Das a-Kernel Projekt

Martin Schauer (martin.schauer@stud.informatik.uni-erlangen.de)

Friedrich-Alexander-Universität Erlangen-Nürnberg
Hauptseminar Aspektorientierte Systemprogrammierung

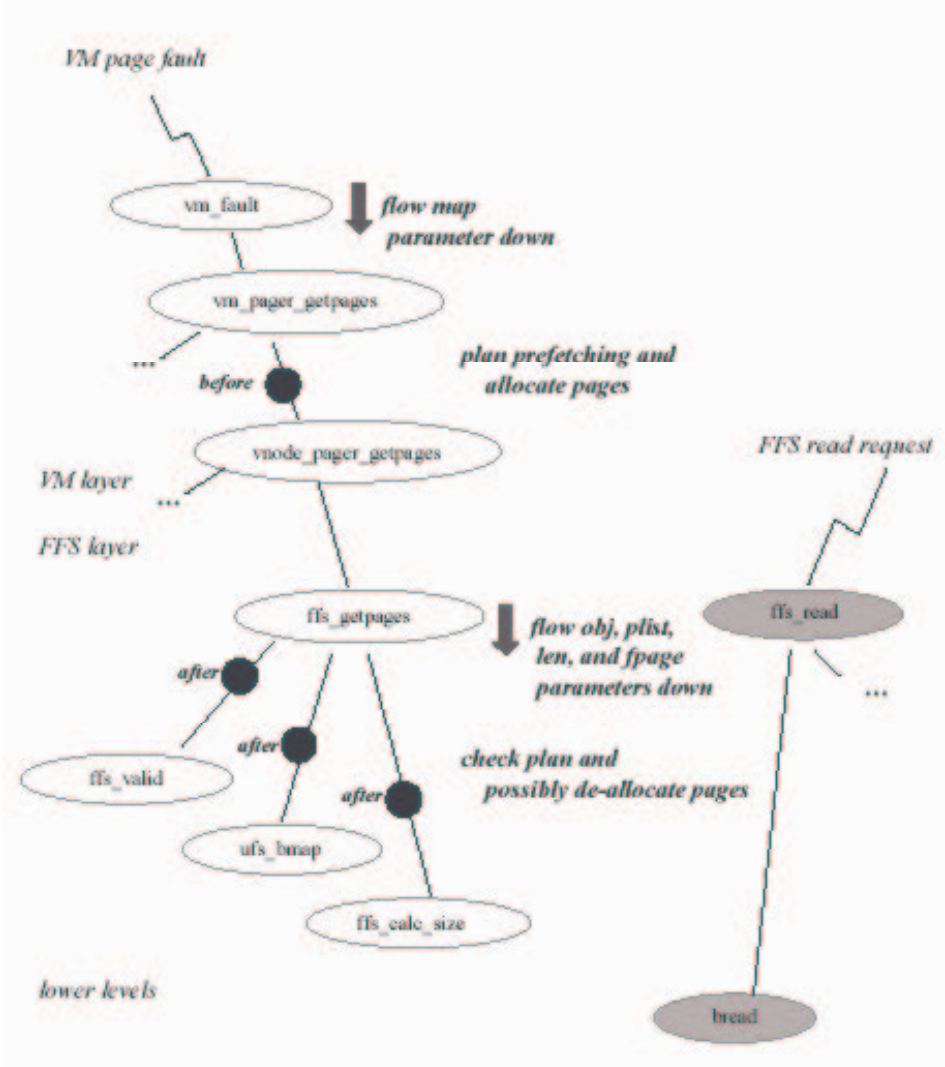
02. Juni 2003

Die Technik der aspektorientierten Programmierung verspricht, die Komplexität eines großen Softwareprojekts dadurch zu verringern, dass Crosscutting Concerns modularisiert werden können, und somit die Lesbarkeit, Wartbarkeit und Übersichtlichkeit des Codes verbessert wird. Das a-Kernel Projekt hat sich zum Ziel gesetzt, die genannten Vorteile der aspektorientierten Programmierung nicht nur zu postulieren, sondern auch anhand eines komplexen Softwareprojekts – eines Dateisystems – zu überprüfen, ob die erhofften Vorteile auch in der Praxis zum Tragen kommen. Dazu wurde zunächst das Dateisystem von FreeBSD v3.3 modifiziert und anschließend der Aufwand, wie er im prozeduralen Ansatz nötig ist, mit dem Aufwand des aspektorientierten Ansatz verglichen. Später wurde das selbe Experiment mit einem verteilten Dateisystem (NFS) wiederholt und ähnliche Vergleiche vorgenommen.

Das a-Kernel Projekt verwendet als aspektorientierte Programmiersprache eine (bis heute noch nicht vollständig implementierte) Erweiterung von C zu AspectC, das eine Untermenge von AspectJ darstellt. Wegen des fehlenden Compilers musste der Aspektcode per Hand kompiliert werden. AspectC stellt Sprachmittel zur Verfügung, die es ermöglichen, ein Aspekt-Modul (im Sinn von C, ein „Modul“ ist also eine Datei) zu definieren. Es können zum einen Pointcuts deklariert werden, die bei Bedarf über das Schlüsselwort „cflow“ Parameterwerte aus dem Ausführungskontext verwenden können, zum anderen Advices, deren Ausführungszeitpunkt durch die Schlüsselwörter „before“, „after“ oder „around“ festgelegt wird. Die Syntax sollte bei den Codebeispielen deutlich werden und wird deswegen hier nicht weiter erläutert werden.

Das erste Anwendungsbeispiel des a-Kernel Projekts modifiziert das Dateisystem von FreeBSD v3.3 in den Bereichen „Page Fault Handling“ zusammen mit „Prefetching“. Ein Prozess löst einen Page Fault aus, wenn er in seinem virtuellen Adressraum eine Seite

anspricht, die gerade nicht im Betriebssystempuffer vorhanden ist. In diesem Fall kümmert sich ein Page Fault Handler darum, dass die benötigte Seite in den Puffer geladen wird. Dazu benötigt der Page Fault Handler die Hilfe des Dateisystems. Dieses ordnet den Block einer Datei zu und gibt den Leseauftrag an den Treiber weiter. Das Zugriffsverhalten der meisten Programme zeigt gewisse Muster auf, so dass bei Beobachtung der zuletzt in den Puffer geladenen Seiten gewisse Heuristiken eingesetzt werden können, um die Effizienz zu erhöhen. Mit anderen Worten kann das Dateisystem alle Zugriffe beobachten und „auf Verdacht“ Seiten im Voraus (Prefetching) in den Puffer laden (ohne dazu aufgefordert worden zu sein). Zur Unterstützung dieses Mechanismus stellt die Anwendung der „VM-Schicht“ Informationen zu ihrem Zugriffsverhalten zur Verfügung: „normal“ oder „sequential“. Sobald ein Page Fault auftritt, nützt das Dateisystem diese Information aus, um das Prefetching anzustoßen. Folgende Graphik soll den Kontrollfluss zeigen:

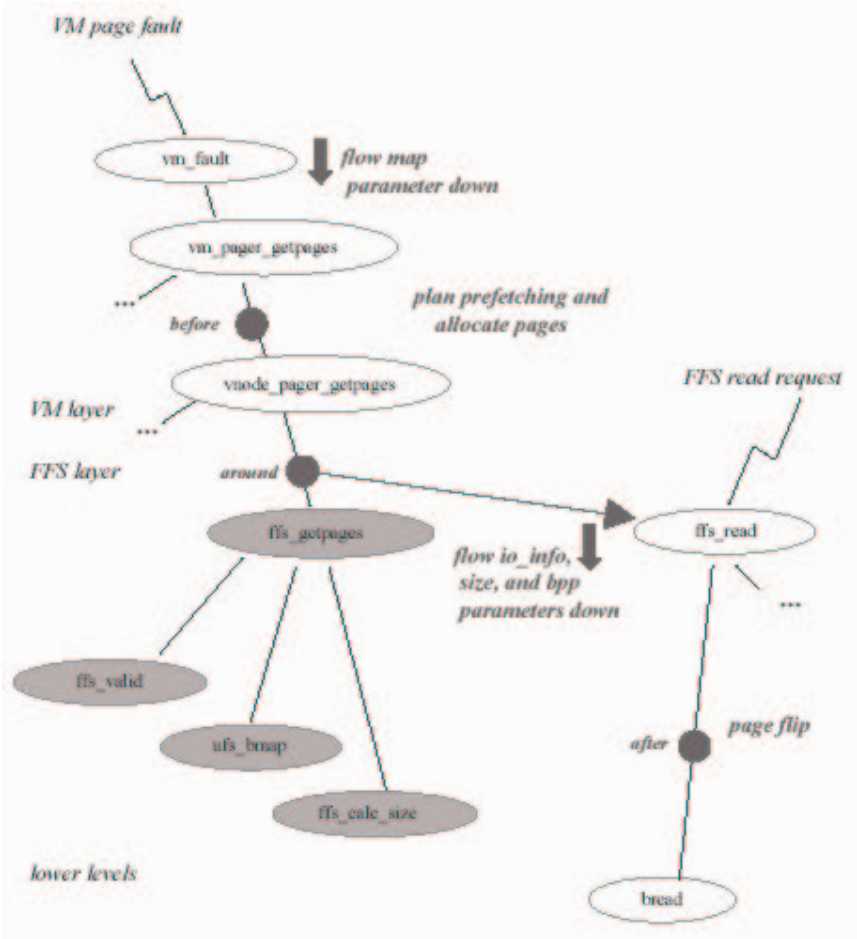


Der entsprechende AspectC-Code sieht folgendermaßen aus:

```
aspect normal prefetching {
	pointcut vm_fault_cflow(vm_map_t map):
		cflow(calls(int vm_fault(map,..)));
	pointcut ffs_getpages_cflow(vm_object_t obj, vm_page_t* plist, int len, int fpage):
		cflow(calls(int ffs_getpages(obj,plist,len.fpage)));
	before(vm_map_t map, vm_object_t obj, vm_page_t* plist, int len, int fpage):
		calls(int vnode_pager_getpages(obj,plist,len,fpage)) && vm_fault_cflow(map){
		if(obj->declared_behaviour == NORMAL) {
			vm_map_lock(map);
			plan_and_alloc_normal(obj,plist,len,fpage);
			vm_map_unlock(map);
		}
	}
	after(vm_object_t obj, vm_page_t* plist, int len, int fpage, int valid):
		calls(valid ffs_valid(..)) && ffs_getpages_cflow(obj,plist,len,fpage) {
		if(valid) dealloc_all_prefetch_pages(obj,plist,len,fpage);
	}
	after(vm_object_t obj, vm_page_t* plist, int len, int fpage, int error, int reqblkno):
		calls(error ufs_bmap(struct vnode*, reqblkno, ..)) &&
		ffs_getpages_cflow(obj,plist,len,fpage) {
		if(error || (reqblkno == -1)) dealloc_all_prefetch_pages(obj,plist,len,fpage);
	}
	after(vm_object_t obj, vm_page_t* plist, int len, int fpage, struct t_args* trans_args):
		calls(int ffs_calc_size(trangs_args)) && ffs_getpages_cflow(obj,plist,len,fpage) {
		dealloc_noncontig_prefetch_pages(obj,plist,len,fpage,trans_args);
	}
}
```

Im Aspektcode sieht man, dass vor der Behandlung eines Page Faults geplant wird, welche Seiten durch das Prefetching (evtl.) geladen werden sollen. Berücksichtigt hierbei werden z. B. das Zugriffsmuster und die Kosten für die Leseoperation. Hier wird auch der Speicher im Systempuffer zur Verfügung gestellt (um die Sperrung der Seitentabelle möglichst kurz zu halten). Im Anschluss erfolgt dann die Leseoperation der Seite, die den Page Fault ausgelöst hat. Danach greifen die drei after Advices. Hier werden Bedingungen überprüft, um zu entscheiden, ob das Prefetching tatsächlich durchgeführt werden soll. Wenn eine der Bedingungen zutrifft ist das Prefetching nicht mehr von Interesse und die bereits allokierten Seiten werden wieder freigegeben. Das ist im ersten Fall, wenn die Seite sich inzwischen schon im Systempuffer befindet, im zweiten, wenn die Seite, die den Page Fault ausgelöst hat, nicht auf der Platte vorhanden ist bzw. neu allokiert wurde, im dritten, wenn die geplanten Seiten nicht mehr zusammenhängend auf der Platte liegen.

Stellt obiger Code das Prefetching für die „normale“ Zugriffsart dar, sieht die Graphik und der Code für den sequentiellen Zugriff zwar ähnlich, aber doch etwas anders aus:




```
aspect sequential_prefetching {
        pointcut vm_fault_cflow(vm_map_t map):
                cflow(calls(int vm_fault(map,..)));
        pointcut ffs_read_cflow(struct vnode* vp, struct uio* io_inf, int size, struct buff** bpp):
                cflow(calls(int ffs_read(vp,io_inf,size,bpp)));
        before(vm_map_t map,vm_object_t obj,vm_page_t* plist,int len,int fpage):
                calls(int vnode_pager_getpages(obj,plist,len,fpage)) && vm_fault_cflow(map) {
                        if(obj->declared_behaviour == SEQUENTIAL) {
                                vm_map_lock(map);
                                plan_and_alloc_sequential(obj,plist,len,fpage);
                                vm_map_unlock(map);
                        }
                }
        around(vm_object_t_obj, vm_page_t* plist, int len, int fpage):
          calls(int ffs_getpages(obj,plist,len,fpage)) {
            if(obj->behaviour == SEQUENTIAL) {
              struct vnode * vp = obj->handle;
              struct uio* io_inf = io_prep(plist[fpage]->pindex, MAXBSIZE,curproc);
              int error = ffs_read(vp,io_inf,MAXBSIZE,curproc->p_ucred);
              return cleanup_after_read(error,obj,plist,len,fpage);
            } else proceed;
        }
        after(struct uio* io_info,int size,struct buf* bpp):
          calls(int bread(..)) && vm_fault_cflow(..) &&
          ffs_read_cflow(struct vnode*,io_info,size,bpp) {
            flip_buffer_pages_to_allocated_vm_pages((char *)bpp->b_data,size,io_info);
        }
}
```

Der Unterschied besteht vor allem darin, dass der Zugriffspfad ein anderer ist. Das drückt sich im Aspektcode im around und dem after Advice aus. Der around Advice leitet den Kontrollfluss im sequentiellen Fall von ffs_getpages auf ffs_read um. Das hat zur Folge, dass ohne weitere Berücksichtigung der Kosten die folgenden Blöcke in der Datei bis zu einer gewissen maximalen Anzahl in den Systempuffer gelesen werden. Auch die Behandlung nach der Leseoperation ist eine andere. Die gelesenen Seiten werden nicht in den Adressraum des VM-Objekts kopiert, sondern dieser Adressraum wird komplett ausgetauscht. Das ist natürlich nur aus diesem speziellen Zugriffspfad heraus sinnvoll.
Die hier vorgestellte aspektorientierte Implementierung wurde von den Mitgliedern des a-Kernel Projekts mit der originalen Implementierung in FreeBSD verglichen. Folgendes Zitat findet man in den Papers des a-Kernel Projekts:

„In the original FreeBSD v3.3 code, the implementation of prefetching is both scattered and tangled. The code is spread out over approximately 260 lines in 10 clusters in 5 core functions from two subsystems."

Das liegt in erster Linie daran, dass der Aspekt die in der Graphik angedeuteten Schichten „senkrecht" schneidet. Darum passt die Implementierung des Aspekts nicht in die „Module" der Implementierungssprache C (also Dateien), sondern muss in mehreren Modulen geschehen. Der Aspekt Prefetching kann aber durch die aspektorientierte Orientierung in sämtliche Schichten eingewoben werden, bleibt aber trotzdem ein Modul der Implementierungssprache AspectC. Man hat somit zum einen sehr viel an Übersichtlichkeit und Wartbarkeit hinzugewonnen, zum anderen aber auch eine gewisse „Plug And Play"-Fähigkeit. Auch dies wurde beim a-Kernel Projekt überprüft, indem man das Dateisystem mit allen Prefetching-Kombination (ohne, normal, sequential, normal + sequential) kompiliert und getestet hat.
Man merkt also, dass aspektorientiertes Design durchaus geeignet ist, ein lokales Dateisystem zu beschreiben und zu implementieren. Die Mitarbeiter des a-Kernel Projekts haben ähnliche Untersuchungen auch für das verteilte Dateisystem NFS (aufbauend auf einer Client-Server Architektur) durchgeführt. Im verteilten Fall handelt man sich zwei gegenläufige Probleme ein. Zur Performancesteigerung können sowohl Server als auch Clients Kopien einer Datei im jeweils lokalen Cache halten. Das führt zu Problemen, um bei Schreibzugriffen (es können ja auch mehrere Clients beteiligt sein) die Konsistenz in allen Caches zu erhalten. Geht man dieses Problem an, wird man notwendigerweise im Gegenzug die hinzugewonnene Performance wieder verlieren. Je nach Anwendungszweck wird man entweder etwas weniger Konsistenz oder etwas weniger Performance hinnehmen. Das Ziel wäre also jedem

Anwendungszweck genau das zur Verfügung zu stellen, was nötig ist, um dem geforderten „Verhältnis“ zwischen Konsistenz und Performance zu genügen.
Dahingehend wurde versucht, aspektorientierte Erweiterungen für beide Gesichtspunkte zu entwickeln. Zur Konsistenzerhöhung sieht die Originalimplementierung bereits vor, gecachte Kopien mit einem Zeitstempel zu versehen, so dass der Client beim Server nach dem aktuellen Zeitstempel diese Blocks fragen und evtl. den Block nachladen kann. Das Problem dabei ist, dass mehrere konkurrierende Schreibzugriffe zur selben Zeit dennoch zu Inkonsistenzen führen können. Um das Intervall, in dem Inkonsistenzen auftreten können, möglichst gering zu halten, ist es wichtig, möglichst zeitnah zu potentiellen Änderungen den Zeitstempel zu prüfen (siehe Codefragment). Ein zweiter, eher Server-basierter Ansatz, sieht vor, den Cache des Clients zu invalidieren, indem der Server sich den Zustand eines Blocks merkt. Beide Varianten sind in folgendem Codebeispiel verdeutlicht:

```
aspect validation_check {
	pointcut validation_check_points(file_id fid):
		calls(int nfs_client_read(fid,..)) || calls(int nfs_client_write(fid,..)) ||
		calls(int nfs_client_cache_check(fid,..)) ||
		calls(void daemon_write(fid,..)) ||
		calls(void daemon_prefetch(fid,..));
	before(file_id fid): validation_check_points(fid) {
		get_fresh_timestamp(fid);
	}
}
aspect active_invalidation {
	after(file_id fid): calls(int nfs_server_read(fid,..)) {
		update_state_info(fid->state_info);
	}
	after(file_id fid): calls(int nfs_server_write(fid,..)) {
		check_state_and_invalidate(fid->state_info);
	}
}
```

Das erste Beispiel zeigt, dass zeitnah zu einer Lese-/Schreiboperation ein neuer Zeitstempel im before Advice geholt wird. Im zweiten Beispiel sieht man die Operationen zum Ändern des Zustands eines Blocks auf dem Server in den after Advices.
Zur Erhöhung der Performance wird vorgeschlagen, bei einem sequentiellen Zugriff (in der Art wie er beim lokalen Dateisystem beschrieben ist) alle folgenden Blöcke (bis zu einem Maximalwert) gleich mitzulesen, also „aggressives Prefetching“ zu betreiben. Auch das lässt sich aspektorientiert sehr kompakt schreiben:

```
aspect sequential_prefetch {
	/* client-side advice */
	before(file_id fid, block_num bnum): calls(int nfs_client_read(fid,bnum,..)) {
		update_pattern_of_access(fid,bnum);
	}
	after(file_id fid, block_num bnum): calls(int nfs_client_read(fid,bnum,..)) {
		if(fid->access_pattern == SEQUENTIAL)
			nfs_daemon_prefetch(fid,bnum+1);
	}
	/* server-side advice */
	after(file_id fid, block_num bnum): cflow(calls(void nfs_daemon_prefetch(..))) &&
	  calls(int nfs_server_read(fid,bnum,..)) {
		aggressive_prefetch_into_server_cache(fid,bnum+1,bnum+MAX_PREFETCH);
	}
}
```

Der Vorteil des aspektorientierten Ansatzes gegenüber des „klassischen“ Vorgehens ist, wie auch beim Dateisystem im lokalen Fall, keinen Eingriff in den Originalcode vornehmen zu müssen. Zusätzlich kann man zur Übersetzungszeit je nach Wunsch des Anwenders den Server/Client so konfigurieren, dass er möglichst gut in das Anwendungsszenario passt.

Als Fazit halten die Mitarbeiter des a-Kernel Projekts fest, dass es mit Hilfe der aspektorientierten Programmierung möglich ist, auch - oder gerade in komplexen Softwareprojekten - bestimmte Teilgebiete getrennt von der restlichen Implementierung zu modellieren. Das unterstützt dabei, einerseits die Übersichtlichkeit und Wartbarkeit des gesamten Projekts zu erhalten, andererseits auch die Konfigurierbarkeit nach Anwendungszweck zu gewährleisten. Bei einer „klassischen“ Implementierung wären diese Kriterien nicht ohne weiteres erfüllt, da es sich bei Aspekten oft um Dinge handelt, die nicht auf ein Modul (im Sinne der Implementierungssprache) beschränkt sind, sondern sich durch mehrere Schichten ziehen. Die Zeichen stehen also gut, dass in Zukunft die Grundlagen zur Lösung dieses Problems vorhanden sind.

Literatur:

[1]: http://www.cs.ubc.ca/labs/spl/projects/a-kernel.html (enthält Links auf alle übrigen Dokumente)

[2]: Yvonne Coady, Gregor Kiczales, Mike Feeley and Greg Smolyn. "Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code", FSE 2001.

[3]: Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong. "Structuring System Aspects", ASOC Workshop at ICSE 2001.

[4]: Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson and Joon Suan Ong. "Structuring Operating System Aspects", Communications of the ACM, October 2001.

[5]: Yvonne Coady, Alex Brodsky, Dima Brodsky, Jody Pomkoski, Stephan Gudmundson, Joon Suan Ong and Gregor Kiczales. "Can AOP Support Extensibility in Client-Server Architectures?", AOP Workshop at ECOOP 2001.

[6]: Yvonne Coady, Gregor Kiczales, Michael Feeley, Norman Hutchinson, Joon Suan Ong and Stephan Gudmundson. "Exploring an Aspect-Oriented Approach to OS Code", ECOOP 2001

[7]: Alex Brodsky, Dima Brodsky, Ida Chan, Yvonne Coady, Stephan Gudmundson, Jody Pomkoski, Joon Suan Ong. "Coping with Evolution: Aspects vs Aspirin?", ASOC Workshop at OOPSLA 2001