Portierung von Bastei auf L4ka::Pistachio

Julian Stecklina

18. Juli 2008

Großer Beleg

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Stefan Kalkowski

Zusammenfassung

Es wird eine vollständige Portierung des Betriebssystems Bastei, das bisher auf dem Fiasco-Mikrokern und auf Linux läuft, auf *L4ka::Pistachio* vorgestellt. Bastei wurde zusätzlich um Primitive für Multiprozessorsysteme erweitert.

Anhand von Benchmarks der IPC-Primitive von Bastei werden potentielle Unterschiede zwischen den Implementationen beleuchtet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

In einer Welt (oder besser einem *Mindshare*), in der es Überzeugungsarbeit benötigte, um C++ als Systemprogrammiersprache zu benutzen, ist Bastei schon eine seltsame Erscheinung: Es nutzt weder einen hoch-optimierenden IDL-Compiler, noch macht es einen Bogen um die sonst im Mach-geschundenen und Performance-orienterten Mikrokernumfeld verpönten C++-Features wie virtuellen Methoden und Templates.

Mit einfachen Methoden entsteht so aus wenig Code ein System, das trotzdem alle Vorteile eines Mikrokern-Betriebssystems bietet, dessen Systemdienste und Trieber aber größtenteils unabhängig vom konkret verwendeten Mikrokern implementiert sind. Zumindestens ist das die Theorie, da Bastei bis jetzt nur auf dem Fiasco-Mikrokern funktionierte.

Um diese Lücke zu schließen, befasst sich diese Arbeit mit der Portierung von Bastei auf den Pistachio-Mikrokern.

Das Portieren von Bastei auf den Pistachio-Kern hat neben der offensichtlichen Vermehrung der Plattformen, auf denen Bastei lauffähig ist, einige andere positive Eigenschaften:

- Es fehlt an einer gemeinsamen Anwendung, die sowohl auf Fiasco als auf Pistachio läuft und als Vergleichsbasis der beiden Kerne dienen kann. Bastei kann diese Lücke schließen und Vor- und Nachteile des hauseigenen Fiasco-Kerns im Vergleich zu Pistachio aufzeigen. Auf diesen Aspekt wird im Abschnitt 4 näher eingegangen.
- Aktuelle Hardware tendiert immer mehr zur Benutzung von Multicore-CPUs. Fiasco fehlt noch SMP-Unterstützung. Solange dieses Feature nicht vorhanden ist, kann eine Evaluation auf Pistachio stattfinden.
- Die Portierbarkeit von Bastei kann untersucht werden. Dies wird in Abschnitt 3.10 versucht.

2 Der Stand der Technik

2.1 Die L4-Mikrokernfamilie

In der Mitte der 1980er Jahre versuchte das Mach-Projekt[ABB+86] den monolithischen Kern von UNIX (4.3BSD) in einen Mikrokern und zugehörige Userlevel-Tasks aufzuteilen, um die Wartbarkeit des damals schnell wachsenden UNIX-System durch Modularisierung des Kernels zu verbessern. Der dabei entstandene Mach-Kern war der erste weithin bekannte *Mikrokernel*, enttäuschte aber auf lange Sicht durch vergleichsweise schlechte Performance[Lie96b]. Der Kern von Berkeley UNIX ist bis zu seinen heutigen Inkarnationen in den Open-Source BSD-Systemen monolithisch geblieben und von Mach bleibt bis heute im Großen und Ganzen nur das Virtual-Memory-System[MNN04].

Trotz dieser scheinbaren Sackgasse entwickelte sich ausgehend vom L4/x86-Mikrokern[Lie95] eine zweite Generation von Mikrokerneln, da die Vorteile von Mikrokern-basierten Systemen überwogen. Darunter sind u.a.

- eine modularere und flexiblere Systemstruktur, in dem Systemkomponenten zur Laufzeit ausgetauscht werden können[1],
- Fehlerisolierung durch Implementierung von Systemdiensten und Treibern in voneinander getrennten Adressräumen,
- Minimierung der Trusted Computing Base,
- die Möglichkeit verschiedene OS Personalities zu implementieren.

Verschiedene Forschungsinteressen und Anwendungsszenarien haben dazu geführt, dass sich die L4-Idee in unterschiedliche Richtungen entwickelt hat, so dass heute mehrere L4-Mikrokernel mit meistens inkompatiblen Kernel APIs existieren (Abbildung 1).

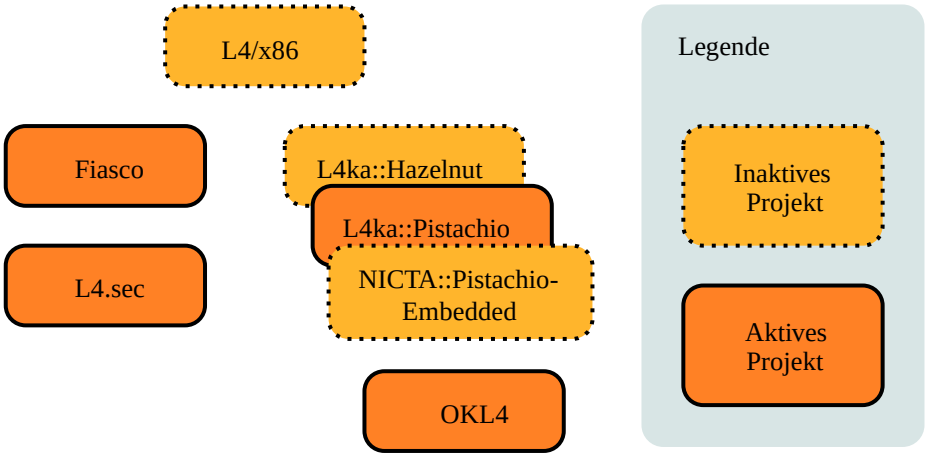

Abbildung 1: Die L4-Mikrokernfamilie (Auswahl)

Ohne ausschweifend die bewegte Geschichte der Mikrokerne[Lie96b] zu beleuchten, möchte ich kurz auf die beiden L4-Mikrokerne eingehen, die in dieser Arbeit im Vordergrund stehen.

2.1.1 Fiasco

Fiasco ist ein echtzeitfähiger Mikrokern, der am hiesigen Lehrstuhl im Rahmen des DROPS-Projekts (*Dresden Real-Time Operating System*) entwickelt wird. Er implementiert die L4v2-Spezifikation[Lie96a], wurde aber im Laufe der Jahre mit einigen Features außerhalb der Spezifikation erweitert.

2.1.2 L4ka::Pistachio

L4ka::Pistachio ist eine L4-Implementation der System Architecture Group der Universität Karlsruhe in Zusammenarbeit mit der DiSy Group der University of New South Wales.

[1] Der originale Mach-Kern hatte eine Implementation von Common Lisp. Ich stelle mir die Kombination einer echten höheren Programmiersprache und der Flexibilität von Mikrokern-basierten Systemen sehr produktiv vor, da durch die Verwendung einer Sprache wie Lisp, ein laufender Task z.B. zum Einspielen von Updates oder Bug-Fixes in den meisten Fällen nicht angehalten werden muss.

Ausgehend von der Pistachio-embedded-Codebasis[hin08], eine Spezialisierung des Pistachio-Kerns für eingebettete Systeme, wurde der kommerzielle OKL4-Kern entwickelt, der als Basis für kommerzielle (Mikrokern-)Produkte vermarktet wird[okl].

2.2 Pistachios Kern-API

Pistachio implementiert die L4 API Version 4 (L4v4), die, solange sie nicht finalisiert ist, auch als eXperimental Version 2 (X2) geläufig ist. Zum Zeitpunkt des Schreibens scheint Pistachio die *einzige* Implementation dieser API zu sein. Die größeren und für diese Arbeit relevanten Änderungen im Vergleich zur L4v2 API sind:

2.2.1 Trennung von API und ABI

Die L4v4-Spezifikation trennt das an C- und C++-Programme exportierte Programmierinterface (API) von der Implementation auf einer konkreten Hardware-Plattform. Code, der direkt mit der L4v4-API interagiert, ist nur in geringem Maße an eine bestimmte Plattform gebunden. Eine dünne Schicht an C/C++-Code bildet diese Abstraktion.

Am Beispiel von IPC wird diese Trennung deutlich. Zum Senden und Empfangen einer Nachricht stehen 64 virtuelle *message register* ($\texttt{MR}_{0-63}$) zur Verfügung, die vom Programmierer via `L4_StoreMR` bzw. `L4_LoadMR` manipuliert werden können. Ob diese virtuellen Register auf Speicher oder Hardware-Register abgebildet werden, ist für den Programmierer transparent. Im Falle der IA-32-Architektur wird beim Senden nur $\texttt{MR}_0$ im Hardware-Register `ESI` übergeben, beim Empfangen einer Nachricht zusätzlich $\texttt{MR}_1$ und $\texttt{MR}_2$ in den Registern `EBX` und `EBP`. Die restlichen `MRs` werden im UTCB (siehe Abschnitt 2.2.2) abgelegt.

Listing 1: Das Senden eines Puffers mittels der L4v4-API

```
L4_Msg_t msg;
L4_StringItem_t sitem = L4_StringItem(size, buffer);

// Prepare message
L4_Clear(&msg);
L4_Append(&msg, sitem);
L4_Load(&msg);

// Send the prepared message
L4_MsgTag_t result = L4_Send(dst_tid);

// Error handling
if (L4_IpcFailed(result)) {
  // ...
}
```

Die API unterscheidet sich dadurch in ihrem Erscheinungsbild deutlich von der L4v2-API. Der im Pistachio-Whitepaper zitierte Vorteil von „portabler Systemsoftware" hat sich in der Praxis nicht bemerkbar gemacht. Das Großteil von unportablem Code in Bastei, wobei unportabel sich auf die CPU-Architektur und nicht auf den benutzten Mikrokern bezieht, besteht aus:

- Assembler-Code, der benötigt wird, um überhaupt C/C++-Code ausführen zu können,
- Inline-Assembler, z.B. zur Implementation von Locks und
- Makefiles und Linker-Scripts.

Dieser Code benutzt die Mikrokern-API nicht und kann somit auch nicht von deren Portierbarkeit profitieren.

2.2.2 User-level thread control blocks (UTCBs)

Jedem Thread ist ein Speicherbereich in seinem Adressraum, der *UTCB*, zugeordnet, der vom Thread sichtbare und teilweise veränderbare Informationen über sich selbst hält.

Der UTCB beinhaltet alle virtuellen Register, für die die jeweilige ABI keinen Platz in Hardware-Registern vorsieht:

TCRs die *Thread Control Registers*. Durch sie kann ein Thread u.a. seinen Pager und seinen Scheduler (siehe Abschnitt 2.2.4) setzen.

MRs die *Message Registers*.

BRs die *Buffer Registers*. Sie beschreiben Speicherbereiche, die für String-IPC bereitstehen.

2.2.3 Trennung von Adressräumen und Thread-IDs

Im Gegensatz zur L4v2 API beinhalten Thread-IDs in Pistachios API *nicht* explizit einen Identifier des Tasks, zu dem sie gehören.

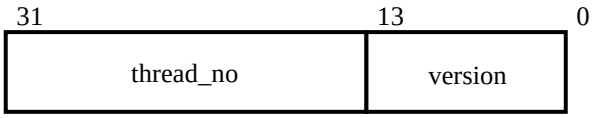

Abbildung 2: Die Struktur einer globalen Thread-ID in der L4v4 ABI für 32-Bit-Systeme

Tasks bzw. Adressräume werden durch eine Thread-ID eines Threads benannt, der zu diesem Adressraum gehört. Wird ein Thread mit seiner eigenen ID als Adressraum erzeugt (mittels `L4_ThreadControl`), entsteht ein neuer, leerer Adressraum. Mit dem Zerstören aller Threads in einem Adressraum wird implizit auch der Adressraum selbst zerstört.

Nachdem Threads mit `L4_ThreadControl` erzeugt wurden, können sie auf zwei unterschiedliche Weisen gestartet werden:

1. Mit `L4_ExchangeRegisters` kann ein Stack- und Instruktionspointer gesetzt werden. Dieser Systemcall ist nur im selben Adressraum effektiv.

2. Der Pager des neu erstellten Threads schickt diesem eine Nachricht, die aus Stackpointer und Instruktionspointer besteht.

Threads werden zerstört, wenn sie mit `L4_ThreadControl` in den Adressraum des *nilthread* verschoben werden.

2.2.4 Scheduling und Multiprozessorunterstützung

Das Scheduling wird wie auch schon in der L4v2 API vom Mikrokern vorgenommen und ist mittels des Systemcalls `L4_Schedule` beeinflussbar.

Listing 2: Der Systemcall `L4_Schedule`

```
L4_Word_t  L4_Schedule (L4_ThreadId_t dest,
                        L4_Word_t TimeControl,
                        L4_Word_t ProcessorControl,
                        L4_Word_t prio,
                        L4_Word_t PreemptionControl,
                        L4_Word_t * old_TimeControl);
```

`L4_Schedule` erwartet fünf Werte (`dest`, `ProcessorControl`, `Prio` und `PreemptionControl`) und gibt den alten Wert von `TimeControl` und einen Statuscode im Rückgabewert zurück.

Mit dem Parameter `dest` wird der Thread ausgewählt, dessen Scheduling-Parameter geändert werden sollen. Der diesen Systemcall aufrufende Thread muss sich im gleichen Adressraum befinden, wie der Scheduler von `dest` (siehe Abschnitt 2.2.2). Anderenfalls hat der Aufruf keinen Effekt.

Mit `ProcessorControl` kann ein Thread auf eine andere CPU migriert werden. CPUs werden von 0 beginnend durchnummeriert. Wird ein Thread nicht explizit durch `L4_Schedule` migriert, wird er auf CPU 0 erzeugt und ausgeführt.

Jedem Thread ist eine Priorität zugeordnet, die via `prio` verändert werden kann. Höher priorisierte Threads werden bevorzugt vom Mikrokern ausgeführt. Der Roottask und σ_0 besitzen die höchste Priorität (255) und werden nie von Threads mit geringerer Priorität preemptiert.

2.2.5 Hardware-Interrupts (IRQs)

Hardware-Interrupts werden auf Threads und IPC abgebildet. Jedem IRQ ist eine Thread-ID zugeordnet. Andere Threads können sich mit diesem Interrupt *assoziieren*, indem sie sich als Pager des Interrupt-Threads setzen. Ab diesem Zeitpunkt empfangen sie eine leere Nachricht vom Interrupt-Thread, wenn der Interrupt ausgelöst wird. Die Zustellung von Interrupt-Nachrichten ist danach solange maskiert, bis der assoziierte Thread seinerseits eine leere Nachricht an den Interrupt-Thread schickt.

Interrupt-Threads werden nach Bedarf vom Mikrokern erzeugt.

2.3 Bastei

Bastei ist der Prototyp eines Betriebssystem-Konzepts, dass mit dem Fokus auf Sicherheit und ein Mindestmaß an Komplexität implementiert ist[FH06].

Bastei bietet Mittel, die *Trusted Computing Base* (TCB) jeder einzelnen Anwendung auf sie zuzuschneiden und somit zu minimieren. Erreicht wird das durch eine

hierarchische Prozessstruktur, wie sie exemplarisch in Abbildung 3 skizziert ist, in der Elternprozesse Dienste für ihre Kindprozesse verwalten. Prozessen kann somit genau der Satz an Diensten zugänglich gemacht werden, der für ihre Funktion notwendig ist.

Momentan läuft Bastei auf dem Fiasco-Mikrokern und zu Debug-Zwecken auch auf GNU/Linux.

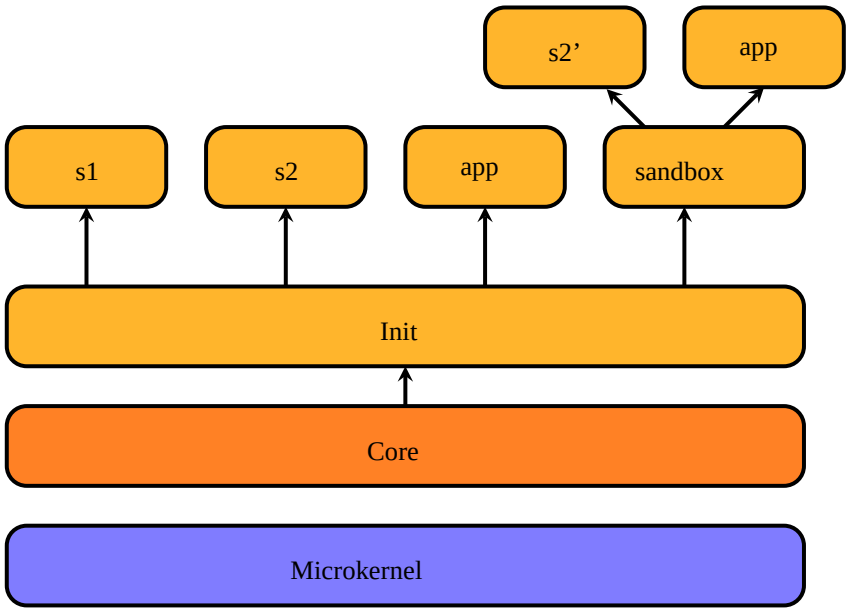

Abbildung 3: Die Struktur eines Bastei-Systems

2.3.1 Repository-Struktur

Der Bastei-Quellcode wird in mehreren Quellcode-Repositories verwaltet. Dies soll das parallele Entwickeln von Komponenten von verschiedenen Entwicklern ermöglichen.

Momentan existieren die folgenden Repositories:

base/ Dieses Repository enthält die grundlegenden Interfaces des Bastei-Systems, den generischen Teil von Core (siehe Abschnitt 2.3.2) und einige wichtige Bibliotheken:

allocator_avl Ein auf AVL-Bäumen basierenden Speicherverwalter,

lock eine Implementation von Mutexen,

thread die Thread-Verwaltung,

elf ein Parser für ELF-Binaries,

env die Schnittstelle zum Eltern-Prozess.

base-fiasco/ Die Portierung auf den Fiasco-Mikrokern befindet sich in diesem Repository. Diese besteht aus dem plattform-spezifischen Teil von Core und den folgenden Bibliotheken:

console dem Fiasco-spezifische Konsolentreiber,

ipc der Abbildung von Basteis IPC und Paging auf Fiascos IPC-Primitive,

task der Bibliothek zum Erzeugen von Tasks,

cxx der nötigen Runtime-Bibliothek für C++-Code.

Weiterhin beinhaltet dieses Repository auch noch Teile des Server-Frameworks.

os/ Hier befinden sich Systemdienste, u.a.:

- Init (siehe Abschnitt 2.3.3),
- Treiber für den VESA-Framebuffer und PS/2-Tastaturen und -Mäuse,
- der *Nitpicker* Window-Manager.

demo/ Das Demo-Szenario ist in diesem Repository implementiert. Dazu gehören u.a.:

- das *Launchpad*,
- der Tutorial-Browser *Scout*,
- einige Demo-Programme.

2.3.2 Core

An der Spitze der Prozesshierarchie steht *Core*, dessen primäre Funktion es ist, eine Plattformabstraktionsschicht zu bilden.

Core besteht aus mehreren Modulen, die sich größtenteils in einen generischen und einen plattformabhängigen Teil aufteilen, von denen sich der generische im `base/` -Repository und der plattformabhängige im jeweiligen Repository für die spezielle Plattform befindet.

Die Dienste (siehe Abschnitt 2.3.4), die Core implementiert, sind:

`cpu_session` Sie kapselt Operationen auf Threads. In Core ist der plattform-spezifische Teil in der Klasse `Platform_thread` gekapselt.

`rm_session` Ein Task hat eine zugehörige `rm_session` (*region manager*), die seinen Adressraum verwaltet. Mittels dieses Dienstes können Dataspaces als Speicherbereiche in einen Adressraum eingeblendet werden. (Siehe Abschnitt 2.3.7)

`ram_session` Eine `ram_session` bildet einen Pool von Speicher mit einer festgelegten Quota, der Maximalgröße dieses Pools. Zwischen `ram_sessions` kann Quota transferiert werden. (Siehe Abschnitt 2.3.8)

`rom_session` Über diesen Dienst können die per Bootloader erzeugten Module als Dataspaces angesprochen werden.

`task_session` Der Task-Dienst hat nur eine Methode `bind_thread` , mit der ein Thread an einen Task gebunden werden kann. Der plattform-spezifische Teil dieses Dienstes ist in der Klasse `Platform_task` implementiert.

`irq_session` Dieser Dienst abstrahiert Hardware-Interrupts. (Siehe Abschnitt 2.3.9)

`io_port_session` Port I/O kann auf der IA-32-Plattform über diesen Dienst realisiert werden.

`io_mem_session` Mittels dieses Dienstes kann I/O-Speicher als Dataspace angefordert werden. (Siehe Abschnitt 2.3.10)

Nachdem diese Dienste initialisiert sind, besteht Cores einzige Aufgabe nur noch daraus *Init* zu starten, dem neu erzeugten Task den vorhandenen Speicher zu transferieren und auf Anfragen auf die Dienste zu antworten.

Durch die zentrale Rolle von Core beschränkt sich der Portierungsaufwand auch fast vollständig darauf, Core zu portieren. Wie das im einzelnen geschehen ist, ist in Abschnitt 3 beschrieben.

2.3.3 Init

Init ist der einzige Kindprozess von Core. Er erstellt anhand einer Konfigurationsdatei (Listing 3) weitere Prozesse, die dann auf die von Core angebotenen Dienste zugreifen können und natürlich eigene Dienste anbieten können.

Der Quellcode von Init ist vollständig plattformunabhängig.

Listing 3: Die Konfigurationsdatei für *Init*

```
<!-- -*- Mode: Xml -*- -->
<config>
        <start>
                <filename>ps2_drv</filename>
                <ram_quota>1M</ram_quota>
        </start>
        <start>
                <filename>timer</filename>
                <ram_quota>1M</ram_quota>
        </start>
        <start>
                <filename>nitpicker</filename>
                <ram_quota>1M</ram_quota>
        </start>
        <start>
                <filename>vesa_drv</filename>
                <ram_quota>1M</ram_quota>
        </start>
        <start>
                <filename>launchpad</filename>
                <ram_quota>64M</ram_quota>
        </start>
</config>
```

2.3.4 Dienste und Sitzungen

Wie schon in den einleitenden Worten dieses Abschnitts erwähnt, ist der Kern von Bastei ein hierarchisches Prozesskonzept, in dem *Dienste* eine wichtige Rolle spielen.

Jeder Task kann einen oder mehrere Dienste anbieten. Damit andere Tasks diese Dienste benutzen können, werden diese Dienste mit Namen beim jeweiligen Elternprozess angemeldet.

Will eine Anwendung einen Dienst in Anspruch nehmen, muss sie zuerst eine *Sitzung* zu diesem Dienst aufbauen. Dieser Vorgang wird vom Elternprozess verwaltet. Am Beispiel des Timer-Dienstes ist dies in Listing 4 dargestellt.

Listing 4: Erstellen einer Sitzung zum Timer-Dienst

```
Timer::Session_client timer(session("Timer",
                                    "ram_quota=8K"));

while (1) {
  do_something();
  timer.msleep(20);
}
```

Die Funktion `session()`, die als Parameter den Dienstnamen und die Dienstparameter bekommt, gibt eine Capability für diesen Dienst zurück. Diese Capability wird vom Dienst erzeugt und durch den Elternprozess des aufrufenden Tasks weitergeleitet, welcher auch die volle Kontrolle hat,

- die Dienstparameter zu verändern,
- eines seiner Kinder, welches diesen Dienst implementiert, auszuwählen,
- die Anfrage an den nächsten Elternprozess weiterzugeben,
- diesen Dienst selbst bereitzustellen.

Nach dem Aufbau einer Sitzung funktioniert die Kommunikation zwischen Client und Server direkt, d.h. der Elternprozess ist darin nicht mehr involviert. Im Beispiel ist das der Aufruf der Methode `timer::msleep()`.

2.3.5 IPC

Dienste basieren auf Basteis IPC-Framework, in dem *Capabilities* als Namen für Kommunikationsbeziehungen benutzt werden. Im Fiasco-Port bestehen Capabilities aus einer Thread-ID und einem lokalen Namen[2], der ein bestimmtes Objekt benennt (im Weiteren *Objekt-ID* genannt).

Ein normaler Aufruf einer Funktion einer Dienstsitzung läuft folgendermaßen ab:

1. Der Client verpackt den Opcode der aufzurufenden Funktion und die Argumente in einem Nachrichtenpuffer.
2. Der Puffer wird zusammen mit der Objekt-ID via String-IPC Call des L4-Kerns an die Thread-ID, die in der Capability vermerkt ist, geschickt.
3. Beim Adressaten wird anhand der Objekt-ID das zugehörige Server-Objekt gesucht und dessen `dispatch` -Methode aufgerufen, die die Nachrichtenparameter aus dem Puffer extrahiert.

[2] In der aktuellen Implementation sind lokale Namen mangels Kernel-Features global. Capabilities können demnach mittels Brute-Force „erraten" werden.

4. Anhand des Opcodes wird die gewünschte Methode aufgerufen, die einen oder mehrere Rückgabewerte produziert. Diese werden wieder in einem Nachrichtenpuffer abgelegt.

5. Der Nachrichtenpuffer wird via String-IPC an den Client geschickt.

6. Der Client kann nun den Rückgabewert aus dem Nachrichtenpuffer holen.

2.3.6 Tasks und Threads

Zu jedem Task und jedem Thread in Bastei existiert ein `Platform_task` bzw. `Platform_thread` -Objekt in Core, das die spezielle Implementation auf der Zielplattform kapselt. In der Fiasco-Implementation hat die Klasse `Platform_task` ein statisches Array `_tasks` von Zeigern auf `Platform_task` Objekte. Soll nun ein Task erzeugt werden, wird in diesem Array nach einem unbenutzten Eintrag gesucht und dessen Index als Fiasco-Task-Nummer verwendet. Mit dem Systemcall `l4_task_new` wird ein Task mit dieser Task-Nummer erzeugt.

Analog zum Erzeugen von Tasks verläuft das Erzeugen von Threads. Jeder Task hat ein Array von `Platform_thread` -Objekten. In diesem wird ein freier Eintrag gesucht und dessen Index als Thread-Nummer verwendet. Task- und Thread-Nummer ergeben zusammen die Thread-ID eines Threads. Gestartet wird dieser Thread, indem er via `l4_inter_task_ex_regs` einen initialen Instruktions- und Stackpointer gesetzt bekommt. Gestoppt wird ein Thread, indem er mit dieser Methode gezwungen wird, mit einem Page Fault auf einer nicht-existierenden Seite zu blockieren[3].

2.3.7 Adressräume und Dataspaces

Adressräume werden durch eine Sitzung zum *Region Manager* verwaltet. Jeder Task hat genau eine solche Sitzung, um seinen eigenen Adressraum zu verwalten.

Shared Memory zwischen verschieden Tasks wird mittels *Dataspaces* erzeugt. Dataspaces werden durch entsprechende Capabilities angesprochen und können durch den Region Manager in einen Adressraum eingeblendet werden. Ein Anwendungsfall ist der Framebuffer des VESA-Treibers.

2.3.8 RAM-Quota

Jeder Task in Bastei bekommt bei seiner Erzeugung eine Sitzung zum RAM-Dienst, mit der er Dataspaces allozieren kann. Die Menge des Speichers, die durch diese Sitzung angefordert werden kann, hat eine Grenze: die RAM-Quota.

Ein spezieller Dienstparameter ist `ram_quota`. Mit ihm kann der Client dem Server, der diesen Dienst anbietet, einen Teil seiner RAM-Quota abgeben, um Zustandsinformation o.ä. zu speichern. Dienste sollten so implementiert sein, dass sie für jede Sitzung nur den dafür donierten Speicher anstatt ihres eigenen benutzen, um *Denial-of-Service*-Angriffe durch eine große Anzahl offener Sitzungen zu verhindern.

[3] Um den Kommentar im Quellcode zu zitieren:

> The Fiasco thread is halted by setting itself as pager and forcing pagefault at 0, where Bastei never maps a page. The bottom line is the thread blocks in IPC to itself.

2.3.9 Hardware-Interrupts

Hardware-Interrupts können über den Dienst `irq_session` angesprochen werden. Die Benutzung dieses Dienstes ist im Listing 5 dargestellt. In diesem Beispiel wird eine Session zum IRQ-Dienst etabliert, um sich mit dem IRQ 0, dem Timer-Interrupt, zu assoziieren. Danach wird nach jedem Interrupt eine kurze Nachricht ausgegeben.

Listing 5: Ein Beispiel der Benutzung von Basteis IRQ-Abstraktion

```
Irq_session_client irq0(session("IRQ",
                                "ram_quota=4K, irq_number=0"));

while (1) {
  irq0.wait_for_irq();
  printf("Got interrupt.\n");
}
```

Interrupt-Handling wird in Core durch durch zwei plattform-spezifische Funktionen abstrahiert:

- `_associate_to_irq`
- `_wait`

`_associate_to_irq` wird aufgerufen, wenn eine Sitzung zum IRQ-Dienst aufgebaut wird. `_wait` wird von der Dienstmethode `_wait_for_irq` aufgerufen.

2.3.10 Gerätespeicherverwaltung

Bastei macht einen Unterschied zwischen normalem physischem Speicher und Speicher, der zur Kommunikation mit diversen Geräten benutzt wird. Diese Unterscheidung ist auf Fiasco notwendig, da Gerätespeicher über ein spezielles Protokoll von σ_0 angefordert werden muss.

2.3.11 Locks

Bastei benutzt zur Synchronisierung von Threads Locks. Locks werden durch eine Integer-Variable implementiert. Zum Erwerben eines Locks wird mittels `cmpxchg` versucht, die Variable von `UNLOCKED` auf `LOCKED` zu setzen. Schlägt das fehl, da die Variable schon den Wert `LOCKED` hatte, legt sich der Thread für eine kurze Zeit schlafen.

3 Die Portierung

3.1 Erstellen eines neuen Repositories

Um dem Repository-Layout (siehe Abschnitt 2.3.1) von Bastei gerecht zu werden, erstellte ich ein neues Repository *base-pistachio/* mit der gleichen Struktur wie *base-fiasco/*. Da ich erwartete einen signifikanten Teil des Codes in *base-fiasco/* wiederverwenden zu können, erstellte ich zusätzlich noch ein zweites Repository *base-l4/*, das gemeinsamen Code der beiden Portierungen halten sollte.

Der erste Meilenstein war, ein leeres Programm zum Linken zu bekommen. Die *cxx*-Bibliothek und das Linkerscript vom Fiasco-Port konnten dafür größtenteils wiederverwendet werden.

3.2 Locks

Locks wurden auf Pistachio äquivalent zu ihrer Fiasco-Version implementiert.

3.3 IPC und Paging

Als nächsten Schritt portierte ich das IPC-Framework. Da IPC bis auf das Aussehen der API auf beiden Kernen sehr ähnlich ist, blieb die Struktur des Fiasco-IPC-Codes erhalten und ich musste nur die entsprechenden Syscalls und Typnamen ersetzen.

Ähnlich verlief es mit dem auf IPC abgebildeten Paging-Framework von Bastei. Auch hier mussten nur die konkreten Syscalls durch ihre Pistachio-Pendante ausgetauscht werden.

3.4 Hardware-Interrupts

Die Portierung des IRQ-Dienstes erachtete ich als trivial, da Pistachio zu den IRQ-Primitiven in Fiasco scheinbar äquivalente Primitive hatte. Im Zusammenspiel mit Hardware-Interrupts gab es dennoch Schwierigkeiten.

Interrupts werden in der L4v4 API als IPC von einem vom Kern erzeugten Interrupt-Thread zugestellt (Abschnitt 2.2.5). Die IRQ-Abstraktion in Pistachio deckt sich mit der in Fiasco bis auf einen entscheidenden Punkt: Nachdem sich ein Thread mit einem Interrupt assoziiert hat, ist die Zustellung von Interrupt-Nachrichten bei Fiasco maskiert, auf Pistachio ist das *nicht* der Fall.

Da es in der L4v4-Spezifikation keine Möglichkeit gibt, Interrupts zu maskieren, ohne auf einen Interrupt zu warten, lässt sich das Verhalten des Fiasco-Ports nicht exakt reproduzieren und es ergeben sich Probleme, da Threads in Core, die sich mit einem Interrupt assoziieren, gleichzeitig noch einen Server-Loop ausführen (siehe Abschnitt 2.3.4).

Während der Thread in Core, der die Sitzung zum IRQ-Dienst implementiert, nach dem Aufruf von `associate_to_irq` mit einem offenen `L4_Wait` auf Anfragen von Clients wartet, bleibt der betreffende Interrupt demaskiert. Dies ist problematisch, da ein an dieser Stelle auftretender Interrupt entgegengenommen und, da er sich in der Form von normaler Bastei-IPC unterscheidet, verworfen wird. Tritt dieser Fall ein, geht dieser Interrupt verloren und die Interruptzustellung ist maskiert.

Soll folgend mit der Dienstmethode `wait` auf das Auftreten des Interrupts gewartet werden, ist unklar, ob der Interrupt maskiert ist. Das unkonditionelle Demaskieren scheitert an dieser Stelle, da Pistachio einen Thread blockieren lässt, der versucht einen nicht maskierten Interrupt zu demaskieren.

Mein erster Ansatz bestand darin, in der `wait` -Methode des IRQ-Dienstes jedes mal erneut zu assoziieren und nach dem Warten auf den Interrupt sofort wieder zu deassoziieren. Dadurch ergab sich aber ein Zeitfenster zwischen den `wait` -Aufrufen, in denen das Auftreten des Interrupts nicht überprüft wurde und somit verloren ging.

Um dieses „Interrupt-Loch" zu schließen, werden in `_wait` folgende Informationen benötigt:

1. Ist der Interrupt maskiert, d.h. muss er mit einer leeren Nachricht an den Interrupt-Thread demaskiert werden?

2. Wurde eine Interrupt-Nachricht in Basteis IPC-Code in diesem Thread empfangen?

Die IPC-Bibliothek und die beiden plattform-spezifischen Methoden im IRQ-Dienst wurden dahingehend erweitert, diese Zustande zu notieren und entsprechend zu handeln. Dazu wird das „User defined"-Wort im UTCB benutzt, das frei verwendbar ist. Der Aufwand für diese Buchhaltung ist im Vergleich zum restlichen IPC-Pfad vernachlässigbar.

3.5 Verwaltung von Tasks und Threads

Während Fiasco explizit einen Syscall `l4_task_new` zum Erstellen von Adressräumen anbietet (siehe Abschnitt 2.3.6) und jede Thread-ID explizit die Nummer ihres Adressraums beinhaltet, gibt es in Pistachios API keine expliziten Task-IDs (siehe Abschnitt 2.2.3).

Mein initialer Ansatz war deshalb *kein* statisches Array zur Vergabe von IDs zu verwenden und Thread-IDs, die sowohl Adressräume wie auch Threads benennen können, aus einer globalen Free-List zu schöpfen.

Soll ein Adressraum erzeugt werden, allozierte diese erste Implementation eine Thread-ID von der Free-List. Mit dieser ID wurde ein inaktiver Thread gestartet, dessen einziger Zweck es ist, diesen neuen Adressraum zu benennen. Soll der Task zerstört werden, genügte es alle Threads in ihm einschließlich dieses inaktiven Threads zu zerstören.

Diese Implementation hatte zwei Probleme:

1. Cores Speicherverbrauch wächst mit der Anzahl gestartetet Tasks und Threads. Da Core fast seinen kompletten Speicher an Init doniert, ist es möglich, dass an einem bestimmten Punkt Core keinen Speicher mehr für diese Buchhaltung von IDs hat.

2. Diese Vergabe von IDs hat einen Bug beim Schließen von Sitzungen provoziert. Siehe Abschnitt 4.1.

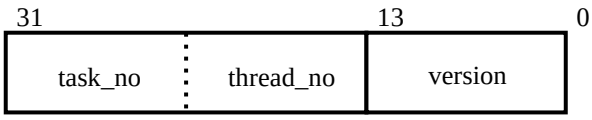

Abbildung 4: Bereitstellung von getrennten Task- und Thread-IDs in der L4v4 ABI für 32-Bit-Systeme

In einem zweiten Ansatz versuchte ich die Vergabestrategie der bestehenden Fiasco-Portierung genau nachzuahmen: Dazu wird die Threadnummer in Pistachios Thread-IDs (siehe Abschnitt 2.2.3) wie in Abbildung 4 dargestellt benutzt, um seperate Thread- und Task-IDs zu simulieren. Die Größe der beiden Felder ist frei wählbar. In der aktuellen Implementation werden je 9 Bit verwendet.

Unterschiede zur Implementierung auf Fiasco bestehen darin, dass der Thread 0 eines Tasks automatisch bei der Erstellung des Tasks erzeugt wird. Dieser Thread

bleibt inaktiv und dient nur als Name dieses Adressraums. Die Zerstörung des Tasks beschränkt sich damit wieder auf das Zerstören aller Threads inklusive dieses inaktiven Threads. Eine Ausnahme bildet hier Core. Da Core nie beendet wird, dient Cores initialer Thread auch als Platzhalter für Cores Adressraum.

3.6 Erzeugen und Zerstören von Threads

Das Fiasco-Backend macht zum Erzeugen und Zerstören von Threads intensiven Gebrauch des Fiasco-Syscalls `l4_inter_task_ex_regs` , der in der L4v4 API kein Äquivalent hat.

Das Starten eines Threads wird in L4v4 auf IPC abgebildet (siehe Abschnitt 2.2.3). Da es nicht ohne weiteres möglich ist, vom Pager die nötige Start-Nachricht an den neu erzeugten Thread zu schicken, benutze ich eine Funktion von Pistachio, die nicht direkt in der L4v4-Spezifikation erwähnt ist: Core, als privilegierter Task, schickt dem erzeugten Thread die Startnachricht.

Threads zu Stoppen wird direkt mit `L4_ThreadControl` erledigt (siehe Abschnitt 2.2.3).

3.7 Dataspaces

Dank der Ähnlichkeit der beiden APIs im Bezug auf Flexpages und die Handhabung derselben, konnte der Code für Dataspaces in Core in großen Teilen übernommen werden.

Durch das monotone Ersetzen von l4_threadid_t in L4_ThreadId_t entstand ein Bug in der Handhabung von Pagefaults, der das Unmappen von Dataspaces betraf. Dies machte sich erst bemerkbar, wenn der virtuelle Adressbereich wiederverwendet wurde und äußerte sich in wahllos überschriebenen Speicherbereichen. Das Problem war das Fehlen zweier Bits in einer `L4_Unmap` überreichten Flexpage.

3.8 I/O-Speicherverwaltung

Das Implementieren des Dienstes für I/O-Speicher erwies sich als problematisch. Über die *Kernel Info Page* ist zwar ersichtlich, welche Speicherbereiche Gerätespeicher sind, doch benötigt der VESA-Treiber auch Teile des BIOS-RAMs. Gibt der Dienst nur Speicher heraus, der als Gerätespeicher markiert ist, funktioniert dieser Treiber nicht.

Als provisorische Lösung macht der Dienst momentan keinen Unterschied zwischen RAM und Gerätespeicher, was offensichtlich eine große Sicherheitslücke darstellt.

3.9 Eine gemeinsame Basis für Fiasco und Pistachio?

In Abschnitt 3.1 habe ich erwähnt, dass ich ein Repository *base-l4/* verwenden wollte, um gemeinsamen Code der Fiasco- und Pistachio-Portierungen zu sammeln.

Dieser Ansatz hat sich nicht bewährt und der Inhalt des *base-l4/*-Repositorys beschränkt sich auf die x86-spezifische Lock-Implementation und Stubs für die nicht vorhandene C++-Laufzeitumgebung.

Zwar ähneln sich signifikante Teile beider Portierungen, die Extraktion dieses Codes hätte aber die Einführung einer neuen Abstraktionsschicht (und damit nicht-triviales Anpassen der Fiasco-Portierung) bedeutet. Dem gegenüber stünde eine sehr wahrscheinlich geringe Reduktion der Gesamtcodebasis bei gleichzeitiger Erhöhung der

Komplexität derselben, da es in meinen Augen fragwürdig ist, ob eine solche Abstraktionsschicht für andere Portierungen von großem Nutzen gewesen wäre.

3.10 Die Portierbarkeit von Bastei

Für die direkte Portierung von Bastei auf den Pistachio-Mikrokern musste der generische Code im *base/*-Repository nicht geändert werden. Trotz der aufgetretenen Probleme hat weder das Buildsystem noch die Aufteilung des Quellcodes in verschiedene Repositories der Portierung nennenswerte Steine in den Weg gelegt. Gerade das Buildsystem hat sich im Gegensatz als sehr praktisch herausgestellt.

Das Portieren des plattform-spezifischen Teils von Core war zwar mühsam und hätte durch eine klar definierte Schnittstelle zwischen generischem und plattform-spezifischem Code in Core deutlich erleichtert werden können[4], doch objektiv betrachtet hat sich die existierende Codebasis auch an dieser Stelle als recht portabel erwiesen.

Inwieweit diese Beobachtungen auf andere Portierungen extrapoliert werden können, ist fragwürdig, da die konzeptionellen Unterschiede zwischen Fiasco und Pistachio eher gering sind.

3.11 Multiprozessor-Primitive

Im Anschluss an die eigentliche Portierung habe ich den CPU-Dienst (siehe Abschnitt 2.3.2) um Methoden erweitert, die es möglich machen, den Multiprozessor-Support von Pistachio (Abschnitt 2.2.4) zu benutzen. Dazu musste in *base/* das Interface der `cpu_session` angepasst werden.

Eine `cpu_session` hat zwei neue Methoden:

- `unsigned int available_cpus()` zum Erfragen der Anzahl der verfügbaren Prozessoren und
- `set_cpu(Capability thread, unsigned int cpu)` zum Migrieren eines Threads auf eine andere CPU.

Die Methode `start` wurde um einen optionalen Parameter `cpu_no` erweitert, der angibt, auf welcher CPU ein Thread gestartet werden soll. Wird dieser Parameter nicht angegeben, wird CPU 0 verwendet.

Zusätzlich implementierte ich für die `cpu_session` einen neuen Dienstparameter `cpu`, mit dem sich eine Sitzung zum CPU-Dienst auf eine einzelne CPU beschränken lässt. Tasks, die mit einer solchen Sitzung zum CPU-Dienst erstellt werden, sind auf diese eine CPU beschränkt.

Die Erweiterungen des CPU-Dienstes sind explizit so gestaltet, dass Code, der sie nicht benutzt, nicht angepasst werden muss. Die Fiasco-Portierung wurde mit Dummys der entsprechenden Methoden ausgestattet, so dass Code, der dieses neue Interface benutzt, auch mit dem Fiasco-Port von Bastei verwendbar ist.

4 Leistungsbewertung

Die Portierung ist in sofern erfolgreich, dass das komplette Demo-Szenario, das für Bastei auf Fiasco entwickelt wurde, unverändert auf der Pistachio-Portierung läuft.

[4] Eine Anmerkung für Norman: Nein, ich glaube nicht, dass das bloße Aufteilen von plattformspezifischem und generischem Code in verschiedene Repositorys als definiertes Interface angesehen werden kann. :-)

CPU	Intel Core Duo L2400
Takt	1,66 GHz
L2 Cache	2048 KB
RAM	2048 MB

Tabelle 1: Technische Daten des für die Benchmarks benutzten Rechners

4.1 Verbleibende Probleme

Trotz intensivem Debugging konnten nicht alle Bugs ausgeräumt werden. Insbesondere ein Defekt hat diese Arbeit über eine lange Zeit begleitet und hat sich bis jetzt einer vollständigen Diagnose entzogen: Das Schließen von Sitzungen provoziert in manchen Fällen Pagefaults in Core oder Speicherkorruption in einzelnen Tasks.

Dieser Bug scheint durch die Vergabe von Thread-IDs beeinflusst zu werden, da die erste Vergabestrategie (Abschnitt 3.5) diesen Bug auch in einfachen Programmen wie den für die Benchmarks benutzten Programmen zum Vorschein gebracht hat. In der finalen Implementation manifestiert sich dieser Fehler nur noch in komplexeren Szenarien, wie z.b. der Demo.

4.2 Multiprozessorunterstützung

Die Tests der Multiprozessorunterstützung habe ich mittels der Kernel-Based Virtual Machine (KVM) von Linux vorgenommen, da sie das effiziente Emulieren eines Rechners beliebiger CPU-Zahl (bis zu einem gewissen Maximum) erlaubt.

Dabei traten recht zufällig IPC-Timeout-Fehler in Server-Loops in Aufrufen zu `L4_ReplyWait` mit Sende-Timout 0 auf, die aber für mich keinen Sinn machten, da alle Clients `L4_Call` benutzen und somit bei einer Antwort des Servers *immer* bereit sind, diese anzunehmen.

Auf echter Hardware sind diese Timeout-Fehler nie aufgetreten und das System verhält sich wie erwartet, was mich zu der Vermutung führt, dass es sich hierbei um einen Bug in KVM oder eine von KVM provozierte Race-Condition in Pistachio handelt.

4.3 Benchmarks

Wie ich anfangs erwähnt habe, kann diese Portierung als Vergleichsbasis zwischen Fiasco und Pistachio dienen. Um ein Stück in diese Richtung zu gehen, habe ich die IPC-Primitive von Bastei auf den beiden Kernen anhand von Benchmarks verglichen.

4.3.1 Der Benchmarkrechner

Zur Erstellung dieser Benchmarks wurde ein ThinkPad X60s verwendet. Die relevanten technischen Daten sind in Tabelle 1 aufgeführt.

Als Compiler kam GCC 3.4.6 zum Einsatz. Pistachio und Fiasco wurden für einen Pentium III und ohne Debug-Optionen konfiguriert und mit ihren Standard-Compilerflags gebaut. Das Bastei-Userland wurde mit `-O2 -march=pentium3` kompiliert.

4.3.2 IPC-Call

Bastei ist in seinem Umgang mit IPC ungewöhnlich, da es von der Client-Seite ausschließlich synchrone RPC-Aufrufe benutzt, die auf beiden Mikrokernen als IPC-Call

mit einem Stringparameter plus ReplyWait auf der Server-Seite implementiert sind (siehe Abschnitt 2.3.5). Diese Aufrufe sind der Grundstein, auf dem kompliziertere Mechanismen wie z.B. der Aufbau von Sitzungen beruhen. Insofern ist es interessant wie der Wechsel des Mikrokerns sich auf dieses Bastei-Primitiv auswirkt.

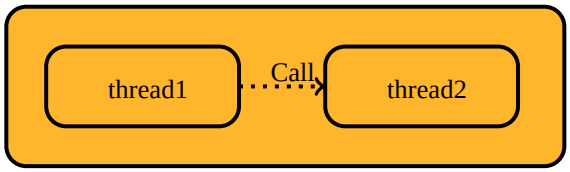

Abbildung 5: IPC innerhalb eines Tasks

Dazu implementierte ich zwei Benchmarks: Der erste Benchmark besteht aus zwei Threads in einem Task (Abbildung 5), bei dem gemessen wurde wie viele Takte ein Call von einem Thread zum anderen benötigt. Dabei wurde die Anzahl der übertragenen Datenworte variiert. Die Ergebnisse sind als *Pistachio IntraAS* resp. *Fiasco IntraAS* in Abbildung 7 dargestellt.

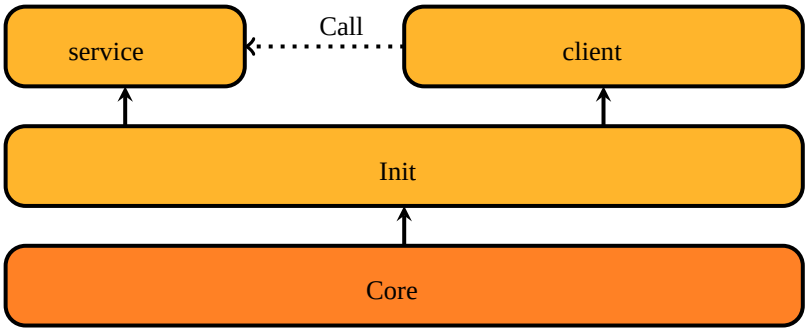

Abbildung 6: IPC zwischen Client und Service

Der zweite Benchmark besteht aus einem Dienst und einem Client (Abbildung 6). Auch bei diesem Benchmark wird gemessen, wie lange ein Call vom Client bei verschiedener Parameteranzahl dauert. Die Ergebnisse sind als *Pistachio C/S* resp. *Fiasco C/S* in Abbildung 7 dargestellt.

Bei den Ergebnissen sticht heraus, dass im Falle der Kommunikation zwischen Threads in einem Adressraum Pistachios IPC-Call ab zwei Datenworten nur etwa die Hälfte der Zeit seines Fiasco-Pendants benötigt. IPC zwischen Tasks ist auf Fiasco zwar langsamer, aber nur um wenige hundert Takte.

Der Grund, an dieser Stelle nur IPC mit wenigen Datenworten zu messen, ist, dass im existierenden Bastei-Code sehr selten IPC mit einer größeren Datenmenge als Nutzlast verwendet wird.

4.3.3 Aufbau von Sitzungen

Will ein Task eine Sitzung zu einem Dienst aufbauen, wird von ihm beginnend der Prozessbaum entlang der Elterntasks traversiert, bis ein Prozess gefunden ist, der die-

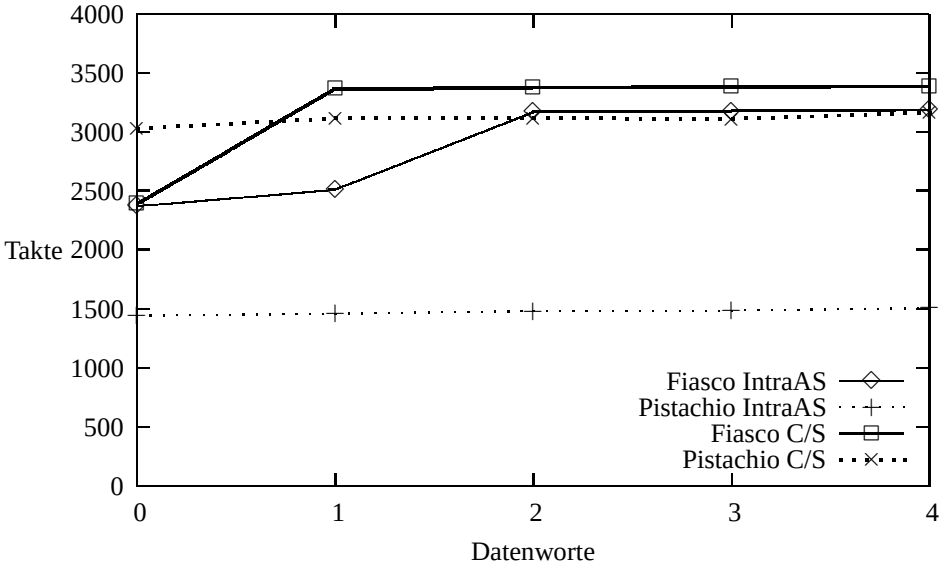

Abbildung 7: IPC-Call Performance

sen Dienst implementiert. Ich habe einen Benchmark implementiert, der zum einen zeigen soll, wie teuer das Aufbauen von Sitzungen ist und zum anderen, wie sich die Hierarchietiefe des beteiligten Prozessbaums auf diese Zeiten auswirkt.

Zum Erzeugen einer bestimmten Hierarchietiefe benutze ich einen Dummy-Task, deren Aufgabe es nur ist, sich selbst als Kind zu starten (analog zu `fork()`) bis die gewünschte Hierarchietiefe erreicht ist. An diesem Punkt wird dann das Programm gestartet, das die eigentlichen Benchmarks durchführt.

Als Dienste, zu denen Sitzungen aufgebaut werden, habe ich den Timer- und den CAP(ability)-Dienst gewählt. Der Timer-Dienst wird durch einen eigenen Task implementiert, wohingegen der CAP-Dienst von Core selbst implementiert wird.

4.3.4 Ergebnisse für den CAP-Dienst

Die Ergebnisse für den CAP-Dienst sind in Abbildung 9 zu sehen. Die Zeit, die zum Aufbau einer Sitzung zum CAP-Dienst benötigt wird, steigt annähernd linear mit der Anzahl der Hierarchielevel. Pistachio ist auch in diesem Benchmark minimal schneller als Fiasco, aber kaum bemerkenswert.

Beim Aufbau einer Sitzung zum CAP-Dienst (ohne Dummy-Eltern) werden vier RPC-Aufrufe getätigt, was acht IPC-Systemcalls entspricht:

1. Der Benchmark-Prozess ruft seinen Elternprozess Init auf, um die Sitzung zu erzeugen.

2. Init benutzt einen RPC, um Quota vom Benchmarkprozess zu Init zu transferieren.

3. Die Sitzungsanfrage wird an Core weitergeleitet.

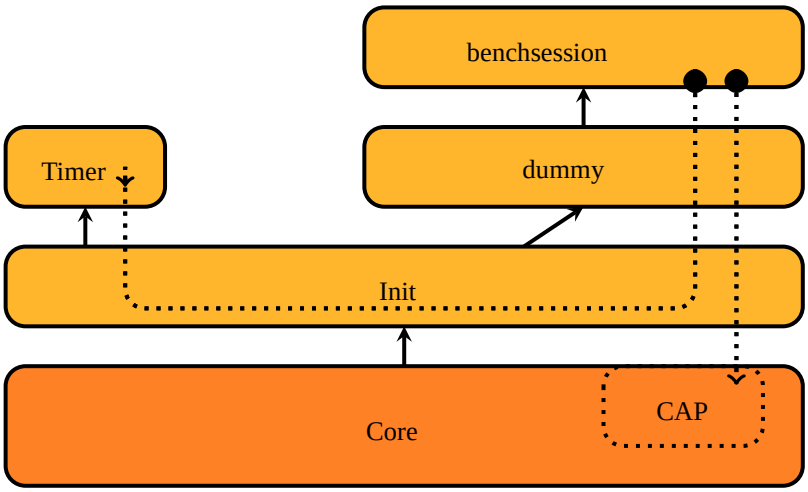

Abbildung 8: Aufbau von Sitzungen mit einem Dummy-Elternprozess ($level = 1$)

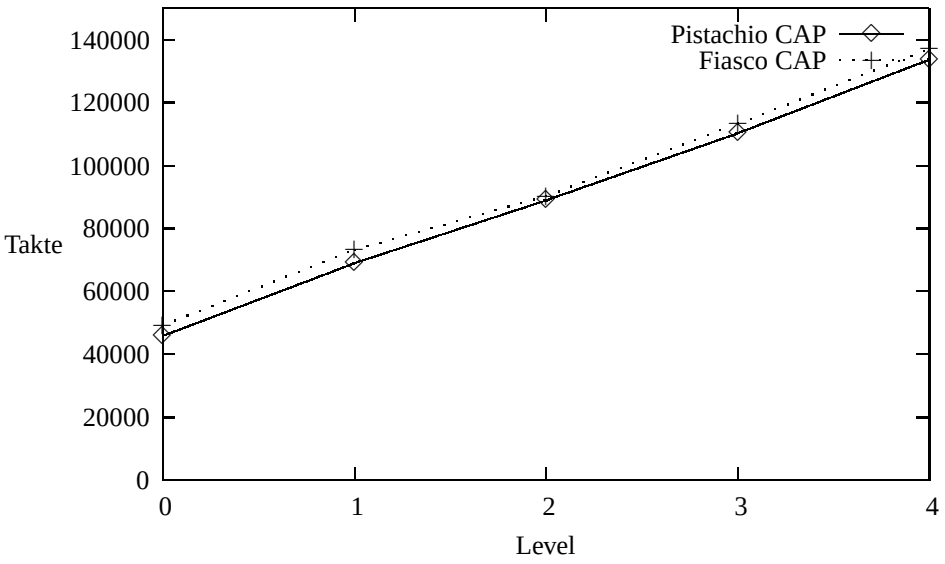

Abbildung 9: CAP Session

4. Core transferiert Quota von Init zu sich selbst, da Core den gewünschten Dienst implementiert.

Von diesen vier RPCs finden drei über Taskgrenzen hinweg statt und einer (4) geschieht innerhalb von Core. Nimmt man ausgehend von den IPC-Benchmarks im vorherigen Abschnitt etwas konservativ 3500 Takte für die ersten drei Calls und 1500 für den vierten an, kommt man auf etwa 12000 Takte reine IPC Kosten. Von den gemessenen 45800 Takten sind das 26%, was erstaunlich wenig ist, da der betreffende Codepfad aus kaum mehr als jenen IPC-Aufrufen besteht. Das legt die Vermutung nahe, dass die IPC-Benchmarks die Zeiten für einen Call für ein realistischeres Szenario unterschätzen. Die gleiche Vermutung betrifft auch die Zahlen für Fiasco. Auch hier kann man ausgehend von den IPC-Benchmarks nur rund ein Viertel der Takte mit IPC erklären.

Pro Hierarchielevel kommen dazu zwei weitere Calls: Die initiale Sitzungsanfrage und das Transferieren der Quota. Auf der Pistachio-Implementation schlägt jedes weitere Level mit etwa 20000-23000 weiteren Takten zu Buche.

4.3.5 Ergebnisse für den Timer-Dienst

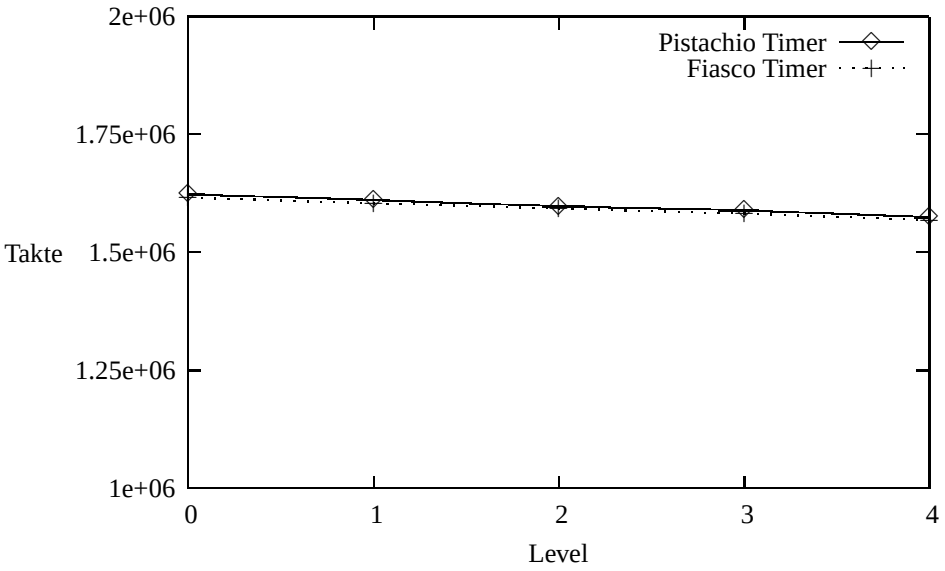

Abbildung 10: Timer Session

In Abbildung 10 sind die Zeiten für den Aufbau einer Sitzung zum Timer-Dienst zu sehen. Es fallen mehrere Dinge ins Auge:

- Die Ergebnisse für Fiasco und Pistachio sind *praktisch identisch*, was für diese recht komplexe Kommunikationsabfolge zwischen mehreren Prozessen eher unerwartet ist.
- Die Anzahl an Hierachiestufen hat einen *sehr geringen* Einfluss auf die benötigte Zeit zum Aufbau der Sitzung.

- Im Vergleich zum CAP-Dienst dauert der Aufbau einer Sitzung zum Timerdienst *unverhältnismäßig* länger.
- Die Dauer nimmt mit zunehmender Hierarchietiefe *ab*.

Etwas versteckt in den Zahlen ist noch folgende Erkenntnis:

- Die Ergebnisse für das Erzeugen einer Sitzung entsprechen recht genau $1ms$.

Die Ursachen für diese Ergebnisse stecken in den Unterschieden zwischen dem Aufbau einer Sitzung zum CAP-Dienst und einer Sitzung zum Timer-Dienst. Die Schritte, die beim Erstellen dieser Sitzung durchlaufen werden, sind:

1. Der Benchmark-Prozess ruft seinen Elternprozess Init auf, um die Sitzung zu erzeugen.
2. Init benutzt einen RPC, um Quota vom Benchmarkprozess zu Init zu transferieren.
3. Init sucht unter seinen Kindern den Task, der den Timer-Dienst implementiert und transferiert ihm Quota.
4. Init leitet die Sitzungsanfrage an den Timer-Dienst weiter.
5. Der Timer-Dienst erstellt eine `Session_component` mit einer zugehörigen `Server_activation` (ein Thread, der Aufrufe der Dienstmethoden empfängt und abarbeitet).

Alles in allem werden dazu 15 RPCs benutzt. Die wirklich teure Operation ist aber das Erstellen der `Server_activation`. In dieser Klasse wird im Konstruktor der Thread gestartet (Listing 6) und darauf gewartet, dass der gestartete Thread eine Capability erzeugt (Listing 7).

Listing 6: Der Konstruktor von `Server_activation`

```
Server_activation(Msgbuf_base *snd_msg, Msgbuf_base *rcv_msg,
                  bool start_on_construction = true,
                  const char *name = "activation")
  : Server_activation_base(snd_msg, rcv_msg)
{
  /* start execution of server activation thread */
  Thread_base::_start((void *)&_stack[STACK_SIZE - 4], name);

  /* wait until capability gets defined */
  _cap_valid.lock();

  /* start immediate request handling if specified */
  if (start_on_construction)
    start();
}
```

Es scheint sowohl auf Pistachio als auch Fiasco immer der Fall zu sein, dass der im Konstruktor gestartete Thread keine Zeit hat die Capability zu erzeugen und das

Listing 7: Der Kopf von `Server_activation::entry()`

```
void Server_activation_base::entry()
{
        Ipc_server srv(_snd_msg, _rcv_msg);
        _cap = srv;
        _cap_valid.unlock();

        /* ... */
```

Lock `_cap_valid` freizugeben, so dass der Konstruktur bei seinem Aufruf zu `_cap_valid.lock()` blockiert.

Durch die einfache Implementation von Locks (Abschnitt 2.3.11) entsteht eine Wartezeit, die auf beiden Kernen $1ms$ beträgt, welche wiederum diese Benchmarkergebnisse dominiert.

Ich vermute, dass die *Abnahme* der benötigten Zeit zum Aufbau einer Sitzung pro Hierarchielevel auch direkt mit diesem Sleep zusammenhängt. Für eine genauere Analyse hat an dieser Stelle die Zeit nicht mehr ausgereicht.

5 Ausblick

Auf der Grundlage dieser Arbeit ist es möglich größere Projekte zu implementieren, die sowohl auf Fiasco als auch Pistachio laufen. Es könnte untersucht werden, wie sich die Performance-Charakteristika der beiden Kerne auf der makroskopischen Ebene ausüben.

Diese Arbeit kann außerdem als Basis für eine Portierung auf den OKL4-Kern dienen. Da dieser ausgehend von der Pistachio-Codebasis implementiert wurde, sind die Unterschiede der APIs, die sich bei einem raschen Blick in das *OKL4 Microkernel Reference Manual*[Ope08], überschaubar. Zusätzlich unterstützt OKL4 Capabilites und Mutexes.

Da die Lock-Implementation eine Schwachstelle des bestehenden Systems darstellt, wäre es auch von Nutzen zu untersuchen, ob es sich lohnt eine performantere Lock-Implementation zu entwerfen oder die Benutzung von Locks im bestehenden Code durch IPC zu ersetzen.

6 Zusammenfassung

In dieser Arbeit wurde die Portierung von Bastei auf den Pistachio-Mikrokern ausgehend von der bestehenden Fiasco-Portierung beschrieben. Anhand der einzelnen Schritte der Portierung konnte gezeigt werden, dass Bastei gut auf einen anderen L4-Mikrokern portiert werden kann. Zusätzlich wurde Bastei um Primitive erweitert, um die Multiprozessorfähigkeiten von Pistachio zu benutzen.

Benchmarks zeigen, dass Basteis IPC-Primitive auf dem Pistachio-Mikrokern sehr ähnliche Performance-Eigenschaften wie auf Fiasco haben. Als Nebenprodukt dieses Vergleichs ist aufgefallen, dass die Lock-Implemention von Bastei ineffizient ist und Verbesserung bedarf.

Literatur

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new foundation for unix development. Technical report, Carnegie Mellon University, 1986.

[FH06] Norman Feske and Christian Helmuth. Design of the bastei os architecture. Technical report, TU Dresden, 2006.

[hin08] A real-time programmer's tour of general-purpose l4 microkernels. *EURASIP J. Embedded Syst.*, 2008(2):1–14, 2008.

[Lie95] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, 1995.

[Lie96a] Jochen Liedtke. *L4 Reference Manual*. German National Research Center for Information Technology, September 1996.

[Lie96b] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, 1996.

[MNN04] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2004.

[okl] Open kernel labs. `http://www.ok-labs.com/`.

[Ope08] Open Kernel Labs. *OKL4 Microkernel Reference Manual*, 2.1 edition, June 2008.