden. Basierend auf dem Mikrokern Fiasco sollen in diesem System Komponenten mit Echtzeit- oder Sicherheitsanforderungen neben Timesharinganwendungen ausgeführt werden.

Mikrokerne der zweiten Generation, zu denen auch Fiasco gehört, stellen nur die minimal notwendige Kern-Funktionalität zur Verfügung: Adressräume, Threads und Kommunikationsmechanismen[1]. Die L4-Mikrokerne zeichnen sich im Allgemeinen durch besonders schnelle IPC aus.

Im Gegensatz zu monolithischen Systemen besteht Drops aus einer Vielzahl von Servern, die aus Sicherheitsgründen in verschiedenen Adressräumen ausgeführt werden. Die Inter Process Communication erhält eine zentrale Rolle, weil damit Adressraumübergreifende Kommunikation möglich ist. IPC wird aber auch zur Synchronisation zwischen Threads eines Adressraumes eingesetzt.

Ein Problem unter Drops ist derzeit die Fehlersuche während der Programmentwicklung, da nur eingeschränkte Debugging-Möglichkeiten existieren. Eine detaillierte Analyse des dynamischen Programmverhaltens umfasst die Aufzeichnung der IPC zwischen Threads und das Aufrufverhalten von Programmfunktionen. Diese Informationen können dazu beitragen, Fehler im Ablaufverhalten des Programms zu finden und die Entwicklungszeit verkürzen.

In dieser Arbeit ein Werkzeug vorgestellt, welches die detaillierte Zeitmessung von Funktionsaufrufen innerhalb eines Threads erlaubt. Dabei wird auch die IPC-Kommunikation zwischen Threads sowie die Threadumschaltung durch Fiasco erfasst. Bei der Implementation wurde auf einen möglichst geringen Overhead durch die Ereignisaufzeichnung geachtet. Alle gesammelten Daten werden aufbereitet und können danach mit dem externen Werkzeug VAMPIR [8] grafisch dargestellt und ausgewertet werden.

Aufbau der Arbeit

Im zweiten Kapitel werden die Grundlagen dieser Arbeit vorgestellt. Dabei werden die verwendeten Begriffe erläutert und bestehende Tracingsysteme vorgestellt. Kapitel 3 erläutert den Entwurf des entwickelten Tracingsystems. Es werden verschiedene Lösungskonzepte vorgestellt und gegeneinander abgewogen.

[1] Inter Process Communication, kurz: IPC

Auf die Implementation und die entstandenen APIs der Komponenten wird in Kapitel 4 eingegangen. Das fünfte Kapitel enthält eine Leistungsbewertung des Tracingsystems. Zum Abschluss folgt eine kurze Zusammenfassung sowie ein Ausblick auf künftige Arbeiten.

Kapitel 2

Grundlagen, Stand der Technik

Zur Analyse des Ablaufes eines Programms müssen Daten über dieses Programm gesammelt werden. Dieser Vorgang wird im folgenden als **Tracing** bezeichnet. Die beim Tracing gesammelten Daten werden werden **Tracerecords**, oder auch **Tracingdaten**, genannt. Zur Zwischenspeicherung der Tracerecords im System dient ein **Tracepuffer**.

Nach dem Programmlauf werden die gesammelten Tracingdaten aller beteiligten Prozesse zusammengeführt und aufbereitet. Anschließend werden die Daten ausgewertet und visualisiert. Anhand der entstehenden Statistiken und Graphen können Erkenntnisse über das Programmverhalten in verschiedenen Anwendungssituationen gewonnen werden. So können Fehler oder Engpässe in Programmen entdeckt und beseitigt werden.

Tracing vs. Debugging

Debugging dient vor allem der Fehlersuche in Programmen. Dazu kann Tracing auch verwendet werden, indem bei der Analyse der Tracedaten beispielsweise falsche Funktionsaufrufe festgestellt werden.

Tracing ist jedoch noch wesentlich mächtiger, weil dabei auch Informationen über das dynamische Programmverhalten (insbesondere auch im Zusammenspiel mit anderen Programmen) ermittelt werden. Dadurch können kritische Pfade, innerhalb einer, oder auch mehrerer, Applikationen entdeckt und verbessert werden. Ebenso können dabei unnütze Funktionsaufrufe, beispielsweise mehrere Aufrufe der Initialisierungsfunktion einer Bibliothek, entdeckt und beseitigt werden.

2.1 Bestehende Tracingsysteme

Zum Erzeugen der Tracingdaten muss die Anwendung instrumentiert werden. Dazu gibt es mehrere Ansätze.

1. Eine einfache, aber recht aufwändige Methode ist es, manuell an allen interessant erscheinenden Stellen Aufrufe zum Erzeugen eines Tracerecords einzufügen.

Der Vorteil dieser Methode besteht darin, dass der Nutzer genau entscheiden kann, welche Teile seiner Applikation instrumentiert werden sollen, und welche nicht. So werden nur die interessanten Funktionen instrumentiert.

Genau darin liegt aber auch die Schwierigkeit zu entscheiden, welche Stellen interessant und welche eher uninteressant sind. Ein weiterer Nachteil besteht darin, dass die Anwendung je nach Anzahl der Funktionen in erheblichem Maße verändert werden muss, um die Tracerecords zu erzeugen.

Ein Beispiel für diese Art der Instrumentierung ist das Linux Tracing Toolkit [13, 14].

2. Eine ähnliche Methode der manuellen Instrumentierung besteht darin, nicht das Programm an sich zu instrumentieren, sondern die benutzten Bibliotheken.

   Der Vorteil gegenüber Methode 1 besteht darin, dass die einmal instrumentierten Bibliotheken mit verschiedenen Programmen genutzt werden können. Ferner können solche Bibliotheken in verschiedenen Versionen (unter anderem mit und ohne Tracing) auf einem System vorhanden sein. So kann Tracing einfach durch Nutzung der passenden Bibliothek erreicht werden (beispielsweise durch den `LD_PRELOAD`-Mechanismus unter Unix).

   Ein Beispiel dafür sind die Vampirtrace MPI Bibliotheken [10].

3. Es ist nicht immer mit vertretbarem Aufwand möglich, Programme oder Bibliotheken manuell zu instrumentieren. Dann sollte die Instrumentierung automatisiert werden.

   Vorteilhaft an dieser Methode ist, dass der Quellcode des Programms nahezu unverändert bleibt. So kann Tracing einfach und ohne Aufwand einem bestehenden Quelltext hinzugefügt werden.

   Ein Problem dieser Methode ist jedoch die Steuerung des Automatismus. Meist werden einfach alle Funktionen instrumentiert, so dass wesentlich größere Tracingdatenmengen als bei den vorangegangenen Methoden entstehen.

   Der beim DROPS-Projekt verwendete GNU-C Compiler *gcc* bietet mit der Option `-finstrument-functions` (siehe [15]) an, bei Aufruf und Verlassen einer Funktion entsprechende Tracingfunktionen aufzurufen.

Bei allen genannten Methoden muss das untersuchte Programm zumindest minimal in der Initialisierungs- sowie in der Beendigungsphase angepasst[1] werden, damit sich das Tracingsystem ordnungsgemäß initialisieren und beenden kann.

2.1.1 Linux Tracing Toolkit

Das *Linux Tracing Toolkit*[13, 14], kurz LTT, ist ein Projekt zum Aufzeichen von mehr als 40 verschiedenen Ereignissen innerhalb Linuxkerns. Diese Ereignisse umfassen die Prozessumschaltung, Systemaufrufe, Interrupts, Datei-IO, Netzwerk-IO, Memory Management und weitere Aktivitäten.

[1] Bei Methode 2 kann diese Veränderung bei dynamisch gelinkten Bibliotheken entfallen, indem Initialisierung und Ende in speziellen Sektionen der Bibliothek durchgeführt werden.

Der Linuxkern wurde dafür um ein Tracingmodul erweitert. Dieses Modul verwaltet einen zweigeteilten Tracepuffer und enthält Funktionen zum erzeugen von Tracerecords. Weiterhin enthält es noch einen Gerätetreiber für das Gerät `/dev/tracer`. Die Tracingfunktionalität kann über dieses Interface jederzeit kontrolliert und verändert werden. Zur Ereignisaufzeichnung wurde der Linuxkern an allen notwendigen Stellen verändert, damit die LTT-Funktionen aufgerufen werden und Tracerecords erzeugen.

Das LTT-Kernmodul sammelt die Tracingdaten im Tracepuffer, dauerhaft gesichert werden sie dort aber nicht. Für diese Aufgabe wurde ein Daemon implementiert, der über `/dev/tracer` mit dem Kernmodul kommuniziert. Wenn ein Tracepuffer gefüllt wurde informiert ihn das LTT-Modul, damit der Daemon den Puffer auf Festplatte abspeichern kann. Der Daemon kontrolliert auch das Kerntracing, indem er im Kern nur die vom Nutzer gewünschten Ereignisse aufzeichnen lässt.

Zur Auswertung der aufgezeichneten Tracingdaten dient eine dafür entwickelte grafische Software. Der Nutzer kann zwischen verschiedenen Ansichten wählen:

- Im Ereignisgraph werden alle Prozesse und die dazugehörigen Ereignisse gleichzeitig dargestellt. Verschiedene Arten des Kerneintritts, wie Systemaufrufe, Interrupts oder Traps werden in verschiedenen Farben dargestellt. So erhält der Nutzer einen Überblick über das Gesamtsystem. Er kann nun beliebig in die Darstellung hineinzoomen, um genaue Angaben über die Abläufe zu erhalten.

- Die Prozessanalyse dient der Anzeige von detaillierten Informationen zu jedem Prozess. Dabei werden neben der Laufzeit in Nutzer- und Kernmodus auch die von diesem Prozess ausgelösten Ereignisse und deren Dauer und Anzahl angezeigt.

2.1.2 Vampirtrace

Die *Vampirtrace MPI Profiling Library*[10, 12] ist eine Tracing- und Profiling Bibliothek für MPI Programme auf Parallelrechnern oder Clustern. Sie zeichnet alle MPI-Aufrufe auf, indem die normalen MPI-Funktionen durch andere, für Tracing instrumentierte Funktionen ersetzt werden.

Eigene Applikationen müssen zur Ereignisaufzeichnung lediglich gegen *Vampirtrace* neu gelinkt werden. Bei der Ausführung entsteht für jeden Knoten des Programms eine Datei mit den Tracerecords. Diese Dateien werden nach dem Programmlauf zusammengefügt und vorverarbeitet. Das daraus resultierende Tracefile wird dann mit VAMPIR (siehe Abschnitt 2.4) visualisiert und ausgewertet.

Für eine genaue Steuerung der Ereignisaufzeichnug muss das untersuchte Programm aber geringfügig angepasst werden. Dann kann das Tracing auf einzelne Ereignisse eingeschränkt werden. Mit den Funktionen `VT_traceon` und `VT_traceoff` (siehe [11]) kann Tracing für bestimmte Programmteile gezielt aktiviert oder deaktiviert werden. Mit *Vampirtrace* auch möglich, eigene Ereignisse aufzuzeichnen.

2.2 Fiasco

Fiasco ist die Dresdner Implementation der L4-Schnittstelle. Dieser Mikrokern wird an der Professur Betriebssysteme [1] des Instituts für Systemarchitektur permanent weiterentwickelt. Er zeichnet sich gegenüber anderen Implementationen dadurch aus, dass er auch im Kern vollständig unterbrechbar ist. Diese Eigenschaft führt zu geringen Latenzzeiten, wie sie für Echtzeitanwendungen notwendig sind. Allerdings sind deswegen bei Veränderungen am Kern Vorkehrungen für die Konsistenz des Systems zu treffen, damit nach einer Unterbrechung nicht mit ungültigen Daten weitergearbeitet wird.

Durch eine nahezu vollständige objektorientierte Implementation in C++ kann Fiasco gut erweitert werden. Eine für die Fehlersuche besonders nützliches Erweiterung ist der in Fiasco integrierte Kerndebugger JDB.

2.2.1 JDB – Fiasco's Kerndebugger

Der Kerndebugger JDB [5] ist fest in Fiasco integriert[2]. Dieser Debugger arbeitet direkt im Kernmodus und hat damit Zugriff auf die Daten aller laufenden Tasks sowie auf alle Kernbereiche. Dadurch kann er nicht nur Nutzerprogramme, sondern auch den Kern vollständig kontrollieren. Solange der Kerndebugger aktiv ist, wird das restliche System, insbesondere die Systemzeit, angehalten. Dadurch sind die möglicherweise langen Unterbrechungen durch JDB transparent für Anwendungsprogramme.

Ein wichtiges Hilfsmittel zum Debuggen von Fiasco und L4-Programmen mittels JDB ist die Ereignisaufzeichnung. Daneben enthält JDB einen Disassembler und unterstützt die Anzeige von Threadzuständen, Speicherauszügen und Seitentabellen.

Zur Nutzung des Kerndebuggers von Nutzerprogrammen aus bietet JDB eine Systemerweiterung an: Das `int3`-Interface. Über diese Schnittstelle diente ursprünglich zur Zeichenausgabe auf die Konsole. Inzwischen kann JDB durch einen `int3`-Aufruf gezielt aktiviert werden. Auch der Disassembler erhält über dieses Interface Zeilen- und Symbolinformationen von den laufenden Programmen.

Ereignisaufzeichnung

Es wurden bereits verschiedene Möglichkeiten der Ereignisaufzeichnung implementiert [5, 6]. So können verschiedene Ereignisse, unter anderem IPC, Seitenfehler oder Systemaufrufe, mitgeloggt werden. Diese Ereignisse können interaktiv, unmittelbar beim Eintreten, manuell bestätigt und analysiert werden. JDB unterbricht dazu den Programm- und Systemablauf. Daneben existiert auch ein Ringpuffer zum Aufzeichnen mehrerer Ereignisse. Dieser kann zeitversetzt ausgewertet werden. Dabei können im Gegensatz zur interaktiven Analyse auch die Zeiten zwischen den Ereignissen gemessen werden.

[2] Zum Zeitpunkt der Compilierung wird festgelegt, ob der JDB mit in den Kern aufgenommen wird, oder nicht.

2.3 Common L4 Environment

Das L4 Common Environment (L4env) [4] ist eine Sammlung von Bibliotheken, welche die Anwendungsentwicklung unter Drops (bzw. unter L4 allgemein) unterstützen. Diese Bibliotheken kapseln grundlegende Funktionalität, die in allen Programmen benötigt wird:

- Verwaltung des Adressraumes (Region Mapper `libl4rm`)
- Verwaltung von Threads (Thread Library `libl4thread`)
- Bereitstellung von Semaphoren (Semaphore Library `libsemaphore`)
- Dynamisches Laden von Programmen (Loader)

Durch Nutzung von l4env muss sich der Anwendungsentwickler nicht mehr an der Basisfunktionalität (Woher kommt Speicher? Wo ist noch Platz im Adressraum? Welche Threads laufen, welche sind noch frei? ...) aufhalten, sondern kann sich auf seine Anwendung konzentrieren. Ein weiterer Vorteil von l4env ist die Unabhängigkeit der Schnittstellen vom API und ABI des verwendeten L4-Kerns. So können Applikationen leichter auf neue Prozessorarchitekturen beziehungsweise andere L4-Implementationen portiert werden.

Eine weitere Komponente von L4env, welche aber auch ohne die oben genannten Bibliotheken für nicht-l4env-Anwendungen genutzt werden kann, ist der Logserver mit seiner Bibliothek `liblog` bzw. `liblogserver`.

2.3.1 Logserver und Logclient

Der Logserver kann unter Drops die Bildschirmausgabe übernehmen. Dabei serialisiert er die Ausgaben verschiedener Programme, damit sie nicht ineinander vermischt werden, wie es bei der klassischen Zeichenausgabe durch den Kerndebugger der Fall sein kann. Ferner bietet er die Möglichkeit der Netzwerkübertragung der Bildschirmausgaben zur permanenten Speicherung auf anderen Rechnern.

Neben dieser Übertragung der Bildschirmausgabe werden in der Netzwerkversion noch binäre Datenkanäle, genannt Channels, angeboten. Der Logserver übernimmt dabei vor der Übertragung ein Multiplexing der verschiedenen Binärkanäle und Bildschirmausgaben.

Auf der anderen Seite der Netzverbindung übernimmt der Logclient ein entsprechendes demultiplexing des Datenstromes. Damit können die einzelnen Datenströme der Channels getrennt weiter verarbeitet oder angezeigt werden. Eine Aufbereitung der der Bildschirmausgaben vom Drops-System ist möglich.

Eine Alternative zum demultiplexing eines gespeicherten Datenstroms vom Logserver ist das neben diesem Beleg entstandene Programm Logdemux. Dieses Programm arbeitet schneller als der Logclient, bietet aber keine weiteren Funktionen an.

2.4 VAMPIR

VAMPIR wurde ursprünglich entwickelt am Zentralinstitut für Mathematik am Forschungszentrum Jülich. Die Software wird weiterentwickelt am Zentrum für

Hochleistungsrechnen der TU-Dresden [12] und von der Firma Pallas [7] vermarktet. VAMPIR dient der Anzeige und Analyse von Programmtraces, welche beispielsweise von der MPI Profiling Library Vampirtrace erzeugt wurden (siehe Abschnitt 2.1.2).

Die dabei auf jedem Knoten eines Parallelrechners entstandenen Tracefiles werden zur weiteren Analyse mit VAMPIR zusammengeführt und vorverarbeitet. VAMPIR übernimmt danach die grafische Anzeige der Daten, die Berechnung der Anzahl und Dauer von Funktionsaufrufen sowie der Nachrichenlaufzeiten.

2.4.1 Tracefiles

Die Tracefiles für VAMPIR bestehen aus zwei Teilen. Im ersten Teil werden mit den im Folgenden vorgestellten Primitiven die Eigenschaften des untersuchten Systems beschrieben.

CPU Der Term CPU beschreibt in VAMPIR je nach Anwendung einen Knoten eines Parallelrechners, einen Prozess oder einen Thread. Jede CPU besitzt eine eindeutige Id und befindet sich zu jedem Zeitpunkt in genau einem Zustand. Funktionsaufrufe auf einer CPU werden als Zustandänderungen abgebildet.

CLUSTER Ein CLUSTER ist eine Sammlung von einer oder mehreren CPUs. VAMPIR kann die Anzeige und Auswertung auf einzelne CLUSTER beschränken oder aber CLUSTER ausblenden.

STATE Der Zustand einer CPU beschreibt zu jedem Zeitpunkt, in welcher Funktion sich das Programm dort gerade befindet. Jeder Zustand ist genau einer Zustandsgruppe, in VAMPIR als ACTION bezeichnet, zugeordnet.

ACTION Eine Zustandsgruppe ist eine Sammlung von ähnlichen Zuständen. Diese Gruppen können von VAMPIR in unterschiedlichen Farben dargestellt und so voneinander abgegrenzt werden. Damit ist es möglich, die verschiedenen Funktionsgruppen eines Programms optisch zu erfassen und die interessanten Stellen gezielt näher zu betrachten.

MESSAGE Nachrichten stellen die Kommunikationsbeziehungen zwischen einer Sender-CPU und einer Empfänger-CPU dar. Nachrichten können zu

Gruppen zusammengefasst und bei der Darstellung ein- oder ausgeblendet werden.

Nach dieser Systembeschreibung folgen die Tracerecords. Jeder Record enthält dabei einen Zeitstempel und eine Aktion, die eine CPU betrifft. Die in der vorliegenden Arbeit verwendeten Aktionen sind Zustandsänderungen sowie Nachrichten.

Funktionsaufrufe werden durch eine Änderung des Zustands auf einer CPU dargestellt. Jede Zustandsänderung betrifft genau eine CPU. Beim Erzeugen eines Tracefiles ist dabei auf einen konsistenten Callstack zu achten, d.h. Funktionen müssen exakt in der umgekehrten Aufrufreihenfolge verlassen werden.

Nachrichten zeigen die Kommunikation zwischen zwei CPUs. Dabei werden für jede Kommunikation zwei Tracerecords benötigt: Ein Record auf der Sender-CPU, und ein Record auf der Empfänger-CPU. So können Nachrichtenlaufzeiten errechnet werden.

2.4.2 Darstellung & Analyse

Nach dem Start liest VAMPIR ein solches Tracefile ein und stellt die Ereignisse graphisch dar. Der Nutzer kann verschiedene Ansichten anwählen und zwischen ihnen beliebig umschalten. Die wichtigsten werden nun kurz beschrieben:

Timeline Die Timeline ist eine Darstellung des Gesamtsystems. VAMPIR öffnet dieses Fenster automatisch nach dem öffnen eines Tracefiles. Hier werden die Zustände aller CPUs und zwischen ihnen versendeten Nachrichten angezeigt. Einzelne CPUs können für die Darstellung einer Process Timeline selektiert werden.

Process Timeline In dieser Darstellung wird der Programmablauf auf einer CPU detailliert angezeigt. Funktionsaufrufe werden hier aber nicht nebeneinander, sondern vertikal versetzt angezeigt. So erhält der Nutzer einen Überblick über die Aufruftiefe der Funktionen seines Programms. Auch werden durch diese Darstellung kurze Funktionsaufrufe nicht von lang andauernden überdeckt.

Call Tree Im Funktionsaufrufbaum kann abgelesen werden, wie sich die einzelnen Funktionen eines Programms aufrufen. Dazu werden die Anzahl und Gesamtdauer der Aufrufe angezeigt.

Summary Chart Der Summary Chart stellt die Gesamtdauer von Zuständen oder Zustandsgruppen dar. Der Nutzer bekommt so einen Überblick über Aktivitäten seines Programms und deren Dauer. Die Darstellung ist zwischen Histogramm, Tortendiagramm oder Tabelle umschaltbar.

Der Detailgrad jeder Darstellung kann durch hineinzoomen beliebig variiert werden. Die Reihenfolge der angezeigten Daten kann jederzeit den eigenen Wünschen angepasst werden. Eine temporäre Filterung von ungewünschten Nachrichten, Prozessen oder Zuständen ist ebenfalls möglich. Für eine Anleitung zur Nutzung von VAMPIR sei auf [9] verwiesen.

Kapitel 3

Entwurf

In diesem Kapitel wird der Entwurf des Tracingsystems diskutiert. Dabei werden verschiedene Lösungskonzepte vorgestellt und gegeneinander abgewogen. Die konkrete Implementation ist Gegenstand des nächsten Kapitel.

3.1 Entwurfsziele

Das zu entwerfende Tracingsystem soll folgenden Anforderungen genügen:

- Bestehende Programme sollen nur minimal verändert werden. Dadurch bleibt das Tracingsystem benutzbar und das instrumentierte Programm wartbar.
- Im Kern soll auf die bestehende Infrastruktur zurückgegriffen werden. Neue Schnittstellen vergrößern FIASCO unnötig, liegen außerhalb der L4-Spezifikation und erschweren eine Portierung auf neue Architekturen.
- Auswertung und Datensammlung sollen getrennt sein. Die Analyse des Programmablaufs direkt unter L4 ist weder notwendig noch sinnvoll. Bei einer zeitversetzten Auswertung können auch die Ergebnisse mehrerer Programmläufe nebeneinander dargestellt und miteinander verglichen werden.

3.1.1 Architektur

Ausgehend von den Entwurfszielen und den bereits vorhandenen Möglichkeiten des *gcc* und des FIASCO JDB ergeben sich die im folgenden beschriebenen Architekturmerkmale. Auf die einzelnen Teile des Tracingsystems wird dann in den folgenden Abschnitten genauer eingegangen.

Für das Tracing von Nutzerprogrammen wurde die Bibliothek **tracelib** entwickelt. Sie enthält alle Funktionalität, die für das Tracing von Anwendungsprogrammen nötig ist.

Die zur detaillierten Analyse des Systemverhaltens benötigten Kerndaten, wie Nachrichten oder Scheduling, werden mit dem vorhanden JDB erzeugt. Dafür wurde die Logging-Infrastruktur von JDB erweitert. Für die Kommunikation mit JDB und die Weiterleitung der Kerntracingdaten wurde der L4-Server **ktraced** implementiert.

Alle im L4-System gesammelten Tracingdaten aus Nutzerprogrammen und dem Kern werden zu einem anderen Rechner übertragen. Dort werden diese Daten gesichert, weiter verarbeitet und aufbereitet. Die dazu notwendigen Werkzeuge wurden ebenfalls implementiert.

3.2 Nutzerprogrammtracing

Nutzerprogammtracing, also die Aufzeichnung von Funktionsaufrufen innerhalb einzelner Threads, kann vollständig im Nutzeradressraum und unabhängig vom verwendeten Kern geschehen. Es müssen lediglich die Tracingdaten über Funktionsaufrufe gesammelt und bis zur Weiterverarbeitung vorgehalten werden.

Die in Kapitel 2 vorgestellten Tracingmethoden 1 und 2 erfordern umfangreiche Änderungen am Quelltext, während bei Methode 3 lediglich geringfügige Änderungen zum Initialisieren und Beenden des Tracingsystems notwendig sind. Diese Tracingart wird außerdem auch vom verwendeten *gcc* unterstützt. Somit ist sie im Hinblick auf das Entwurfsziel *Minimale Veränderung des untersuchten Programms* hervorragend für das Tracing im Nutzeradressraum geeignet.

Es wurde dazu eine Bibliothek mit der notwendigen Funktionalität entwickelt. Diese Bibliothek enthält neben den vom *gcc* aufgerufenen Funktionen zur Erzeugung von Tracerecords auch Funktionen zur Tracingkontrolle und verwaltet die Puffer zur Datensammlung. Zur Aufzeichnung von Kerntracingdaten wendet sich die Bibliothek an den in Abschnitt 3.3 vorgestellen Server.

3.2.1 Tracingkontrolle

Die Bibliothek benötigt Informationen über das instrumentierte Programm. Ebenso werden die Größe und die Lage der einzelnen Tracepuffer benötigt, da innerhalb der Bibliothek keine Annahmen über das untersuchte Programm gemacht werden können. Deshalb soll die Bibliothek solange inaktiv sein, bis sie vom Programm initialisiert wird. Dazu muss vom Programm die Funktion `trace_init()` einmal aufgerufen werden. Auf Anforderung wird dabei auch eine Verbindung zum Traceserver (Abschnitt 3.3.2) hergestellt. Erst nach diesem Aufruf können die anderen Funktionen der Bibliothek genutzt werden.

Sobald die Initialisierung abgeschlossen ist, kann das Tracing der Applikation jederzeit aktiviert oder wieder unterbrochen werden. Dazu stellt die Bibliothek die Funktionen `trace_start()` und `trace_stop()` zur Verfügung. Beide Funktionen arbeiten global für alle Threads einer Anwendung.

Schließlich muss das Tracingsystem am Programmende ebenfalls beendet werden. Dabei müssen alle noch nicht übertragenen Inhalte der Tracepuffer gesichert werden. Diese Aufgabe übernimmt die Funktion `trace_done()`, welche vor dem Programmende einmal aufzurufen ist.

3.2.2 Pufferverwaltung

Die gesammelten Tracingdaten eines Threads müssen im System zwischengespeichert werden, da direktes Übertragen beim Generieren der Tracerecords zu aufwändig und zu teuer ist. Deshalb werden die Daten im System gepuffert und später, wenn der benutzte Puffer keine weiteren Tracerecords mehr aufnehmen kann, zur Übertragung weitergeleitet. Für eine effizienteVerwaltung und Weiterverarbeitung besteht der Tracepuffer aus einem zusammenhängenden Speicherbereich. Dort werden die Tracerecords nacheinander abgelegt.

Zur Zwischenspeicherung können eigene Puffer pro Prozess oder eigene Puffer pro Thread benutzt werden. Bei ersteren ist beim Pufferzugriff wechselseitiger Ausschluss zu gewährleisten. Eine mögliche Lösung dafür ist die atomare Recordallokation mittels unteilbarer Befehle (z.B. `cmpxchg`[1]). Allerdings kann das prioritätengesteuerte Scheduling von L4/Fiasco dazu führen, dass ein niederpriorer Thread beim füllen seines Tracerecords von Threads mit hoher Priorität für beliebig lange Zeit unterbrochen wird. Die Folge davon sind einerseits unvollständige Puffereinträge und andererseits das Überschreiben von Einträgen anderer Threads, wenn die Unterbrechung länger dauert als die komplette Füllung des Puffers.

Die beschriebenen Probleme treten nicht auf, wenn ein eigener Puffer für jeden Thread verwendet wird. Dabei können auch je nach Art des Threads unterschiedlich große Puffer, beispielsweise große Puffer für Arbeitsthreads, aber kleine für Interruptthreads, verwendet werden.

Die Bereitstellung der Puffer für die einzelnen Threads erfolgt bei der ersten Benutzung. Eine andere Möglichkeit wäre die Pufferallokation während der Initialisierung der Bibliothek. Dabei sind aber Puffer für alle Threads anzulegen, welche die Applikation jemals verwenden könnte. Dies führt zu erhöhtem Speicherverbrauch, wenn keine speziell auf diese Applikation angepasste Pufferallokation verwendet wird, sondern nur eine generische. Daher wurde diese Idee wieder verworfen.

Als Tracepuffer erwartet die Bibliothek eine Flexpage. Somit beträgt die minimale Puffergröße 4 KB. Für l4env-Anwendungen wird eine generische Allokationsfunktion bereitgestellt. Bei Anwendungen, die nicht das L4 Common Environment verwenden, muss vom Bibliotheksnutzer eine Allokationsfunktion für die Tracepuffer zur Verfügung gestellt werden (Siehe dazu auch Abschnitt 4.1.5).

Am Anfang jedes Tracepuffers sind 16 Byte Verwaltungsdaten für die Weiterverarbeitung enthalten. Darin wird die L4-Threadid, die Pufferversion und die Menge der nachfolgenden Tracingdaten gespeichert.

3.2.3 Tracerecords

Die anfallenden Tracingdaten müssen strukturiert in den Tracepuffern abgespeichert werden. Bei Records fester Größe ergeben sich beim Zugriff leichte Geschwindigkeitsvorteile durch einfachere Adressierung und Überlaufprüfung. Dabei sind jedoch Kompromisse bei der Bestimmung der optimalen Recordgröße einzugehen. Sonst ist nicht genügend Speicherplatz für umfangreichere Daten vorhanden, oder es bleibt bei geringen Datenmengen zu viel Speicherplatz durch interne Fragmentierung ungenutzt.

Dieses Problem tritt bei Records variabler Größe nicht auf, deswegen wurden diese implementiert. Als minimale Größe und Granularität wurden dabei 16 Byte gewählt. Darin können alle minimal notwendigen Daten für ein Ereignis abgespeichert werden: 8 Byte Zeitstempel, 4 Byte Daten zu diesem Ereignis, 1 Byte Ereignistyp und 3 Byte zusätzliche Daten. Diese Größe entspricht auch

[1] compare and exchange, auch compare and swap

genau einer halben L1-Cacheline[2]. Dadurch ist die Performance beim Zugriff hervorragend, denn alle folgenden Speicherzugriffe auf den Tracerecord finden auf ausgerichtete Adressen statt. Durch eine Granularität von 16 Byte wird die interne Fragmentierung des Tracepuffers eingeschränkt.

Die maximale Größe für einen Tracerecord beträgt knapp 4 KB, die minimale Tracepuffergröße abzüglich der oben beschriebenen Verwaltungsdaten, damit ein Record stets vollständig in den Tracepuffer passt. Es kann also keine Records geben, die größer als der Tracepuffer sind.

Der erste Record

Zur weiteren Verarbeitung der Tracingdaten mit den in Abschnitt 3.5 beschriebenen Werkzeugen sind Informationen über das laufende Programm nötig. Diese müssen ebenfalls gesichert und übertragen werden. Daher wird direkt nach der Pufferallokation ein spezieller Datensatz im Tracepuffer abgelegt welcher die Programmidentifikation enthält. Das genaue Format dieses Datensatzes ist in Abschnitt 4.1.6 beschrieben.

3.3 Kerntracing

Beim Kerntracing werden Daten zu Ereignissen gesammelt, welche im Nutzermodus nicht oder nur unvollständig zur Verfügung stehen. Dies sind Informationen zur Threadumschaltung, zum Scheduling, das Eintreffen von Interrupts oder Seitenfehlern und deren Behandlung, sowie die IPC-Kommunikation zwischen Threads.

Zur Aufzeichnung dieser Daten wurde FIASCO's Kerndebugger JDB erweitert. Es wurde ein L4-Server als Schnittstelle zum JDB implementiert. Alle Anwendungen, welche Tracingdienste des JDB in Anspruch nehmen wollen, wenden sich an diesen Server.

3.3.1 Kerndebuggererweiterungen

Die bereits vorhandene Logging- bzw. Tracinginfrastruktur des JDB wurde um folgende Eigenschaften erweitert:

- Export des JDB-Tracepuffers in den Nutzeradressraum, damit die Kernereignisse zur zeitversetzten Weiterverarbeitung gesichert werden können, beispielsweise durch den Versand per Netzwerk. Der Tracepuffer ist in allen Adressräumen sichtbar. Die Adresse des Puffers kann durch einen `int3`-Systemaufruf festgestellt werden.

- Die Größe des Puffers ist während des Systemstarts vom Nutzer festlegbar. Für eine effiziente Füllstandserkennung wird die angegebene Größe auf die nächstgrößere Flexpage aufgerundet.

- Erzeugen einer Nachricht bei vollem Puffer[3], damit die Nutzer des Puffers ohne ständiges Pollen über den Füllstand des Puffers informiert werden.

[2] Bei Intel Pentium Prozessoren beträgt die Größe einer Cacheline im L1-Cache 32 Byte. Bei AMD Athlon Prozessoren beträgt sie jedoch 64 Byte, es passen dann vier einfache Records in eine Cacheline.

[3] Genauer: beim Umschalten auf die andere Pufferhälfte.

Zur Nachrichtenerzeugung bei vollem Puffer wurde das IRQ-System von FIASCO um *virtual IRQs* erweitert. Diese sind im Gegensatz zu den bereits vorhandenen *device IRQs* von der Hardware unabhängig und dienen der asynchronen Benachrichtigung von Nutzerprogrammen durch den Kern.

- Zweiteilung des Tracepuffers, damit eine Hälfte vom Kern gefüllt werden kann, während die andere Hälfte weiter verarbeitet wird.
- Atomares Erzeugen von Tracerecords, damit keine Inkonsistenzen (Halb ausgefüllte Records) durch Unterbrechung entstehen.
- Neue Ereignistypen für IPC-Tracing sowie Schedulingereignisse.
- Kontrollsequenzen für den JDB können nun auch nicht-interaktiv von Programmen gegeben werden. So kann das Tracing von Kernereignissen jederzeit mittels `int3`-Interface an- oder abgeschaltet werden.

3.3.2 Traceserver

Der Kern sammelt die Tracingdaten in einem Ringpuffer, sichert diesen aber nicht dauerhaft. Für die Weiterleitung der Tracingdaten des Kerns und die damit verbundenen Aufgaben wurde der L4-Server **ktraced** entwickelt.

Dieser Server erfragt nach dem Start vom Kerndebugger (mittels `int3`-Interface) die Position und Größe der beiden Kerntracepuffer. Dann startet er einen hochprioren Thread, welcher sich zur Weiterleitung der Kerntracepuffer an den bereitgestellten *virq* verbindet.

Neben der Weiterleitung der Tracingdaten stellt dieser Server für andere Programme eine IPC-Schnittstelle zu den Tracing- und Loggingdiensten des JDB zur Verfügung. So wenden sich alle Programme, welche die Aufzeichnung von Kernereignissen wünschen, an **ktraced**. Dieser sorgt dann für die Synchronisation der verschiedenen Anforderungen einzelner Programme. Das IPC-Protokoll wird mit seinen Parametern in Abschnitt 4.2 genau beschieben.

3.4 Datentransfer

Nach der Sammlung der Tracedaten müssen diese direkt weiter verarbeitet oder aber zur späteren Weiterverarbeitung gesichert werden.

Die einfachste Methode zur Sicherung ist die Übertragung über die serielle Schnittstelle (z.B. mittels Kerndebugger) zu einem anderen Rechner. Es stellte sich jedoch sehr schnell heraus, dass die Datenrate zu gering ist um selbst kleinere Tracepuffer zu übertragen[4]. Deshalb wurde diese Idee sehr schnell wieder verworfen.

Die entstehenden Daten lassen sich auch lokal auf Festplatte speichern. Dies setzt jedoch voraus, dass es einen zusagenfähigen Treiber für den jeweiligen Festplattencontroller gibt. Derzeit ist dies nur für die NCR-SCSI-Controller der Fall und somit nicht weit verfügbar.

Deshalb wurde die Datenübertragung vom L4-System via Netzwerk auf einen anderen Rechner gewählt. Als Netzwerktreiber wird der Logserver verwendet,

[4] Die serielle Datenrate beträgt maximal 115 Kb/s (knapp 15 Kilobyte pro Sekunde), es entstehen aber beispielsweise beim `pingpong`-Benchmark mehrere Megabyte pro Sekunde.

da er eine große Auswahl an Netzwerkkartentreibern unterstützt. Auch bietet er neben der reinen Textübertragung von Bildschirmausgaben binäre Kanäle zur Übertragung beliebiger Daten an. Dabei werden alle Kanäle und Bildschirmausgaben über eine einzige TCP-Verbindung übertragen. Auf der Gegenseite wird dieser Datenstrom aufgezeichnet (beispielsweise mittels `netcat`) und in einer Datei abgespeichert. Dann wird diese Datei vom Logclient oder von Logdemux zur weiteren Verarbeitung wieder in die einzelnen Kanäle aufgeteilt.

Die Bibliothek **tracelib** nutzt für alle Applikationen einen einzigen Kanal. Aus dem in jedem Tracepuffer enthaltenen Header (siehe 3.2.2) können die Auswertungswerkzeuge die Zuordnung von Puffern zu L4-Threads rekonstruieren.

ktraced nutzt einen eigenen Kanal, da hier kein Platz für einen zusätzlichen Header vor den Tracingdaten ist.

3.5 Werkzeuge

Nachdem die Tracingdaten gesammelt und auf einen anderen Rechner übertragen wurden, müssen sie zur weiteren Verarbeitung mit VAMPIR aufbereitet werden. Dazu wurde das Werkzeug **convtrace** entwickelt.

convtrace übernimmt die Konvertierung der binären Tracingdaten in ein für VAMPIR geeignetes Format. Dazu werden die Dateien mit Nutzerprogrammtraces und das dazugehörige Kerntrace schrittweise eingelesen.Die Tracingdaten werden auf die in Abschnitt 2.4 vorgestellten VAMPIR-Primitive abgebildet.

Zur weiteren Analyse und Darstellung werden Symbolinformationen des untersuchten Programms benötigt, da die in den Tracerecords gespeicherten Adressen nicht sehr aussagekräftig sind. Die Zuordnung einer Funktion zu ihre Adresse kann jedoch aus dem Programm selbst herausgelesen werden.

3.6 Zusammenfassung

In diesem Abschnitt werden noch einmal die wichtigsten Entwurfsentscheidungen zusammengefasst.

Nutzerprogramme werden automatisch durch *gcc* instrumentiert. Es wurde die Bibliothek **tracelib** entwickelt, welche die entstehenden Tracingdaten in einem Tracepuffer pro L4-Thread zwischenspeichert, Auf Anforderung vom Programm, spätestens jedoch, wenn sie keine weiteren Tracerecords mehr aufnehmen können, werden die Tracepuffer über das Netzwerk zu einem anderen Rechner übertragen.

Die Aufzeichnung der IPC-Kommunikation und Kontextwechsel übernimmt FIASCO's Kerndebugger. JDB zeichnet diese Ereignisse in seinem Tracepuffer auf und informiert nach erfolgter Füllung den Traceserver **ktraced**. Dieser Server überträgt die Tracingdaten dann ebenfalls via Netzwerk zu einem anderen Rechner. **ktraced** dient auch als Schnittstelle zu den Tracingdiensten von JDB.

Nach der Übertragung werden die Tracingdaten von **convtrace** aufbereitet und in ein VAMPIR-Tracefile konvertiert. Die Auswertung und Analyse des Programmablaufs wird mit VAMPIR durchgeführt.

Kapitel 4

Implementation

In diesem Kapitel wird auf verschiedene Implementationsdetails der Arbeit eingegangen. Dabei wird zuerst das API und die Nutzung der neu entwickelten Bibliothek vorgestellt. Im zweiten Abschnitt wird auf die Schnittstelle zum Server **ktraced** eingegangen. Dabei werden auch die Veränderungen am FIASCO-Kern beschrieben. Der letzte Abschnitt erklärt die Umwandlung und Aufbereitung der Tracingdaten in das VAMPIR-Format.

4.1 Die Bibliothek tracelib

Die Bibliothek **tracelib** stellt ein API zur Verfügung, über welches das Programm das Verhalten der Bibliothek und somit des Tracings beeinflussen kann.

4.1.1 Initialisierung

Gleich nach Programmstart muss die Bibliothek vom Nutzer initialisiert werden. Dafür wird die Funktion `trace_init(int flags, const char *ident)` bereitgestellt. Diese Funktion liefert nach erfolgter Initialisierung 0 zurück, im Fehlerfall einen Wert ungleich 0. Erst nach erfolgreichem Aufruf von `trace_init()` können die anderen Funktionen der Bibliothek genutzt werden. Vorher kehren diese ohne Fehlermeldung zurück, da es beim automatischen Instrumentieren möglich ist, dass bereits beim Programmstart instrumentierte Funktionen aufgerufen werden.

Der Parameter `ident` enthält einen String zur Programmbeschreibung. So kann **convtrace** (siehe 4.4) während der Auswertung die Tracingdaten einem Programm zuordnen.

Mit `flags` wird der Kontakt zum Kerntraceserver gesteuert. Es können folgende Konstanten angegeben werden:

TRACE_NO_FLAGS Keine Veränderung des Standardverhaltens. Es wird eine Verbindung zum Traceserver hergestellt. Wenn kein kein **ktraced** vorhanden ist, oder wenn er keine weiteren Klienten mehr akzeptiert, dann ist der Rückgabewert von `trace_init()` ungleich 0.

TRACE_NO_SERVER Es wird keine Verbindung zu **ktraced** aufgenommen. In diesem Modus ist Kerntracing deaktiviert, es werden nur Ereignisse im Nutzeradressraum aufgezeichnet.

TRACE_ALLOW_SERVERFAIL Die Bibliothek versucht eine Verbindung zum Traceserver herzustellen, um Kerntracingdienste zu nutzen. Schlägt der Verbindungsaufbau fehl, weil kein **ktraced** vorhanden ist, oder weil er keine weiteren Klienten mehr akzeptiert, dann arbeitet die Bibliothek ohne Server.

4.1.2 Beendigung

Die Funktion `trace_done()` dient zum Beenden der Tracingbibliothek. Dabei wird das Tracing für diesen Prozess beendet und alle noch in den Tracepuffern verbliebenen Daten werden an das Hostsystem übertragen. Eine bestehende Verbindung zum Traceserver **ktraced** wird beendet.

Nach dem Aufruf vom `trace_done()` haben alle Funktionen der **tracelib** keine Wirkung mehr und kehren ohne Fehlermeldung zurück.

4.1.3 Aktivieren und Deaktivieren des Tracings

Die Bibliotheksfunktionen `trace_start()` aktiviert das Tracing des Nutzerprozesses. Bei bestehender Verbindung zum **ktraced** wird diesem mitgeteilt, dass die Kerntracingdienste ab sofort benötigt werden.

Analog dazu deaktiviert die Funktion `trace_stop()` das Tracing im Nutzerprozess und im Kern.

Diese beiden Funktionen arbeiten auf Prozessebene, sie beeinflussen somit alle Threads dieses Prozesses.

Achtung: Es ist vom Anwender sicher zu stellen, dass das Tracing immer auf der selben Funktionsaufrufsebene deaktiviert wird, auf der es aktiviert wurde. Diese Forderung ist nötig, da bei deaktiviertem Tracing keine Informationen über die Tiefe der Funktionsaufrufe gesammelt werden können. Ein Abweichen von dieser Forderung führt während der Auswertung mit VAMPIR zu falschen Ergebnissen.

4.1.4 Threadidentifikation

In jeder Funktion, welche auf den Tracepuffer zugreift, wird die Thread-Id des aktuellen Threads benötigt. Diese Information liegt jedoch nirgends permanent vor[1], da innerhalb der Bibliothek keinerlei Annahmen über das laufende Programm und dessen Struktur gemacht werden können.

Daher ist vom Bibliotheksnutzer die Funktion `traceifl4_get_id()` bereitzustellen. Diese Funktion liefert die eindeutige Id des aktuellen Threads der Anwendung zurück. Das Ergebnis muss im Bereich 0...127 liegen[2]. Verschiedene Threads müssen unterschiedliche Ids haben. Diese Funktion darf, wie alle anderen von der Bibliothek aufgerufenen Funktionen auch, nicht zum Tracing instrumentiert sein. Sie sollte außerdem sehr schnell arbeiten, da sie sehr häufig aufgerufen wird.

Für l4env-Anwendungen wird eine Implementation mitgeliefert, die sich auf die in l4env enthaltene Threadbibliothek stützt.

[1] Es könnte bei jedem Ereignis der Systemaufruf `l4_myself()` durchgeführt werden, das ist aber viel zu teuer.

[2] Die derzeit verwendete L4-Version unterstützt 128 Threads pro Task.

4.1.5 Supportfunktionen für Tracepuffer

Jeder Thread einer Applikation kann den Inhalt seines Tracepuffers jederzeit durch Aufruf der Funktion `trace_flush()` zum Remote-Rechner übertragen, um mit einem leeren Puffer weiter zu arbeiten. So kann der Puffer gezielt geleert werden, damit nachfolgende Funktionsaufrufe ohne Unterbrechung aufgrund eines vollen Puffers durchgeführt werden können.

Wie bereits in Abschnitt 3.2.2 beschrieben, benötigt die Bibliothek für jeden Thread einen eigenen Tracepuffer. Innerhalb der **tracelib** können keinerlei Annahmen über den Adressraum der untersuchten Anwendung gemacht werden. Zur Allokation der Tracepuffer muss deshalb vom Nutzer die Funktion `traceifl4_get_buffer(l4_int32_t threadid)` bereitgestellt werden. Diese Funktion liefert für die ihr übergebene Thread-Id eine Flexpage zurück, welche Position und Größe des Tracepuffers beschreibt. Im Fehlerfall muss diese Funktion den Wert 0 (null) zurückgeben, die Bibliothek bricht das Programm in diesem Fall ab.

4.1.6 Recordtypen

Die Bibliothek benutzt, wie in Abschnitt 3.2.3 beschrieben, Tracerecords variabler Größe. Jeder Record beginnt dabei mit folgender Struktur:

```
struct simple_event
{
  l4_uint64_t timestamp;  /* Zeitstempel */
  l4_uint32_t data;       /* Daten       */
  l4_uint8_t type;        /* Eventtyp    */
  l4_uint8_t add[3];      /* Zusatzdaten */
};
```

Während der Auswertung der entstandenen Traces kann anhand des Feldes `type` (eventuell unter Zuhilfenahme von `data` oder `add`) die erzeugte Recordgröße rekonstruiert werden.

Derzeit werden von der entstandenen Bibliothek vier verschiedene Recordtypen erzeugt.

Typ	Größe	Beschreibung
TRACEEVENT_NOP	simple_event	Leerer Record
TRACEEVENT_ENTER	simple_event	Funktionseintritt
TRACEEVENT_LEAVE	simple_event	Funktionsende
TRACEEVENT_INTRO	variabel	Intro Daten

Tabelle 4.1: Recordtypen

TRACEEVENT_ENTER

Dieses Ereignis wird bei jedem Funktionseintritt in eine instrumentierte Funktion aufgezeichnet. Im Feld `data` wird die Adresse der Funktion abgelegt.

TRACEEVENT_LEAVE

Am Funktionsende, vor der Rückkehr zum Aufrufer, wird bei instrumentierten Funktionen dieses Ereignis erzeugt. Hier wird im Feld `data` die Adresse der Funktion, welche gerade verlassen wird, abgelegt.

TRACEEVENT_INTRO

Dieses spezielle Ereignis wird in jedem neu allozierten Tracepuffer abgelegt (siehe auch Abschnitt 3.2.3 auf Seite 16). Zur Programmbeschreibung enthält dieser Record den String, der der Bibliothek bei `trace_init()` übergeben wurde. Damit erhalten die Auswertungsprogramme Informationen über das untersuchte Programm. Da der String eine variable Länge hat, wird im Feld `data` die Länge der nachfolgenden Zeichenkette gespeichert.

4.1.7 Recordallokation

Zur Allokation von Tracerecords werden zwei Funktionen zur Verfügung gestellt:

```
trace_get_rec(unsigned size)
trace_simple_rec(l4_uint32_t data, l4_uint_8t type)
```

Die allgemeine Funktion `trace_get_rec()` liefert, wenn Tracing aktiviert ist, einen Zeiger auf ein Pufferstück der Größe `size` zurück. Hier kann nun der Tracerecord eingetragen werden. Im Fehlerfall, d.h. es wurde mehr als die maximale Recordgröße verlangt, oder wenn Tracing deaktiviert ist, liefert sie das Ergebnis `NULL`. Auch dieser Fall ist vom Aufrufer geeignet zu behandeln.

Für einfache Records des Typs `simple_event` steht für die Allokation und Füllung die Funktion `trace_simple_rec()` bereit. Bei aktivem Tracing allokiert diese Funktion einen einfachen Tracerecord, setzt den Zeitstempel und überträgt die beiden übergebenen Werte in den Record[3]. Bei deaktiviertem Tracing kehrt diese Funktion zurück, ohne einen Tracerecord zu erzeugen.

Beide Funktionen leeren den Puffer, wenn nicht mehr genügend Platz für die angeforderte Größe vorhanden ist.

4.1.8 Nutzung der Bibliothek

Zum Tracing eigener Applikationen sind nur wenige Anpassungen am Quelltext und dem `Makefile` nötig. Im `Makefile` des Programms ist zu den `CFLAGS` die *gcc*-Option `-finstrument-functions` und zu `LIBS` die Tracingbibliothek `-ll4trace` hinzuzufügen. Am Anfang der `main()`-Funktion sind die oben beschriebenen Aufrufe von `trace_init()` und `trace_start` einzufügen, damit die Bibliothek initialisiert und das Tracing gestartet wird. Am Ende von `main()` muss zum Beenden der **tracelib** ein Aufruf von `trace_stop()` eingefügt werden. Danach ist das Programm neu zu übersetzten.

Für nicht-l4env-Anwendungen sind zusätzlich die weiter oben beschriebenen Supportfunktionen für Tracepufferallokation und Threadidentifikation zu implementieren.

[3]**Bemerkung:** Das Feld `add[3]` bleibt unverändert, kann demzufolge alte Daten enthalten. Dies ist bei der späteren Auswertung zu beachten.

Instrumentierung bestimmter Funktionen verhindern

Wenn Funktionen von der automatischen Instrumentierung ausgeschlossen werden sollen (beispielsweise weil sie von der **tracelib** aufgerufen werden, oder sie sehr oft aufgerufen werden und daher zu viele unnütze Tracerecords erzeugen würden), dann sind diese Funktionen speziell zu markieren. Dazu bietet *gcc* das Funktionsattribut `no_instrument_function` an. Funktionsattribute sind eine C-Erweiterung von *gcc*. Die Syntax von Attributdefinitionen ist in [15] beschrieben. Zur Kapselung und einfacheren Verwendung wird in `tracelib.h` das Makro `TRACE_NO_INST_FUNC` definiert. Ein Funktionsprototyp sieht dann wie folgt aus:

```
int trace_init(int flags, const char *ident) TRACE_NO_INST_FUNC;
```

Inline Funktionen

Inline Funktionen, die als `extern inline` deklariert werden, benötigen bei instrumentierten Programmen eine gesonderte Behandlung: Der *gcc* erzeugt für diese Funkionen keine eigene Instanz mit einem Funktionsnamen. Daher bringt dann der Linker eine Fehlermeldung, wenn er die Adresse dieser Funktionen ermitteln möchte.

Eine mögliche Lösung für dieses Problem ist die Umwandlung dieser Funktionen nach `static inline`. Bei Nutzung des Makros `L4_INLINE` ist dazu die Präprozessoroption `-DSTATIC_L4_INLINE` anzugeben. Eine Alternative ist die Bereitstellung von Dummy-Instanzen.

In der **tracelib** werden für die Inline-Funktionen der Pakete *l4sys*, *l4util* und *semaphore* solche Dummies bereitgestellt. Die Angabe von `-DSTATIC_L4_INLINE` ist dafür also nicht nötig.

4.2 Der Traceserver

Der Server **ktraced** stellt das Interface zu Fiasco's Kerndebugger JDB. Alle Threads, welche Tracingdienste des Kerndebuggers in Anspruch nehmen wollen, wenden sich an **ktraced**. Dieser signalisiert dann dem Kerndebugger, welche Ereignisse zu Tracen sind, und welche nicht bzw. nicht mehr.

Die Funktionen des Servers, und somit die Konfiguration der Tracingdienste des Kerndebuggers, können über ein einfaches Short-IPC-Protokoll in Anspruch genommen werden. Dieses Protokoll wird im Folgenden beschrieben.

Jede Nachricht an **ktraced** besteht aus zwei Worten. Dazu werden ersten Wort (`msg.w0`) das gewünschte Protokoll, die Subfunktion und eventuelle Optionen übergeben. Diese Aufteilung von `msg.w0` wurde vom L4-Resourcenmanager `rmgr` übernommen. Im zweiten Wort (`msg.w1`) werden funkionsspezifische Parameter übergeben.

Die Antwort von **ktraced** besteht aus einem Wort (`msg.w0`). Wenn nicht anders beschrieben, dann zeigt ein Rückgabewert vom 0 an, dass die angeforderte Aktion erfolgreich durchgeführt werden konnte. Ein Rückgabewert ungleich 0 bedeutet, dass die Bearbeitung der Anfrage fehlgeschlagen ist.

4.2.1 "HELLO" Protokoll

Mit dem Hello-Protokoll kann eine Task herausfinden, ob sie mit dem **ktraced** oder mit einem anderen Server kommuniziert. Weiterhin können sich Anwendungen mit diesem Protokoll registrieren, um danach das TRACECTRL-Protokoll zu benutzen. Sollte die Anwendung keine weiteren Dienste des **ktraced** mehr benötigen, dann kann sie sich mit diesem Protokoll auch deregistrieren.

Funktion	Beschreibung
HELLO	**ktraced** kontaktieren
REGISTER	Task registrieren
UNREGISTER	Task deregistrieren

Tabelle 4.2: HELLO Protokoll

HELLO_HELLO

Durch die HELLO_HELLO-Nachricht können Programme feststellen, ob sie mit dem **ktraced** kommunizieren oder mit einem anderen Server. **ktraced** antwortet auf diese Nachricht mit einer "Magic Number" (definiert in `ktraced.h`) und seiner Version.

Dieser Nachricht kann die Option "REGISTER" mitüberreicht werden. Damit entfällt im Erfolgsfall der sonst notwendige Aufruf von HELLO_REGISTER.

HELLO_REGISTER

Durch diese Nachricht registriert sich ein Klient bei **ktraced** und kann danach die anderen Protolle verwenden. Diese Nachricht schlägt Fehl, wenn **ktraced** keine weiteren Klienten akzeptiert.

HELLO_UNREGISTER

Klienten, welche die Dienste des **ktraced** nicht mehr in Anspruch nehmen, deregistrieren sich mit dieser Nachricht. Dabei werden alle von diesem Klienten angeforderten Tracingdienste gestoppt.

4.2.2 "TRACECTRL" Protokoll

Dieses Protokoll dient zum geordneten Aktivieren und Deaktivieren von Tracingdiensten des Kerns. Dabei wird sichergestellt, dass alle angeforderten Tracingdienste (sofern sie sich nicht gegenseitig ausschließen) im Kern aktiviert sind, solange sie von mindestens einem Klienten benötigt werden.

Für alle Funktionen dieses Protokolls werden in `msg.w1` die Dienste als Bitmaske angegeben, auf welche diese Funktion anzuwenden ist. Diese Maske entsteht durch ODER-Verknüpfung der in Tabelle 4.4 angegebenen Konstanten aus `ktraced.h`.

Funktion	Beschreibung
START	Startet Tracingdienste
STOP	Beendet Tracingdienste

Tabelle 4.3: TRACECTRL Protokoll

Name	Aufzeichnung von
KTRACE_EVENT_IPC	IPC-Tracinginformationen
KTRACE_EVENT_PAGEFAULT	Seitenfehlern
KTRACE_EVENT_UNMAP	`l4_fpage_unmap()`-Operationen
KTRACE_EVENT_CONTEXT_SWITCH	Threadumschaltungen
KTRACE_EVENT_IPC_SHORTCUT	Nutzung des IPC-Shortcuts
KTRACE_EVENT_IRQ_RAISED	Eintreffen von IRQs
KTRACE_EVENT_TIMER_IRQ	Uhrticks
KTRACE_EVENT_THREAD_EX_REGS	`l4_thread_ex_regs()`-Aufrufe

Tabelle 4.4: Tracingdienste des Kerns

TRACECTRL_START

Ein Klient, welcher Tracingdienste des Kerns in Anspruch nehmen möchte, sendet dem **ktraced** eine TRACECTRL_START-Nachricht mit den gewünschten Tracingdiensten. Der Server überprüft die Anforderung und sendet die entsprechenden Kommandos an den JDB.

Dabei werden bereits aktivierte Tracingdienste anderer Tasks nicht noch einmal aktiviert, sondern nur neu hinzugekommene.

TRACECTRL_STOP

Wenn ein Klient einen oder mehrere Tracingdienste nicht mehr benötigt, dann sendet er diese Nachricht an **ktraced**.

Der Server überprüft nun, ob die zu deaktivierenden Dienste noch von anderen Tasks angefordert wurden. Wenn der Aufrufer der letzte noch aktive Klient für einen Dienst war, dann sendet **ktraced** entsprechende Kommandos zum deaktivieren an den JDB.

4.3 Veränderungen am Kern

Hier wird genauer auf verschiedene Aspekte der Änderungen am FIASCO-Kern und dabei auftretende Probleme und deren Lösung eingegangen.

4.3.1 Tracingpfad

In einer ersten Implementation des IPC-Tracings wurde der vorhandene IPC-Loggingpfad erweitert, damit pro IPC nur noch ein Datensatz entsteht. Dieser enthält im Gegensatz zum bereits vorhandenen Logging sowohl Informationen zum Sende- als auch zum Empfangsteil. Bisher entstanden beim IPC-Logging bis zu zwei Records, einer für die Sendeoperation und optional einer für die Empfangsoperation.

Es stellte sich jedoch heraus, dass diese Implementation noch recht langsam ist. Dadurch ändert sich das Zeitverhalten und somit auch das Scheduling zwischen verschiedenen Threads erheblich. So treten womöglich die zu beobachtenden und aufzuzeichnenden Anomalien nicht mehr auf.

Zur Verminderung der Aufzeichnungszeit wurde ein eigener Tracingpfad entwickelt, welcher unabhängig vom bestehenden Loggingpfad ist. Zur weiteren

Beschleunigung wurde dieser Pfad noch weiter optimiert: Da Short-IPCs wesentlich häufiger als Long-IPCs stattfinden wird zuerst die schnelle, weil nur auf Short-IPC optimierte, Funktion `ipc_short_cut()` aufgerufen. Sollte diese Fehlschlagen, wird in die normale `sys_ipc()`-Funktion verzweigt.

4.3.2 Atomarer Tracepufferzugriff

Nach der Analyse der ersten gesammelten Testdaten zeigten sich verschiedene Probleme mit der Tracepufferimplementation. So enthielten die empfangenen Kerndaten leere oder unvollständige Records, doppelt vergebene Id's (Eintragsnummern) und sich sprunghaft verändernde Zeitstempel.

Diese Probleme lassen sich alle auf die vollständige Unterbrechbarkeit von Fiasco zurückführen: Während der Allokation und Füllung eines Tracepuffereintrags kann ein Interrupt auftreten, und dadurch zu einem anderen Thread umgeschaltet werden. Jedoch waren weder die Pufferallokation (dabei werden auch Zeitstempel, Eintragsnummer sowie andere in jedem Eintrag vorhandene Felder geschrieben) noch die nachfolgende Füllung atomar. Somit traten im Kern die in Abschnitt 3.2.2 beschriebenen Probleme mit nur einem Puffer auf.

Eine Aufteilung in mehrere Puffer ist im Kern nicht praktikabel, also musste der Zugriff auf den Tracepuffer atomar gestaltet werden. Dazu wurde die Tracepufferbehandlung aufgeteilt. Im ersten Schritt erfolgt die Allokation mittels `jdb_tbuf::new_entry()`. Nach erfolgter Füllung des Records wird die Methode `jdb_tbuf::commit_entry()` aufgerufen, um bei Bedarf den virtuellen Interrupt auszulösen. Diese Aufteilung allein garantiert noch keine Ununterbrechbarkeit, jedoch gewährleistet sie, dass der *virq* erst nach vollständiger Füllung ausgelöst wird. Damit während dieser drei Schritte keine Unterbrechungen stattfinden, ist vom Tracepuffernutzer (also beispielsweise dem oben erwähnten Tracingpfad) vor der Allokation ein CPU-Lock zu erwerben und nach dem Commit wieder frei zu geben. Dieses Lock garantiert im Einprozessorfall die Ununterbrechbarkeit[4], denn es sperrt die Interrupts auf dieser CPU.

Im Multiprozessorfall, welcher in dieser Arbeit nicht weiter betrachtet wird, ist die Allokation eines Tracepuffereintrags (in `jdb_tbuf::new_entry()`) gesondert zu sichern, damit auf verschiedenen Prozessoren nicht der selbe Eintrag oder die gleiche Id vergeben werden.

4.3.3 vIRQ-Auslösung

Die Auslösung des *vIRQ* wurde anfänglich genauso implementiert wie die bei normalen Interrupts. Dabei werden für möglichst geringe Latenz die Interrupts sofort zugestellt, wenn der Empfänger bereit ist und eine höhere Priorität als der gerade laufende Thread hat.

Dieses Verhalten bereitet aber beim Tracing von Threadumschaltungen Probleme, da während der Umschaltung zu einem anderen Thread ein Tracerecord erzeugt und somit (bei vollem Tracepuffer) ein *vIRQ* ausgelöst wird. Das führt zu Inkonsistenzen in Fiasco, da der Pfad zur Threadumschaltung atomar durchlaufen werden muss.

[4] Im Kern treten an diesen Stellen weder Seitenfehler noch andere Exceptions auf. Wenn doch, so ist der Kern fehlerhaft und die Ununterbrechbarkeit dadurch nicht mehr gewährleistet.

Damit so etwas nicht passiert wurde die Auslösung bis zum nächsten Timertick verzögert, indem der Empfängerthread lediglich in die Warteschlange der bereiten Threads eingeordnet wird.

4.4 Werkzeuge

Das entwickelte Werkzeug **convtrace** konvertiert die Tracingdaten der **tracelib** ins VAMPIR-Format. Dabei werden auf Anforderung des Nutzers auch die Kerntracingdaten zur Verarbeitung herangezogen.

convtrace ist über Kommandozeilenoptionen und eine Konfigurationsdatei konfigurierbar.

4.4.1 Konvertierung zu VAMPIR

Während der Konvertierung wird jedem L4-Thread, zu welchem es Tracerecords der **tracelib** gibt, eine eindeutige CPU-Id für VAMPIR zugeordnet. Alle Threads (also alle CPUs) eines Prozesses werden zu einem CLUSTER zusammengefasst. So können Selektionen in VAMPIR einfach auf komplette Prozesse angewendet werden.

Aus den im ersten Record eines jeden Threads enthaltenen Introdaten kann **convtrace** das Programm, dessen Daten es gerade verarbeitet, ermitteln und auffinden. Daraus werden die Symbolinformationen (die Namen) der aufgerufenen Funktionen, wie im nächsten Abschnitt beschrieben, ermittelt. Auch können so die CPUs bzw. CLUSTER einem Programm zugeordnet werden.

Die Tracingdaten des Kerns dienen zur Ermittlung und Darstellung von IPC-Nachrichten sowie Threadumschaltungen. IPCs, zu denen es sowohl Sende- als auch Empfangsinformationen gibt, werden auf MESSAGEs abgebildet und in die Ausgabedatei geschrieben. Wenn Sende- oder Empfangsdaten fehlen, dann wird der entsprechende Record mit einer Warnung verworfen. Dieses Verhalten ist unproblematisch, da solche unvollständigen Informationen nur die wenigen IPCs betrifft, welche während des (de-)aktivierens von Kerntracing bereits gestartet sind.

Threadumschaltungen werden ebenfalls auf MESSAGEs abgebildet. Dazu wird aber ein anderer Nachrichtentyp als für IPCs verwendet, so dass diese MESSAGEs in VAMPIR anders dargestellt werden.

4.4.2 Funktionsnamen

Zur Darstellung von Funktionsaufrufen werden die Funktionsnamen benötigt, denn sonst werden nur sehr wenig aussagende Adressen angezeigt. Auch macht ein Gruppieren vom Funktionen anhand der Adresse meist nur wenig Sinn.

Zum extrahieren von Symbolinformationen aus Programmen wird die Bibliothek *libbfd, Binary File Descriptor,* benutzt. Diese Bibliothek stellt plattformunabhängigen Zugriff auf verschiedene Objektformate zur Verfügung. Dabei wird unter anderem auch der Zugriff auf die jeweiligen Symbolinformationen von der *libbfd* gekapselt.

So werden aus den L4-Binaries die Funktionsnamen und -adressen herausgelesen, ohne verschiedene Objekt- oder Ausgabeformate beachten zu müssen.

Anhand der Funktionsnamen können die Funktionen zu Gruppen, ACTION in VAMPIR, zusammengefasst werden.

Kapitel 5

Leistungsbewertung

Alle Messungen wurden auf einem Pentium-133 mit 64 MB RAM vorgenommen. Der Rechner ist ausgestattet mit einer Netzwerkkarte des Typs "DECchip 21142/43", oft auch einfach als "Tulip" bezeichnet.

Als Benchmark wurden 100.000 Runden Pingpong verwendet. Beide Threads des Pingpong-Prozesses arbeiten im selben Adressraum und kommunizieren über Short-IPC. Die angegebenen Zeiten zeigen die durchschnittliche Dauer eines Zyklus, also die Zeit zwischen Senden einer Nachricht und dem Empfang der Antwort darauf.

Zur Zeitmessung wurde der in den Prozessor integrierte *Time-Stamp Counter* (TSC) herangezogen, welcher sich in jedem Prozessortakt um eins erhöht. So ist eine Messung mit hoher Auflösung und Genauigkeit möglich.

5.1 Kerntracing

In Tabelle 5.1 sind IPC-Dauer und -Kosten für verschiedene Tracingarten im Kern aufgeführt. Diese Messungen wurden auf einem minimalen L4-System (nur rmgr, sigma0 und pingpong) durchgeführt.

Es wird ersichtlich, dass reines IPC-Tracing die Kosten (Dauer) einer IPC um knapp 60 Prozent erhöht, das Tracing von Threadumschaltungen um etwa 40 Prozent. Die Kombination von beidem, wie sie auch während des Tracings von Anwendungen von der **tracelib** verwendet wird, verdoppelt also die Kosten einer IPC. Dadurch wird das Zeitverhalten von stark IPC-lastigen Anwendungen relativ stark beeinflusst. Anwendungen mit vielen Berechnungen, aber wenigen Nachrichten werden dadurch nicht sehr in ihrem Verhalten verändert.

Tracingart	JDB-Kommando	μs	Takte	Zusatzkosten
Kein Tracing		6	821	-
IPC Tracing	IT*	9	1308	487
Context Switch Tracing	O0+	8	1149	328
IPC und Context SW	IT*O0+	12	1667	846

Tabelle 5.1: **Tracingkosten im Kern**

Tabelle 5.2 zeigt den Einfluss der Netzwerkübertragung der Tracepuffer des Kerns auf die durchschnittliche Dauer einer IPC. Auf dem L4-System liefen während der Messung folgende Server: rmgr, sigma0, names, log_net, dm_phys und **ktraced**.

Tracingart	JDB-Kommando	μs	Takte
Kein Tracing		6	841
IPC Tracing	IT*	40	5392
Context Switch Tracing	O0+	45	6019
IPC und Context SW	IT*O0+	83	11123

Tabelle 5.2: **Netzwerkeinfluss**

Bei der Interpretation der Tabelle ist jedoch zu beachten, dass die meisten IPCs nur die in Tabelle 5.1 angegebenen Zeiten benötigen. Nur bei wenigen IPCs findet eine Unterbrechung aufgrund eines vollen Kerntracepuffers statt.

Es zeigt sich aber auch, dass die derzeit verwendete Übertragung der Tracingdaten durch die Netzwerkfunktionalität des Logservers sehr viel Zeit in Anspruch nimmt. Dadurch wird der Ablauf des untersuchten Systems erheblich beeinflusst.

5.2 Programmtracing

Beim Tracing von Programmen spielen neben der Verzögerung durch Kerntracing auch die Kosten durch Tracing im Nutzeradressraum eine Rolle. Zur Messung dieser Zeit, sowie der Gesamtzeit für Tracing in Kern und Programm, wurde pingpong instrumentiert.

So ergeben sich die in Tabelle 5.3 aufgeführten Zeiten. In der Spalte "Kerntracing" wird angegeben, ob eine Verbindung zum **ktraced** besteht und Kerntracing aktiv ist. "Datenübertragung" beschreibt, ob sich der Logserver im Netzwerkmodus befindet, oder nicht. In letzterem Fall werden natürlich keine Tracingdaten übertragen, sondern die Puffer nur immer wieder überschrieben. Zur Messung liefen unter L4 die Server rmgr, sigma0, names, log (bzw. log_net), dm_phys und (nur "mit Server") **ktraced**.

Kerntracing	Datenübertragung	μs	Takte
ohne	ohne	11	1584
mit	ohne	24	3304
ohne	mit	27	3693
mit	mit	113	15068

Tabelle 5.3: **pingpong mit Tracing**

Es entstanden während der Messung mit Netz und aktivem Kerntracing rund 38 MB an Tracingdaten, die in knapp 10 Sekunden übertragen wurden. Davon entfallen 32 MB auf die Kerndaten und 6 MB auf Tracingdaten im Nutzeradressraum: Pro pingpong-Zyklus entstehen im Nutzerprogramm vier Tracerecords zu je 16 Byte, pro Thread einer für den Aufruf von `l4_i386_ipc_call()` und einer für das Verlassen dieser Funktion. Im Kern entstehen dabei ebenfalls vier Records, jeder 64 Byte groß: Einmal IPC-Tracing pro Thread, und zwei Kontextwechsel. In Summe ergibt das 25 MB. Die Differenz zu 32 MB ist auf Ereignisse von anderen Threads (vor allem Logserver und **ktraced**) zurückzuführen.

Auch hier zeigt sich, dass die Datenübertragung sehr starken Einfluss auf die Gesamtlaufzeit von pingpong und somit auch auf die durchschnittlichen IPC-Kosten hat.

Kapitel 6

Zusammenfassung

Im Rahmen dieses Belegs wurde ein Tracingsystem für L4/Fiasco entwickelt und implementiert. Die dabei auftretenden Probleme, insbesondere die innerhalb des Fiasco-Kerns, wurden bei der Implementierung berücksichtigt.

Mit diesem Tracingsystem ist es möglich, L4-Anwendungen automatisch zu instrumentieren. So wird eine detaillierte Zeitmessung von Funktionsaufrufen ermöglicht. Ein Programmablauf wird aufgezeichnet und kann zeitversetzt ausgewertet werden. Die dafür notwendigen Tracingdaten von Ereignissen, welche für ein Programm vollkommen transparent sind, beispielsweise Scheduling, werden vom Fiasco-Kern gesammelt.

6.1 Ausblick

Wie in Kapitel 5 dargestellt, ist derzeit nicht die Erzeugung, sondern vor allem die Übertragung der anfallenden Tracingdaten ein Problem, welches gravierenden Einfluss auf das Systemverhalten bei aktivem Tracing hat. Diese Probleme sollten in weiterführenden Arbeiten vorranging gelöst werden. Dafür bieten sich eine Reihe von Ansätzen an:

- Der Einsatz besserer Netzwerktreiber, welche die Fähigkeiten moderner Netzwerkkarten (beispielsweise *DMA* und *Scather-Gather-Lists*) nutzen. Dann lassen sich unnötige Kopieroperationen der Tracingdaten vermeiden. Dadurch erhöht sich die Performance das Gesamtsystem auch mit aktiven Tracing.

- Wünschenswert ist auch ein Streaming der Tracingdaten zum Netzwerktreiber, indem auch die Puffer der **tracelib** aufgeteilt werden. So können die Applikationen weiter laufen, während Teile ihrer Tracepuffer übertragen werden. Dabei könnte sich die Übertragung adaptiv an das Datenaufkommen anpassen, indem Puffer von Threads bevorzugt werden, deren nächster Puffer ebenfalls schon zur Übertragung ansteht.

- Zur Einschränkung der erzeugten Tracingdatenmenge bei großen Applikationen sollten unnütze Tracerecords bereits während der Erzeugung im System ausgefiltert werden. Dies entlastet Übertragung und sogt auch für mehr Übersichtlichkeit während der Darstellung in VAMPIR.

- Ein anderer Ansatz zur Reduktion des Datenvolumens ist die Verkleinerung der Tracerecords. Dabei ist zu überprüfen, wo kleinere Tracerecords möglich und sinnvoll sind.
- Die Tracepufferimplementation des FIASCO-Kerns sollte auf Multiprozessortauglichkeit untersucht und gegebenenfalls entsprechend angepasst werden.
- Eine bessere Steuerung der automatischen Instrumentierung des *gcc*-Compilers könnte zu besseren Ergebnissen führen, indem vom Nutzer bestimmbare Funktionen nicht instrumentiert werden. Dabei wäre auch eine Option nützlich, mit der das Instrumentieren von Inline-Funktionen und Templates (bei C++) verhindert wird.

Literaturverzeichnis

[1] Professur Betriebssysteme. http://os.inf.tu-dresden.de/

[2] Fiasco Webseiten. http://os.inf.tu-dresden.de/fiasco/

[3] Das Drops Projekt. http://os.inf.tu-dresden.de/drops/

[4] The L4 Common Environment, Dokumentation, http://os.inf.tu-dresden.de/L4/bib.html#l4env

[5] Jan Glauber, Kerndebugger für den Mikrokern Fiasco, Großer Beleg, TU-Dresden, 2001.

[6] Jan Glauber, Frank Mehnert und Jochen Liedtke, FIASCO Kernel Debugger Manual, TU-Dresden.

[7] Pallas GmbH, Brühl. http://www.pallas.de/

[8] VAMPIR, Visualization and Analysis of MPI Programs, Pallas GmbH. http://www.pallas.de/pages/vampir.htm

[9] VAMPIR User's Guide, Pallas GmbH. http://www.pallas.de/pages/vampir.htm

[10] Vampirtrace MPI Profiling Library, Pallas GmbH. http://www.pallas.de/pages/vampirt.htm

[11] Vampirtrace Users Guide, Pallas GmbH. http://www.pallas.de/pages/vampirt.htm

[12] VAMPIR und Vampirtrace am Zentrum für Hochleistungsrechnen, TU-Dresden. http://www.tu-dresden.de/zhr/Service/Tools/Vampir/vampir.html

[13] Karim Yaghmour and Michel R. Dagenais, Measuring and Characterizing System Behavior Using Kernel-Level Event Logging, In *Proceedings of the 2000 USENIX Annual Technical Conference.*

[14] Linux Tracing Toolkit. http://www.opersys.com/LTT/

[15] *gcc* Manual, GNU-Info-Seiten des *gcc*