Großer Beleg

Entwicklung eines IDL-Compilers für L4Re

Jan Bierbaum

6. August 2012

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Alexander Warg

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig erstellt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Dresden, den 6. August 2012

Jan Bierbaum

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung

Die Rechenleistung der Computersysteme unserer Zeit wächst zunehmend an, wohingegen Platz- und Energiebedarf weiter sinken. Während die Geräte kleiner werden und aus dem Alltag vieler Menschen kaum mehr wegzudenken sind, konnte die eingesetzte Software mit dieser rasanten Entwicklung nicht Schritt halten. Die hohe Komplexität von Betriebssystemen und Anwendungsprogrammen geht einher mit einer hohe Fehleranfälligkeit und führt damit letztlich zu Instabilität und mangelhafter Sicherheit der eingesetzten Software. Obwohl Rechentechnik mittlerweile allgegenwärtig ist, bleibt sie doch unzuverlässig.

Mikrokerne Systeme auf Basis von Mikrokernen bieten einen Ausweg aus diesem Dilemma. Der auf minimale Größe ausgerichtete Betriebssystemkerns kann verifiziert werden [KEH+09] und Ähnliches ist auch für die Systembasisdienste vorstellbar. Damit rückt formal fehlerfreie Software zumindest für kritische Anwendungen in greifbare Nähe. Der Philosophie des Kern folgend, stellen darauf aufbauende Systeme ihre Funktionalität typischerweise in Form isolierter Server bereit. Jeder dieser Server bietet lediglich eine begrenzte Funktionalität (beispielsweise die eines Gerätetreibers oder die einer Benutzerverwaltung); erst ihre Kombination ermöglicht die Realisierung komplexer Anwendungen. Durch diese Modularisierung bleiben Flexibilität und Zuverlässigkeit des Mikrokerns in hohem Maße für den Benutzer erhalten. Er kann sein System aus Komponenten aufbauen, deren Entwicklern er vertraut beziehungsweise von vertrauenswürdigen Personen oder Organisationen zusammengestellte Distributionen einsetzen.

IDL-Compiler Für Programmierer dagegen bedeutet diese Client-Server-Architektur zusätzlichen Aufwand bei der Erstellung und Pflege von Kommunikationscode. Bei der Arbeit mit verteilten Systemen, die eine sehr ähnliche Struktur aufweisen, ist daher die Verwendung von IDL-Compilern üblich. Dabei handelt es sich um spezialisierte Programme, durch deren Einsatz sich die Erzeugung geeigneten Kommunikationscodes auf die reine Spezifikation der gewünschten Kommunikation reduziert – vereinfacht betrachtet gibt der Entwickler lediglich vor, welche Daten übertragen werden müssen anstatt die Kommunikationsmechanismen des verwendeten Betriebssystems selbst anzusteuern.

Die Implementierung eines vollwertigen IDL-Compilers stellt allerdings ebenfalls einen nicht unerheblichen Aufwand dar. Die (syntaktische und semantische) Analyse der verwendeten IDL gestaltet sich schwierig, sofern sie nicht nur einfachste Szenarien abbilden soll. Einfache Parser-Generatoren stoßen aufgrund der meist kontextsensitiven Natur der IDLs schnell an ihre Grenzen. Darüber hinaus ist die direkte Einbindung externen Codes einer höheren Programmiersprache von Vorteil, um beispielsweise vorhandene Typdeklarationen in die IDL einbinden zu können. In diesem Fall muss natürlich auch

der IDL-Compiler fähig sein, diese Definitionen und damit eine Teilmenge der jeweiligen Hochsprache zu verarbeiten.

Ziel Das Ziel der vorliegenden Arbeit ist es, einen IDL-Compiler für Fiasco.OC/L4Re zu entwickeln. Im Zentrum stehen dabei Bemühungen bei der Umsetzung bereits bestehende Infrastruktur – insbesondere einen vorhandenen Hochsprachen-Compiler – in möglichst großem Umfang mitzunutzen. Neben der Einsparung eines erheblichen Teils des Entwicklungsaufwandes führt eine solche Wiederverwendung von bewährtem Programmcode auch zu einer geringeren Anzahl von Fehlern im fertigen Compiler.

2. Grundlagen

Dieses Kapitel vermittelt die Grundlagen, die zum Verständnis der vorliegenden Arbeit nötig sind: Zunächst werden Mikrokerne und darauf basierende Systeme vorgestellt. Ein besonderer Fokus liegt dabei auf Fiasco.OC und L4Re, dem System, für das Cid entworfen wurde. Die folgenden Abschnitte erläutern das Konzept einer IDL und geben einen kurzen Überblick über die C++-Compiler, die als mögliche Basis für den neuen IDL-Compiler in Betracht gezogen wurden.

2.1. Fiasco.OC und L4Re

Mikrokern-basierte Betriebssysteme, wie Fiasco.OC, unterscheiden sich in Aufbau und Funktionsweise grundlegend von den herkömmlichen, monolithischen Systemen. Diese Unterschiede erfordern eine besondere Herangehensweise, die in groben Zügen an die Arbeit mit verteilten Systemen erinnert.

2.1.1. Mikrokern-basierte Betriebssysteme

Viele der verbreiteten Betriebssysteme (Windows, GNU/Linux, BSD, ...) basieren auf sogenannten monolithischen Kernen. Ein derartiger Systemkern – das heißt alle Komponenten, die im privilegierten Modus der CPU ausgeführt werden – ist sehr umfangreich und umfasst neben den eigentlichen Verwaltungsaufgaben auch Gerätetreiber und weitere, nicht essentielle Elemente bis hin zu komplexen Bestandteilen wie Dateisystemen oder Protokoll-Stacks.

Mit Mikrokernen hingegen wird, wie der Name bereits andeutet, der Ansatz eines möglichst kleinen Betriebssystemkerns verfolgt. Dieser stellt ausschließlich die zur Umsetzung konkreter Richtlinien erforderlichen Mittel bereit, zwingt dem System aber keine auf. Statt als Bestandteile des Kerns werden so viele Komponenten wie möglich als separate, im Benutzermodus ablaufende Programme implementiert. Diese Entflechtung und die damit verbundene Verminderung von Menge und Komplexität des im Kernmodus laufenden Codes führt zu wesentlichen Verbesserungen im Hinblick auf Sicherheit, Stabilität und Flexibilität des Systems [Lie95]:

- Fehlerhafte Systemkomponenten, die nun im nicht-privilegierten CPU-Modus laufen, verfügen nicht mehr über Zugriffsrechte auf Speicherbereiche des Kerns. Folglich ist der Betriebssystemkern stabiler und beim Auftreten eines Fehlers ist der Neustart einzelner abgestürzter Komponenten (*„micro reboot“*) möglich. Insbesondere Gerätetreiber, die häufigste Ursache für Systemabstürze [CYC+01], werden so wirkungsvoll isoliert.

- Ein modularer Aufbau des System, sowie die Definition und Verwendung von klaren Schnittstellen zwischen dessen einzelnen Komponenten wird nahegelegt, wenn auch nicht erzwungen. Anwendungen im Benutzermodus können vom Benutzer natürlich auch in Mikrokern-basierten Systemen monolithisch strukturiert sein.

- Anpassungen an spezifische Anforderungen (beispielsweise in Bezug auf das Auslagern von Speicher zeitkritischer Anwendungen) sind verhältnismäßig einfach umzusetzen, indem der Benutzer passende Systemkomponenten bereitstellt. Am Kern selbst hingegen sind keine Änderungen erforderlich.

- Die vertrauenswürdige Basis (*„trusted computing base“*, TCB) fällt signifikant kleiner aus als bei monolithischen Systemen.

2.1.2. Fiasco.OC

Nachdem die Mikrokerne der ersten Generation diese Erwartungen aufgrund ihrer sehr geringen Ausführungsgeschwindigkeit nicht befriedigend erfüllen konnten, setzte ein Umdenken ein. Der L4-Mikrokern gilt als Vertreter einer zweiten Generation von Mikrokernen, die nicht, wie ihre Vorgänger, durch das schrittweise Auslagern von Funktionalität aus monolithischen Betriebssystemkernen geschaffen, sondern von Grund auf als Mikrokern entworfen und optimiert wurden. Das Augenmerk lag dabei insbesondere auf der schnellen Ausführung von Inter-Prozess-Kommunikation (*„inter-process communication“*, IPC), die eine zentrale Rolle innerhalb eines Mikrokerns spielt und sich daher maßgeblich auf dessen Leistungsfähigkeit auswirkt [HHL+97].

Der L4-Philosophie gemäß, werden im Kern ausschließlich Funktionen toleriert, die außerhalb nicht umsetzbar sind oder deren Nichtaufnahme in den Kern massive Leistungseinbußen, insbesondere in Bezug auf die Ausführungsgeschwindigkeit, zur Folge hätte. Dementsprechend stellen L4-Systeme lediglich drei Paradigmen zur Verfügung [Lie96a, Lie96b]:

Task/Adressraum → Isolation und Ressourcenverwaltung
Thread → Aktivitätsträger innerhalb eines Adressraums
IPC → (synchroner) Datenaustausch zwischen Threads

Mit *„Fiasco“* wurde an der TU-Dresden bereits seit mehreren Jahren ein echtzeitfähiger L4-Mikrokern entwickelt. Der Nachfolger *„Fiasco.OC“* ist, wie auch der ursprüngliche Fiasco-Kern, in C++ implementiert, bricht aber mit dem L4-API und erweitert das System um eine Capability-basierte Rechteverwaltung, Hardware-Virtualisierung sowie Mehrprozessorunterstützung. Er gehört damit der dritten Generation von Mikrokernen an [fia, LW09].

Unter Fiasco.OC stehen Kerndienste für Anwendungen ausschließlich in Form von Capabilitys zur Verfügung; der einzige verbliebene Systemaufruf ist dementsprechend der Aufruf einer solchen Capability (*„invocation“*). Der erforderliche Datenaustausch mit dem Kern findet dabei entweder direkt als Teil dieses Aufrufs oder unter Verwendung des Thread-eigenen UTCB (*„User Level Thread Control Block“*) statt.

2.1.3. L4Re

Aufgrund des definitionsgemäß sehr beschränkten, von einem Mikrokern direkt bereitgestellten Funktionsumfangs, ist es zweckmäßig diesem ein Basissystem zur Seite zu stellen, um effizient arbeiten zu können. Diese Laufzeitumgebung stellt Funktionen auf einer höheren Abstraktionsebene bereit, so wie sie in monolithischen Systemen in der Regel vom Kern angeboten werden: Gerätetreiber, Dateisysteme, usw. – gegebenenfalls bis hin zur vollen POSIX-Kompatibilität. Für Fiasco.OC stellt L4Re (*„L4 Runtime Environment“*) diese Basis bereit und löst damit L4Env ab, das bei den nicht-Capability-basierten Vorgängern von Fiasco.OC zum Einsatz kommt.

L4Re bildet verschiedene Dienste (Dateisystem, Speicherverwaltung, Konsole, ...) durch Protokolle ab. Ein Protokoll definiert die Schnittstelle, die ein Server bereitstellen muss, der diesen Dienst anbietet. Bei Anfragen an einen solchen Server wird das gewünschte Protokoll im Rahmen der IPC durch seine Protokoll-ID (eine einfache Ganzzahl, vergleichbar dem in Abschnitt 2.2 auf dieser Seite noch diskutierten Opcode) ausgewählt. Protokoll-ID und Opcode gewährleisten gemeinsam die eindeutige Unterscheidung aller von einem Server bereitgestellten Funktionen, auch wenn dieser mehrere Protokolle implementiert. Für die Interaktion mit dem Mikrokern kommen ebenfalls einige vordefinierte Protokolle zum Einsatz, so beispielsweise für die Behandlung von Seitenfehlern oder die Verwaltung von Adressräumen sowie deren Threads.

2.2. IDL und IDL-Compiler

Bei eingehender Betrachtung weisen Mikrokern-basierte Betriebssysteme eine Struktur auf, die verteilten Systemen sehr ähnlich ist: Funktionalität wird von verschiedenen eigenständigen Programmen, den sogenannten Servern, bereitgestellt und von anderen Programmen, den sogenannten Clients, genutzt. Dieses Prinzip ist als Fernaufruf einer Prozedur/Funktion (*„remote procedure call“*, RPC) bekannt. Dabei ist es nicht nur möglich, sondern durchaus üblich, dass ein Programm beide Rollen einnimmt, das heißt Dienste anbietet und zu deren Erbringung wiederum andere Server nutzt.

Die Kommunikation zwischen den einzelnen Systemkomponenten ist zur Ausführung nicht-trivialer Programme zwingend notwendig und läuft praktisch immer nach dem folgenden Schema ab:

1. **Vorbereiten des Nachrichtenpuffers (*„Marshalling“*)** — Alle vom Server benötigten Daten werden durch den Client in einen durch das System bereitgestellten Nachrichtenpuffer kopiert. Bietet der Server mehrere Funktionen an, so befindet sich unter den Daten auch eine eindeutige, zumeist numerische Kennung der aufgerufenen Funktion; der sogenannte Opcode.

2. **Übertragung des Puffers an den Server** — Das unterliegende Kommunikationssystem[1] überträgt den Inhalt des Nachrichtenpuffers an den Server.

[1] Im Fall von L4Re ist das der von Fiasco.OC bereitgestellte IPC-Mechanismus.

3. **Auslesen des Nachrichtenpuffers (*„Unmarshalling“*)** — Alle eingehenden Daten müssen, um vom Server verwendet werden zu können, wieder in ihre ursprüngliche beziehungsweise eine vom Server erwartete Form gebracht werden. In aller Regel geschieht das, indem sie aus dem Nachrichtenpuffer in server-lokale Puffer kopiert werden.
4. **Funktionsausführung durch den Server** — Der Server verarbeitet die Daten. Dabei dient der Opcode dazu die vom Client gewählte Funktion zu bestimmen.
5. **Vorbereiten des Nachrichtenpuffers** — Alle für den Client bestimmten Daten werden durch den Server im Nachrichtenpuffer abgelegt.
6. **Übertragung des Puffers an den Client** — Analog zu Punkt 2
7. **Auslesen des Nachrichtenpuffers** — Analog zu Punkt 3

Die wiederholte Erstellung des entsprechenden Programmcodes ist sowohl zeitraubend als auch monoton für den Programmierer und damit in besonderer Weise anfällig für Flüchtigkeitsfehler. Zusätzlich ist es erforderlich, geeignete Opcodes zu vergeben und zu gewährleisten, dass bei Opcodes sowie der Anordnung der Parameter im IPC-Puffer die Konsistenz zwischen Client- und Serverseite gewahrt bleibt. Aus diesem Grund haben sich IDL-Compiler etabliert. Mithilfe einer Schnittstellenbeschreibungssprache (*„interface definition language“*, IDL) spezifiziert der Nutzer detailliert die Schnittstelle des Servers, das heißt alle für andere Systemkomponenten bereitgestellten Funktionen. Zur Angabe der Daten, die zwischen Client und Server übertragen werden sollen, stehen hierbei die bekannten einfachen Typen (wie `int`, `long`, ...) direkt zur Verfügung. Darüber hinaus besteht in der Regel die Möglichkeit eigene Datentypen zu definieren.

Ein spezielles Programm, der IDL-Compiler, erzeugt aus dieser vollständigen Beschreibung anschließend eine Implementierung des nötigen Kommunikationscodes in der Zielprogrammiersprache. Dabei werden auch Codeteile (häufig *„stub“* oder *„skeleton“* genannt) erzeugt, mit deren Hilfe der Benutzer eigenen Code an den generierten anbindet; auf diese Weise wird zum einen die eigentliche Serverfunktionalität eingefügt und zum anderen den Clients ein einfacher Zugriff auf die Funktionen des Servers ermöglicht. Die gesamte Kommunikation zwischen Client und Server, die sonst aufwendig zu implementieren wäre, reduziert sich damit auf die reine Spezifikation der gewünschten Schnittstelle.

Einen fehlerfrei arbeitenden IDL-Compiler vorausgesetzt, nimmt die Produktivität zu, während gleichzeitig die Fehleranfälligkeit des fertigen Codes sinkt. Ein weiterer positiver Effekt besteht in der Abstraktion von Programmiersprache und Plattform, die vom erzeugten Kommunikationscode genutzt werden. Sofern der eingesetzte IDL-Compiler die neue Zielsprache- beziehungsweise Plattform unterstützt, kann eine Portierung mit geringem Aufwand durchgeführt werden.

2.2.1. DICE

Beim *„Dresden IDL Compiler“* (DICE) [dic, Aig01] handelt es sich um den IDL-Compiler des *„Dresden Real-Time Operating System“* (DROPS). Als Schnittstellenbeschreibungssprachen werden die weit verbreiteten DCE- und CORBA-IDL unterstützt. Die volle

Funktionalität setzt allerdings Attribute und damit die Verwendung von DCE-IDL voraus. Der Kommunikationscode wird in C oder C++ erzeugt und nutzt wahlweise Linux-Sockets oder den IPC-Mechanismus von Fiasco, dem nicht-Capability-basierten Vorgänger von Fiasco.OC.

DICE versucht die Laufzeitverlängerung (*„overhead“*) des erzeugten Codes zu minimieren, indem er Inline-Assemblercode einfügt und, wenn Fiasco verwendet wird, die verschiedenen verfügbaren IPC-Varianten möglichst optimal einsetzt. Benutzerdefinierte Datentypen können direkt aus vorhandenen C/C++-Quellcodedateien eingebunden werden; die wiederholte und damit sowohl aufwendig zu wartende wie auch potentiell inkonsistente Deklaration dieser Typen durch den Benutzer wird so vermieden.

Mithilfe zahlreicher Attribute innerhalb der IDL sowie Kommandozeilenparameter lässt sich die Codeerzeugung sehr genau steuern. Möglich ist unter anderem auch die Erzeugung von Kommunikationscode, der anstelle vollständiger RPCs lediglich Einwegnachrichten nutzt (*„message passing“*); der Absender der Nachricht erwartet hier keine Antwort vom Empfänger und arbeitet umgehend weiter. Im eingangs beschriebenen allgemeinen RPC-Ablauf entfallen somit die Punkte 5 bis 7 (Einwegnachricht vom Client an den Server) beziehungsweise Punkte 1 bis 3 (Einwegnachricht vom Server an den Client). Dieser Ansatz bietet in Szenarien, in denen Daten ohnehin ausschließlich in einer Richtung übertragen werden, große Vorteile bezüglich Komplexität und Laufzeit des erzeugten Kommunikationscodes.

DICE verwendet einen eigenen, auf Bison und Yacc basierenden Parser für IDL und eingebundene C/C++-Quelltexte. Für Schnittstellen ist eine einfache Form der Vererbung vorgesehen, bei der die erbende Schnittstelle alle Funktionen der vererbenden übernimmt. Auf diese Art ist es möglich, selbst komplexe Schnittstellen schnell, einfach und in überschaubarer Form zu spezifizieren. Dabei weist DICE jeder Schnittstelle eine Interface-ID und jeder Schnittstellenfunktion einen Opcode zu, um verschiedene Funktionsaufrufe innerhalb eines Servers auch bei Verwendung von Vererbung eindeutig unterscheiden zu können. Beide Kennungen können über die bereits erwähnten Attribute auch individuell vom Nutzer vorgegeben werden und identifizieren gemeinsam jede Funktion des Servers eindeutig.

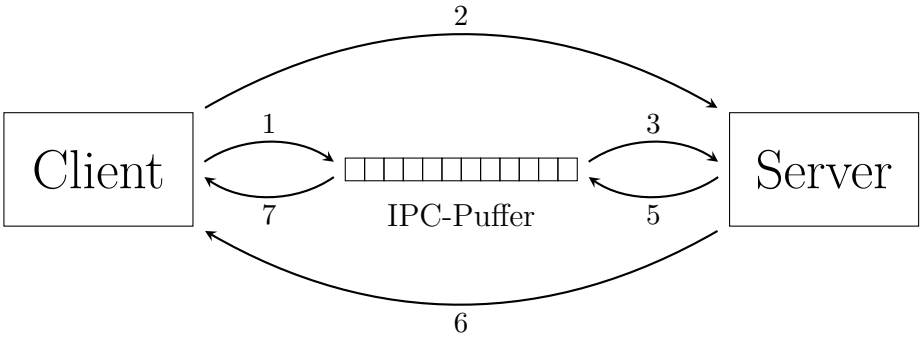


Abbildung 2.1.: Ablauf eines RPC – Die Ziffern beziehen sich auf die Beschreibung in Abschnitt 2.2 auf Seite 11.

2.2.2. IPC-Streams

Mit seinen IPC-Streams bietet L4Re eine Abstraktion der IPC-Funktionen des unterliegenden Fiasco.OC-Mikrokerns. Wie bereits der Name andeutet, arbeiten IPC-Streams analog zu den IO-Streams der C++-Standardbibliothek: Spezielle Klassen überladen die Verschiebeoperatoren `<<` und `>>` und verändern dadurch deren Semantik zu Ein- und Ausgabeoperatoren. Der Programmierer nutzt nun diese Operatoren um Daten in den IPC-Puffer zu schreiben beziehungsweise daraus zu lesen. Ebenfalls durch die IPC-Stream-Klassen bereitgestellte Methoden dienen zum Auslösen des eigentlichen IPC-Vorgangs, so dass eigene RPCs, einschließlich des zugehörigen Marshalling, damit umsetzbar sind.

Die IPC-Streams sind eine Umsetzung des *„dynamischen RPC-Marshalling“* [Fes07]. Für dieses Konzept spräche, so Feske in seiner Arbeit, dass einige zentrale Vorteile von IDL-Compilern – Entkopplung von Zielsprache und unterliegendem System, Behandlung komplexer Datentypen – bei Mikrokern-basierten Systemen aufgrund der systemnahen Programmierung, die diese erfordern, ohnehin kaum zum Tragen kämen. Darüber hinaus stelle die erforderliche Einarbeitung der Benutzer in eine ungewohnte Sprache (die IDL) einen erheblichen Aufwand und eine zusätzliche Quelle für Fehler dar. Die noch verbleibenden Vorzüge von IDL-Compilern – höherer Komfort für den Programmierer sowie automatische Optimierung des erzeugten Kommunikationscodes – wögen diese Nachteile und den zusätzlichen Codeeintrag in die TCB nicht auf.

Das eingangs beschriebene Nutzungsschema lässt das Hauptproblem der IPC-Streams bereits erahnen: Aufgrund der formal getrennte Implementierung von Client- und Server-Code kommt es leicht zu Fehlern. Sie entstehen, wenn Parameter in einer anderen Reihenfolge aus dem IPC-Puffer entnommen werden als derjenigen, in der sie eingefügt wurden. Eine inkonsistente Verwendung von Datentypen führt ebenfalls zu Problemen. Weder den IPC-Streams noch dem Compiler ist es möglich Fehler dieser Art zu erkennen und auch für den Programmierer sind sie bei späteren Tests nur schwer auffindbar. Durch die manuellen Vergabe und Verwendung von Opcodes können weitere Inkonsistenzen auftreten.

Weitere Nachteile ergeben sich aus der dynamischen Natur von IPC-Streams: Da die zu übertragenden Daten erst zur Laufzeit bekannt werden, müssen sie zwangsläufig in genau der Reihenfolge im IPC-Puffer abgelegt werden, in der sie vom Benutzer an den Stream übergeben werden. Dabei kann es dazu kommen, dass Füllbytes eingefügt werden müssen, um eine korrekte Ausrichtung (*„alignment“*) aller Daten zu garantieren. Die Füllbytes belegen Speicherplatz, der dann für Nutzdaten nicht mehr zur Verfügung steht. In Abschnitt 3.7 auf Seite 32 wird diese Problematik noch einmal aufgegriffen.

Um Arrays auf Empfängerseite wieder korrekt aus dem Puffer lesen zu können, muss deren jeweilige Länge bekannt sein. IPC-Streams legen deshalb die Länge eines Arrays unmittelbar vor den Datenelementen selbst im IPC-Puffer ab, wobei stets der Datentyp `unsigned long` Verwendung findet. Falls die Datenelemente des Arrays nicht ebenfalls von diesem Typ sind, steigt so sie Zahl der eingefügten Füllbytes weiter an. Auch die erst zur Laufzeit stattfindende Prüfung auf Pufferüberläufe und insbesondere der Verzicht auf Rückmeldungen im Falle eines Fehler („überhängende“ Daten werden stillschweigend ignoriert) können zu schwer nachvollziehbaren Probleme führen.

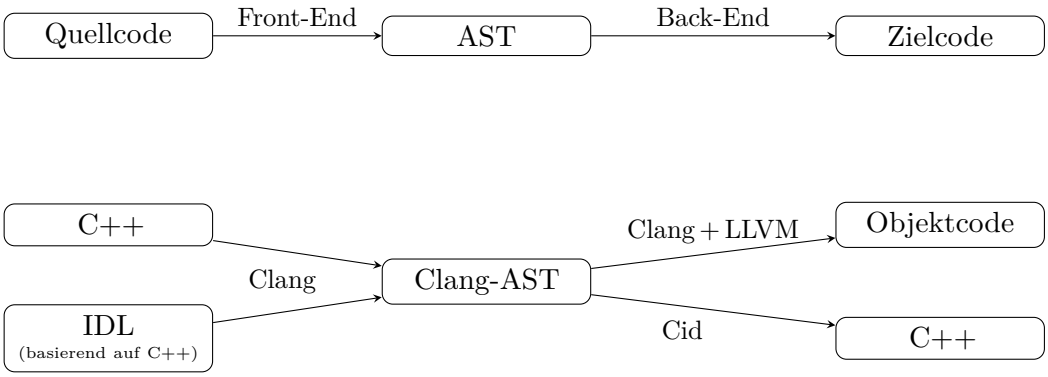


Abbildung 2.2.: Übersetzungsprozess

2.3. C++-Compiler

Compiler folgen in Organisation und Implementierung in der Regel einem Standardaufbau – sie untergliedern sich in eine Analyse (Front-End) und Synthese (Back-End). Im Zuge der Analysephase überführt der Compiler den Quelltext über verschiedene Verarbeitungsschritte in eine interne Repräsentation, die auch als abstrakter Syntaxbaum (*„abstract syntax tree“*, AST) bezeichnet wird. Aus dem AST wiederum entsteht in der Synthesephase dann das Endergebnis; im Fall eines C++-Compilers der Objektcode. Ein wesentlicher Vorteil dieses zweigliedrigen Aufbaus besteht darin, dass für zusätzliche Ausgangssprachen lediglich ein passendes Front-End und für neue Zielsprachen nur ein Back-End zu implementieren ist. Der Aufwand für die Unterstützung zusätzlicher Sprachen wird gegenüber der Neuentwicklung eines vollständigen Compilers also wesentlich reduziert.

Während sich das Back-End eines bestehenden C++-Compilers wegen der grundverschiedenen Zielsprachen offensichtlich nicht zur Wiederverwendung in einem IDL-Compiler eignet, liegt der Fall beim Front-End anders. Ähneln sich die Ausgangssprachen genügend stark, läuft die Analysephase praktisch identisch ab und der vorhandene Code kann, in entsprechend angepasster Form, auch im neuen Compiler genutzt werden. Als Grundlage der geplanten Entwicklung eines IDL-Compilers für L4Re bietet sich C++ besonders an:

- C++ stellt mit Namensräumen, Klassen und deren Methoden ein Grundkontingent an Sprachelementen zur Verfügung, die geeignet sind, Schnittstellen zwischen verschiedenen Programmen zu spezifizieren.

- L4Re selbst ist größtenteils in C++ implementiert, das heißt die Entwickler sind mit der Sprache bestens vertraut. Eine eng daran angelehnte IDL wäre für sie folglich leicht erlernbar.

- Basiert die IDL auf C++, so kann bei der Implementierung des IDL-Compilers das Front-End eines vorhanden C++-Compilers verwendet werden; wie in Abbildung 2.2 auf dieser Seite veranschaulicht. Die aufwendige und fehleranfällige Überführung

einer IDL-Spezifikation in eine einfach handhabbare Repräsentation (den AST) muss so nicht neu entwickelt werden.

Die beiden C++-Compiler GCC und Clang bieten sich für eine praktische Umsetzung besonders an. Sie unterstützen weitestgehend den C++98-Standard und können so auch bei der Übersetzung von L4Re selbst zum Einsatz kommen. Des Weiteren stehen beide unter freien Lizenzen, die insbesondere die Arbeit mit dem Quellcode der Compiler, sowie Erstellung und Verbreitung darauf aufbauender Software gestatten.

2.3.1. GCC

Über viele Jahre hinweg ist GCC (*„GNU Compiler Collection“*) [gcca] von einem einfachen C-Compiler auf seine heutige, multifunktionalen Form angewachsen, die seit geraumer Zeit auch einen C++-Compiler (`g++`) umfasst. Die Codebasis ist historisch gewachsen und besteht nahezu vollständig aus C sowie einer Vielzahl an Präprozessor-Makros.

Der vom C++-Front-End erzeugte AST (von den Entwicklern schlicht als *„tree“* bezeichnet) setzt sich zusammen aus C++-spezifischen Elementen sowie einer *„GENERIC“* genannten, von der Ausgangssprache unabhängigen Repräsentation. Leider weist die dazu vorhandene Dokumentation erhebliche Lücken auf, die eine Einarbeitung stark erschweren [gccb]:

> „There are many places in which this document is incomplet and incorrekt[sic]. It is, as of yet, only preliminary documentation.“

> „If you are developing a ‘back end’, [...], that uses this representation, you may occasionally find that you need to ask questions not easily answered by the functions and macros available here. If that situation occurs, it is quite likely that GCC already supports the functionality you desire, but that the interface is simply not documented here. In that case, you should ask the GCC maintainers (via mail to gcc@gcc.gnu.org) about documenting the functionality you require.“

Im Hinblick auf die Ziele der vorliegenden Arbeit ist die Möglichkeit GCC mittels Plug-ins zu erweitern von besonderem Interesse. Diese setzen an ausgewählten Stellen des Übersetzungsprozesses an und erlauben es, vorkompilierte GCC-Binärprogramme – ein Standardbestandteil der meisten GNU/Linux-Distributionen – einfach um zusätzliche Funktionalität zu ergänzen. Das ist nicht nur für den Benutzer von Vorteil, der so keinen zusätzlichen, vollständigen Compiler auf seinem System einrichten und pflegen muss. Auch der Entwicklungsaufwand reduziert sich, da kein vollständiges Programm implementiert wird, sondern (neben den wenigen von GCC vordefinierten Plug-in-Schnittstellen) ausschließlich die benötigten Zusatzfunktionen.

2.3.2. Clang

LLVM (*„Low Level Virtual Machine“*) [llva] ist ein reines, hochoptimierendes Compiler-Back-End, das ursprünglich als Forschungsprojekt der Universität von Illinois entwickelt

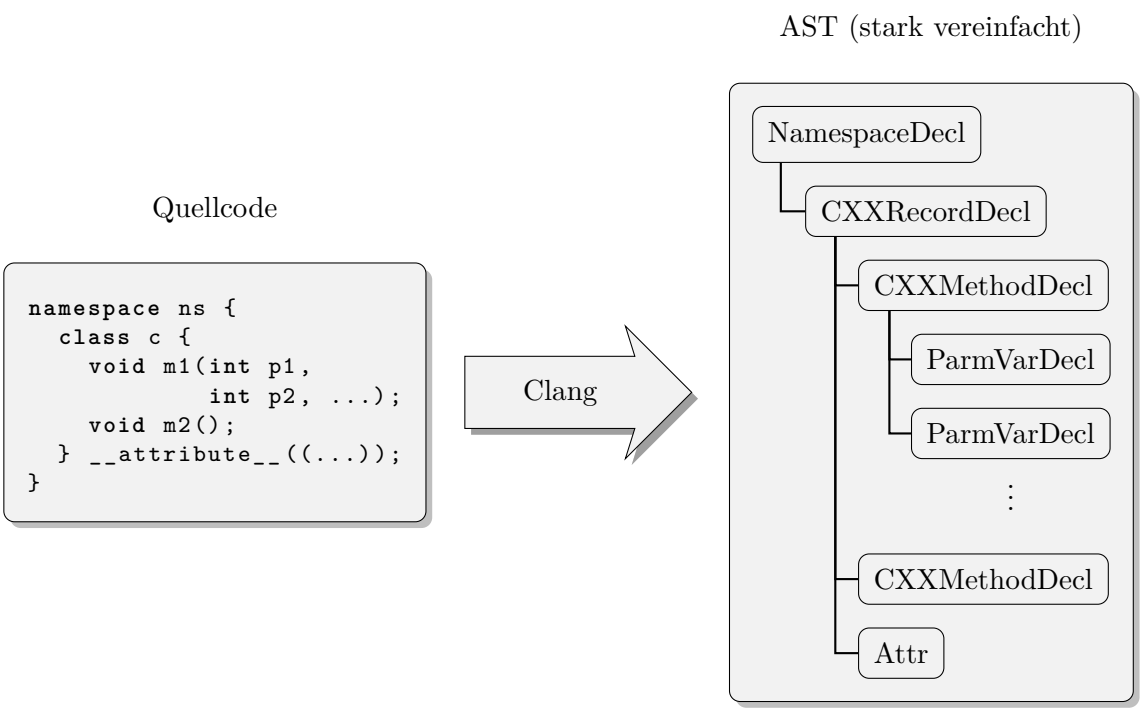


Abbildung 2.3.: AST-Repräsentation einer C++-Deklaration durch Clang

und später unter eine freie Lizenz gestellt wurde. Verschiedene Front-Ends erzeugen einen speziellen maschinen- und sprachunabhängigen Zwischencode (*„intermediate form“*), der dann von LLVM für Optimierungen und die letztendliche Codeerzeugung genutzt wird.

Nachdem zunächst GCC-Front-Ends angepasst und genutzt wurden, kam im Laufe der Zeit Clang als eigenständig entwickeltes Front-End für die Sprachen C, C++ und Objective-C hinzu, die in der aktuellen Version 2.9 praktisch vollständig unterstützt werden. Nach dem Vorbild von LLVM ist Clang von Grund auf in C++ und mit dem Ziel der Modularität entwickelt worden. Praktisch die gesamte Funktionalität ist über Bibliotheken für die Nutzung in anderen Projekten zugänglich. Zusammen mit einer umfassenden Dokumentation und einer (derzeit noch experimentellen) Plug-in-Schnittstelle ermöglicht dies einen verhältnismäßig einfachen Einstieg in die Nutzung der Codebasis [claa, clab]. Weitere Aspekte, wie effiziente Ressourcennutzung und benutzerfreundliche Diagnosemeldungen, gaben schließlich den Ausschlag, Clang als Basis für den im Rahmen dieser Arbeit entwickelten IDL-Compiler auszuwählen. Diese Herkunft findet auch im Namen *„Cid“* (kurz für *„**C**lang-basierter **ID**L-Compiler“*) ihren Niederschlag.

AST Clang bildet den AST fertig analysierten Codes durch eine Klassenhierarchie ab. Jedes auftretende C++-Sprachelement wird durch eine Instanz einer spezifischen Klasse repräsentiert; siehe dazu auch Abbildung 2.3 auf dieser Seite. Die AST-Struktur spiegelt dabei den ursprünglichen Quellcode ungewöhnlich detailliert und umfassend wider; Typinformationen sowie Details zum Aufbau der Quelldateien bleiben darin erhalten. Unter anderem referenziert jede Deklaration die zugehörige Quellcodezeile, für

Präprozessormakros sind sowohl Definition als auch die jeweilige Instanziierung abrufbar und selbst die Einbindung anderer Dateien durch den Präprozessor (`#include`) kann nachvollzogen werden. Von Clang werden diese erweiterten Information vor allem für die Ausgabe sehr ausführlicher Fehlermeldungen verwendet.

Nach der Erstellung des AST sind Veränderungen an dessen Bestandteilen nicht mehr möglich; eine Entscheidung, die von den Clang-Entwicklern damit begründet wird, dass bei Änderungen viele, teils indirekte Wechselwirkungen zu berücksichtigen sind und sich aus der Unveränderlichkeit Vorteile bei der Serialisierung des AST ergeben [clac].

2.3.3. GNU-Attribute

GNU-Attribute [gccb] waren ursprünglich eine GCC-eigene Erweiterung, die folglich nicht vom C++-Standard abgedeckt sind. Dennoch werden sie heute von sehr vielen Compilern unterstützt; so auch von Clang. Im Programmcode werden GNU-Attribute dem Sprachelement beigefügt, das sie semantisch beeinflussen und dienen primär dazu, dem Compiler zusätzliche Informationen zum übersetzten Programm zu vermitteln. Mit ihrer Hilfe werden Funktionen als „speziell“ gekennzeichnet (beispielsweise `printf`-ähnlich, häufig/selten/niemals aufgerufen, nicht zum Aufrufer zurückkehrend, ...), die Ausrichtung von Variablen im Speicher genau vorgegeben und vieles mehr. Diese Metainformationen erlauben dem Compiler weitergehende Prüfungen und Optimierungen oder weisen ihn explizit an, bestimmten Code zu erzeugen, der den jeweiligen Anforderungen entspricht.

3. Entwurf

Dieses Kapitel beleuchtet einige theoretische Aspekte des entwickelten IDL-Compilers, genauer: die von Cid unterstützte IDL und ihre Entstehung aus der Sprache C++, die Einbindung von Clang, den Umgang mit fehlerhaften Eingaben durch den Benutzer sowie die verschiedenen Möglichkeiten zur Codeerzeugung und -optimierung. Für einige ausgewählte Fälle werden dabei verschiedene, in Erwägung gezogene Alternativen vorgestellt und die letztlich getroffenen Entscheidungen begründet.

3.1. Schnittstellenbeschreibungssprache/IDL

Die unterstützten Quell- und Zielsprachen sind von elementarer Bedeutung für jeden Compiler. Da Cid selbst auf einem C++-Compiler aufbaut, basiert die verarbeitete IDL ebenfalls auf einer Teilmenge von C++. Diese wird ergänzt um spezielle GNU-Attribute, die im Weiteren auch als *„Cid-Attribute“* bezeichnet werden. Sprachschatz und -struktur orientieren sich an der von DICE genutzten, erweiterten DCE-IDL (siehe Abschnitt 2.2.1). Zum einen hat sie sich beim Einsatz von DICE nun über mehrere Jahre bewährt, zum anderen wird Nutzern so der Umstieg beziehungsweise die parallele Nutzung beider IDL-Compiler erleichtert.

3.1.1. C++ als Grundlage

Die Sprache C++ eignet sich hervorragend als Basis einer IDL:

- Die Schnittstelle eines Server wird vom Konzept der C++-Klasse perfekt abgebildet. Die Methoden der Klasse treten dabei an die Stelle der einzelnen Funktionen der Schnittstelle.

- Mit der nativen Unterstützung für die Vererbung von C++-Klassen steht eine bequeme Möglichkeit zur Erweiterung bereit; vorhandene Schnittstellen können kombiniert und bei Bedarf um zusätzliche Funktionen ergänzt werden, um so neue, komplexere Schnittstellen zu schaffen.

- Zur Definition eigener Datentypen stehen dem Benutzer alle Möglichkeiten offen, die ihm bereits aus C++ vertraut sind (`enum`, `struct`/`class`, `typedef`, ...). Die geschickte Einbindung des C++-Präprozessors in die IDL erleichtert diese Arbeit. Sie ermöglicht es dem Benutzer Typdeklarationen aus bereits vorhandenem Code – beispielsweise den Header-Dateien von L4Re – direkt innerhalb einer IDL-Spezifikation zu verwenden. Das spart nicht nur Aufwand, sondern beseitigt gleichzeitig die Gefahr inkonsistenter Typdeklarationen in IDL-Spezifikation und Programmcode.

Erwartungsgemäß wird ein Großteil der Möglichkeiten von C++ nicht benötigt. Im Rahmen einer IDL wirkt er sich im ungünstigsten Fall sogar kontraproduktiv aus. So sind beispielsweise eingeschränkte Sichtbarkeiten (`private` beziehungsweise `protected`) für eine „IDL-Klasse“/Schnittstelle nicht sinnvoll – die Intention einer IDL ist schließlich gerade die Spezifikation der öffentlichen, von anderen Programmen zu nutzenden Schnittstelle eines Servers. Benötigt werden ausschließlich die deklarativen Sprachelemente von C++. Eine Ausnahme hiervon stellen einzig und allein Konstanten- und `enum`-Definitionen dar, auf die bestimmte Cid-Attribute zurückgreifen können [Bie].

An anderer Stelle mangelt es C++ dagegen an Möglichkeiten Informationen abzubilden, die für eine vollwertige IDL von essentieller Bedeutung sind. Ein einfaches Beispiel, das die Notwendigkeit von Metainformationen vor Augen führt, ist die Übertragungsrichtung einzelner Funktionsparameter (siehe dazu Abschnitt 2.2). Natürlich kann diese Lücke mittels Heuristiken[1] oder durch eine pauschale Übertragung jedes Parameters in beide Richtungen ausgefüllt werden. Die Nutzung von Heuristiken ist jedoch zwangsläufig fehleranfällig und für den Benutzer schlecht nachvollziehbar, während mangelnde Differenzierung unnötigen Übertragungen und damit einhergehenden Laufzeitverschlechterungen zur Folge hat. Beide Optionen scheiden daher aus.

Zu Beginn der Entwicklung wurde kurzzeitig der Ansatz verfolgt, die Teilmenge von C++, die nicht für die konzipierte IDL verwendet wird, zur Abbildung derartiger Metainformationen zu nutzen. Die willkürlich veränderte Semantik vertrauter Sprachbestandteile erwies sich als verwirrend für den Benutzer und damit als sehr fehleranfällig. Der Ansatz wurde daher rasch wieder verworfen. Darüber hinaus ist diese Lösung nicht in allen Situation praktikabel. Die Sichtbarkeiten innerhalb von C++-Klassen eignen sich zwar gut zur Abbildung der oben angesprochene Übertragungsrichtung, doch bereits die Verknüpfung von Arrays mit ihren Längenparametern, auf die im Folgenden noch detailliert eingegangen wird, bereitet größere Probleme. Zur erweiterten Beschreibung der verschiedenen Sprachelemente der IDL wurden stattdessen zahlreiche Cid-Attribute eingeführt; Abschnitt 3.4 auf Seite 26 wird sich ausführlich mit diesen befassen.

3.1.2. Überblick

Ursprünglich war für die IDL eine vollständige Einbettung in beliebigen C++-Code angedacht. Alle nicht mit Cid-Attributen versehenen Sprachelemente sollten dabei unverändert in den generierten Code übernommen werden und hätten so neben der reinen Schnittstellenbeschreibung auch die direkte Implementierung von Server-Funktionen durch den Nutzer erlaubt. Leider stellte sich diese Version schnell als zu komplex heraus: Im Rahmen eines IDL-Compilerlaufs wäre die Behandlung des vollständigen C++-Sprachumfang nötig geworden; ein unverhältnismäßig großer Mehraufwand angesichts des relativ geringen Mehrwerts.

Aus diesem Grund folgt die letztendlich umgesetzte Form der IDL [Bie] einem Mittelweg: Mit Cid-Attributen versehene, anonyme Namensräume (*„Sektionen“*) unterteilen jede IDL-Datei in eigentliche IDL-Spezifikation (*„IDL-Sektion“*) und beliebigen, ergänzenden C++-Code (*„Kopiersektion“*). Dabei weist ein Cid-Attribut den Typ der

[1] Ein möglicher, sehr primitiver Ansatz: Übertragung einfacher Datentypen vom Client an den Server, Referenzen und Zeiger (beziehungsweise der Daten auf die diese verweisen) in der Gegenrichtung.

jeweiligen Sektion eindeutig aus. Wie der Name bereits andeutet, werden die Inhalte von Kopiersektionen von Cid nicht näher analysiert, sondern unverändert in den erzeugten Quelltext übertragen. Ihr Zweck besteht vor allem in der Definition eigener Datentypen. Sie können aber allgemein zur Bereitstellung von beliebigem, benutzereigenen C++-Code verwendet werden, der später sowohl für den Client wie auch für den Server bereitstehen soll.

Die Wahl fiel an dieser Stelle auf anonyme Namensräume, da sie sich im Rahmen der Sprache C++ ausschließlich auf die Bindung (*„linkage“*) darin enthaltener Deklarationen auswirken. Darüber hinaus verhalten sie sich semantisch „transparent“; eine sehr wünschenswerte Eigenschaft für die ihnen zugedachte Funktion als Sektionen einer IDL-Datei. Im Verlauf der Entwicklung traten dann jedoch Komplikationen auf: Wichtige Teile des L4Re-Codes definieren eigene Operatoren, darunter auch `new` und `delete`. Der C++-Standard untersagt allerdings deren Definition innerhalb von Namensräumen [ISO03]. Dennoch sollen L4-Header natürlich in IDL-Dateien eingebunden werden können, um Typdeklarationen oder Konstantendefinitionen bereitzustellen. Das ursprüngliche Schema musste also erweitert werden: Die Nutzung der `#include`-Präprozessordirektive bleibt auch außerhalb der oben vorgestellten Sektionen zulässig. In Abhängigkeit von ihrer Dateiendung wird der Inhalt so eingebundener externer Dateien semantisch behandelt als trete er in der IDL-Datei (Endung `.idl`) beziehungsweise in einer Kopiersektionen (beliebige andere Endung) auf.

Innerhalb von IDL-Sektionen dienen C++-Klassen zur Spezifikation von Schnittstellen; Methoden repräsentieren dabei die vom Server angebotenen Funktionen. Die von C++ gewohnte Vererbung bleibt in vereinfachter Form verfügbar und erlaubt die einfache Konstruktion einer Schnittstelle aus bereits vorhandenen. Dabei stehen alle Funktionen der Elternschnittstellen in der abgeleiteten Schnittstelle ebenfalls zur Verfügung und können bei Bedarf noch um weitere ergänzt werden. Alle Vererbungen werden von Cid als öffentlich betrachtet, ein Überschreiben von Methoden ist nicht vorgesehen. Auf diese Weise bleibt die vollständige Kompatibilität abgeleiteter Schnittstellen zu ihren Eltern erhalten und ein erwartungskonformes Verhalten der dahinterstehenden Server im Sinne des Substitutionsprinzips gewährleistet. Methodenüberladungen hingegen sind (auch über Vererbungshierarchien hinweg) zulässig.

Protokoll-ID und Opcode Protokoll-ID und Opcode stehen in enger Beziehung zur Umsetzung von RPC unter L4Re. Gemeinsam ermöglichen sie eine eindeutige Identifizierung aller durch einen Server bereitgestellten Funktionen – siehe auch die Abschnitte 2.1.3 und 2.2. Die IDL sieht dabei vor, jeder IDL-Klasse genau eine Protokoll-ID zuzuordnen. Mittels Vererbung ist es dennoch möglich, Schnittstellen zu schaffen, die mehrere Protokolle in sich vereinen.

Für beide Kennungen bietet Cid die Möglichkeit der manuellen oder automatischen Vergabe [Bie]; eine Mischung beider Verfahren (manuelle Vergabe nur für ausgewählte Protokolle/ IDL-Funktionen) wird ebenfalls unterstützt. Die manuelle Vergabe ist vor allem dann von Bedeutung, wenn etablierte Protokolle, wie zum Beispiel die Systemprotokollen von L4Re, modelliert werden sollen. Sie wird durch die Cid-Attribute `PROTO` und `OPCODE` umgesetzt, die IDL-Klassen beziehungsweise IDL-Funktionen beigefügt werden

können. In der Regel jedoch sind die zugewiesenen IDs für den Programmierer nicht von Interesse und Vergabe erfolgt automatisch:

- Ist die betrachtete IDL-Klasse von keiner anderen abgeleitet und wurde auch manuell wurde keine Protokoll-ID festgelegt, so wird Null als Standardwert eingesetzt. Erbt die IDL-Klasse dagegen von einer oder mehreren anderen, so übernimmt sie auch deren Protokoll-ID soweit diese für alle Elternklassen identisch ist. Ist die ID nicht für alle Eltern die gleiche, so bricht die Übersetzung mit einem Fehler ab. In einem solchen Konfliktfall ist es die Aufgabe des Nutzers durch manuelle Festlegung der Protokoll-ID eine eindeutigen Entscheidung zu treffen: Verwendung einer der ererbten Protokoll-IDs und damit die Erweiterung des zugehörige Protokolls um zusätzliche Funktionen oder Verwendung einer bisher ungenutzen ID und damit eines neuen Protokolls.

- Opcodes werden, beginnend mit Null, fortlaufend an alle IDL-Funktionen eines jeden Protokolls vergeben. Sollten einzelne Werte dabei schon belegt sein – wegen einer manuellen Zuweisung durch den Nutzer oder weil eine Elternschnittstelle sie bereits verwendet – werden diese übergangen, um unzulässige Doppelbelegungen auszuschließen.

Zeiger und Referenzen Ihrer besonderen Eigenschaften wegen ist es nicht sinnvoll den Inhalt von Zeigervariablen im Rahmen einer IPC unverändert zu übertragen. Nach dem Übergang in einen anderen Adressraum verweist der Zeiger nicht länger auf die ursprünglichen Daten, sondern in aller Regel auf unvorhersehbare Speicherinhalte. Vor diesem Hintergrund sind Zeiger in der Cid-IDL zwar verfügbar, dienen aber (wie auch Referenzen) vorrangig dazu, Parameter von IDL-Funktionen zu markieren, die vom Server an den Client übertragen werden. Anstelle des Zeigers übermittelt der erzeugte Kommunikationscode dabei grundsätzlich die referenzierten Daten selbst.

Arrays Für Arrays sind zusätzliche Informationen über ihre Länge – sowohl maximal als auch während der Ausführung einer konkreten IPC – erforderlich. Diese Angaben werden für die Überprüfung auf Überläufe sowie für die Erzeugung von Puffern zwingend benötigt und durch die Cid-Attribute `MAX` (maximale Länge) und `LEN` (aktuelle Länge) innerhalb der IDL-Spezifikation festgelegt. Je nach verwendetem `LEN`-Attribut unterscheiden man Arrays mit statisch spezifizierter Länge und Arrays mit dynamisch spezifizierter Länge: Während `MAX` die Übergabe eines konkreten Zahlwertes oder einer zuvor definierten Konstanten verlangt, gestattet `LEN` daneben noch den Verweis auf einen anderen Parameter der IDL-Funktion, die auch das Array selbst umfasst. Im letztgenannten Fall handelt es sich um ein Array mit dynamisch spezifizierter Länge – der mittels `LEN` assoziierte Parameter enthält zur Laufzeit die aktuelle Länge des Arrays. Die Verwendung eines konkreten Zahlenwertes für `LEN` (direkt oder indirekt über den Verweis auf eine Konstante) dagegen führt zu einem Array mit statisch spezifizierter und somit stets gleicher Länge. Dieser Unterschied spielt im Rahmen der Codeerzeugung eine wichtige Rolle, wie in den Abschnitte 3.6.2 und 3.7 auf Seite 29 und auf Seite 30 noch ausgeführt wird.

Darüber hinaus bietet die Cid-IDL die Möglichkeit, mit dem STRING-Attribut ein Array vom Typ `char` explizit als C-Zeichenkette auszuzeichnen. Für Parameter dieser Art werden weitere Mechanismen aktiv. So enthält der erzeugte Marshalling-Code beispielsweise Prüfungen auf eine korrekte Nullterminierung von `STRING`-Parametern.

Capabilitys Capability-Parameter sind ein weiterer Spezialfall und müssen mit dem `CAP`-Attribut explizit als Capability ausgezeichnet werden. Andernfalls behandelt Cid sie wie gewöhnliche nummerische Datenwerte. Diese Kennzeichnung durch den Benutzer wurde bewusst einer automatischen Erkennung allein anhand des Datentyps vorgezogen – Abschnitt 3.3 auf Seite 25 befasst sich ausführlicher mit diesem Grundsatz, möglichst wenige implizite Annahmen zu treffen. Fehlt einem Parameter das `CAP`-Attribut, obwohl er einen für Capabilitys typischen Datentyp (`l4_cap_idx_t` oder `L4::Cap`) besitzt, reagiert Cid mit einer Warnung, um den Nutzer so auf den möglichen Flüchtigkeitsfehler aufmerksam zu machen.

Rückgabewerte IDL-Funktionen verfügen, wie gewöhnliche C++-Funktionen, über einen optionalen Rückgabewert mit frei wählbarem Typ. Im Rahmen des Marshalling wird dieser behandelt wie jeder andere Parameter, der vom Server an den Client übertragen wird. Sofern keine Fehler auftreten stellt sich ein RPC bei Verwendung des erzeugten Kommunikationscodes daher wie ein gewöhnlicher lokaler Methodenaufruf dar.

Nicht immer wird dieses gewohnte Nutzungsverhalten durch IDL-Compiler unterstützt, da der clientseitige Rückgabewert oft zur Signalisierung von Übertragungsfehlern beziehungsweise allgemein des Status der im Hintergrund ausgeführten IPC verwendet wird. Zu diesem Zweck nutzt Cid stattdessen C++-Ausnahmen. Diese werden ausgelöst, um den vom Benutzer geschriebenen Code darüber zu unterrichten, falls ein Fehler aufgetreten ist. Nähere Informationen lassen sich dann dem geworfenen Objekt entnehmen.

Vererbung von Cid-Attributen In den meisten IDL-Spezifikationen treten bestimmte Cid-Attribute außerordentlich häufig auf; ein typisches Beispiel ist die Festlegung der Übertragungsrichtung eines Parameters (Cid-Attribute `IN`, `OUT`, `INOUT`). Für ausgewählte Cid-Attribute wurde daher die Möglichkeit einer „Attributvererbung“ geschaffen, um so die Nutzung der IDL zu erleichtern und Flüchtigkeitsfehlern vorzubeugen [Bie].

Vererbbare Attribute können nicht nur direkt dem semantisch betroffenen Sprachelement beigefügt werden, sondern auch solchen, die diesem im AST übergeordnet sind. Das Attribut wirkt sich dann auf alle „eingeschlossenen“ Elemente aus. Widersprüchliche Spezifikationen durch inkonsistente Attributausprägungen auf verschiedenen Ebenen sind ausgeschlossen, da näher am betroffenen Sprachelement stehende Attribute weiter entfernte überdecken – Abbildung 3.1 auf der nächsten Seite illustriert diesen Zusammenhang. Für die eingangs erwähnte Spezifikation der Übertragungsrichtung genügt somit ein einziges Cid-Attribut, das der IDL-Klasse beigefügt wird und für alle Parameter sämtlicher darin enthaltener IDL-Funktionen wirksam ist.

Schematisch dargestellt ist ein Cid-Attribut, das sich auf IDL-Funktionen bezieht. Es ist darüber hinaus vererbbar und kann aus diesem Grund auch Schnittstellen und Sektionen beigefügt werden.
Jede geometrische Figur/Farbe repräsentiert eine mögliche Ausprägung des Cid-Attribut: die ausgefüllte Form steht dabei für die explizite Verwendung des Attributes an dieser Stelle, der Umriss für geerbte Werte.

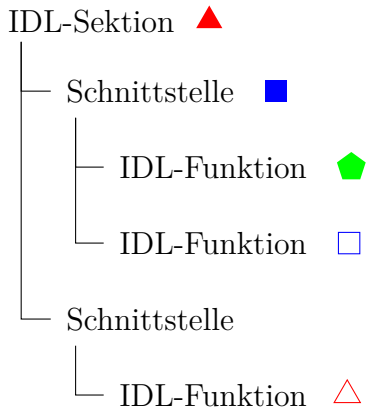


Abbildung 3.1.: Vererbung von Cid-Attributen

3.2. Zugriff auf den AST

Zur Nutzung des AST, den Clang aus dem Quelltext erzeugt (siehe Abschnitt 2.3.2), bestehen zwei grundsätzliche Alternativen. Zum einen kann Clang mithilfe von Kommandozeilenargumenten, wie zum Beispiel `-ast-dump`, angewiesen werden den generierten AST auszugeben beziehungsweise in einer Datei abzulegen. Vor einer Weiterverarbeitung durch Cid wäre dann ein Neuparsen des ausgegebenen AST erforderlich – ein großer Aufwand und gravierender Nachteil dieser Option. Zum anderen besteht die Möglichkeit einer direkten Integration von Clang. In diesem Fall ist der AST in Form von C++-Objekten im Speicher zugänglich und kann direkt verwendet werden. Dieser Direktzugriff ist wiederum auf zwei Arten möglich:

Eigenständiges Programm Die Funktionalität von LLVM/Clang steht weitestgehend in Form von Bibliotheken für eigene Projekte zur Verfügung. Durch die konsequente Verwendung dieser Bibliotheken ist es möglich, einen vollwertigen und eigenständigen IDL-Compilers mit verhältnismäßig geringem Aufwand zu entwickeln.

Plug-in Neuere Versionen von Clang verfügen über eine Erweiterungsschnittstelle, mit deren Hilfe zusätzlicher Code nachgeladen und anstelle der üblichen Objektcode-Erzeugung ausgeführt werden kann. Ein bereits vorhandenes, unverändertes Clang-Binärprogramm wird so mitgenutzt und zu einem IDL-Compiler erweitert.

In beiden Fällen liegen die Analyse der Eingabedateien sowie die Erstellung des AST vollständig in der Hand von Clang beziehungsweise der entsprechenden Bibliotheken. Auch zahlreiche weitere Bestandteile des C++-Compilers stehen zur Verfügung; beispielsweise dessen hocheffiziente Verwaltungsdatenstrukturen und Parsing-Mechanismen für Kommandozeilenparameter. Abschnitt 4.1 auf Seite 37 wird auf diesen Aspekt noch näher eingehen.

Insbesondere wegen anfänglicher Schwierigkeiten mit dem noch experimentellen Plug-in-Mechanismus konzentrierte sich die Entwicklung zunächst stark auf die Variante des eigenständigen Programms. Nachdem diese Probleme aber überwunden waren, erwies sich das Plug-in-Konzept für die Entwicklungsarbeit als sehr vorteilhaft: Die direkte Mitnutzung eines existierenden Clang-Binärprogramms erspart die Einbindung umfangreicher Bibliotheken bei der Erstellung des Plug-ins, reduziert damit die benötigte Zeit und beschleunigt Testläufe so enorm.

Die Erweiterungsschnittstelle von Clang ist noch immer als experimentell anzusehen, deshalb bleiben manuelle Anpassungen am Quellcode bis auf Weiteres erforderlich, um Plug-ins verwenden zu können. Aufgrund dieser Einschränkung fiel die Entscheidung, beide Ansätze weiterzuentwickeln und zusammenzuführen. Das Hauptaugenmerk von Entwicklung und Wartung liegt auf der Plug-in-Variante. Zusammen mit einem einfach gehaltenen Starter (hauptsächlich bestehend aus dem leicht angepassten Startcode von Clang selbst) sowie den LLVM-/Clang-Bibliotheken kann alternativ ein eigenständig lauffähiges Programm erzeugt werden – praktisch eine erweiterte Version von Clang mit integriertem Cid-Plug-in. Beide Programmversionen unterscheiden sich bewusst nicht in der Bedienung[2], um einerseits den beliebigen Wechsel zwischen beiden Varianten zu erleichtern und andererseits den Starter einfach zu halten. Die so erreichte Kapselung sorgt dafür, dass im Wesentlichen nur die Plug-in-Version von Cid weiterentwickelt und gewartet werden muss und hält so den Aufwand in Grenzen.

3.3. Fehlerbehandlung

Wie bei jedem Programm, das Benutzereingaben verarbeitet, muss auch bei einem IDL-Compiler mit Fehlern in den Eingabedaten gerechnet werden. Zwar ist die eingesetzte IDL bewusst einfach gehalten, dennoch bleibt sie hinreichend komplex, um Raum für fehlerhafte IDL-Spezifikationen zu lassen.

Die meisten Compiler bieten Mechanismen, die Fehler im Quellcode kompensieren und dem Benutzer dadurch die Arbeit zu erleichtern – so auch GCC und Clang. Zwar werden entsprechende Fehlermeldungen ausgegeben, der Übersetzungsprozess läuft jedoch soweit wie möglich weiter. Je nach Art des Fehlers kommen dabei Standardwerte zum Einsatz oder der fehlerhafte Code wird schlicht von der Übersetzung ausgenommen. Zwangsläufig bergen Hilfsmechanismen der beschriebenen Art auch Nachteile: Während implizite Korrekturen helfen, den Übersetzungslauf trotz Fehlern fortzusetzen, können sie ebenso zu schwer nachvollziehbaren und für den Benutzer irritierenden Folgefehlern führen. Nicht unberücksichtigt bleiben sollte auch der zusätzliche Aufwand, der zur zuverlässigen Weiterverarbeitung von fehlerbehaftetem IDL-Code notwendig ist.

Aus diesen Gründen arbeitet Cid gemäß dem Prinzip des schnellen Abbruches (*„fail fast“*) – bereits beim ersten auftretenden Fehler wird der Übersetzungsvorgang (nach Ausgabe einer möglichst detaillierten Meldung) abgebrochen. Auf unklare oder widersinnige Situationen reagiert Cid mit einer Warnung, setzt seine Arbeit aber fort. Ein

[2] Auch bei der eigenständigen Version muss der Benutzer Clang explizit anweisen, das eingebettete Cid-Plug-in auszuführen. Auch die Übergabe von Kommandozeilenargumenten erfolgt analog zur Plug-in-Version.

typisches Beispiel dafür sind Parameter mit Capability-typischem Datentyp aber ohne entsprechenden Cid-Attribut; siehe dazu auch Abschnitt 3.1.2.

Das von Cid vorgesehene Standardverhalten des generierten Codes ist so gewählt, dass ein reibungsloser Ablauf sichergestellt wird. Nötigenfalls werden dafür auch Leistungseinbußen hingenommen. Der Benutzer kann den IDL-Compiler durch Verwendung optionaler Cid-Attribute anweisen, von diesen Vorgaben abzuweichen und erweiterte Optimierungen durchzuführen, auf die Abschnitt 3.7 auf Seite 33 noch detailliert eingehen wird.

3.4. Umsetzung der Cid-Attribute

Wie in Abschnitt 3.1.1 bereits erläutert, genügt der Sprachschatz von C++, der in die Cid-IDL übernommen wurde, allein nicht, um eine vollwertige Beschreibungssprache zu schaffen. Vielmehr sind zusätzliche Metainformationen notwendig, zu deren Abbildung die IDL Cid-Attribute vorsieht. Die verschiedenen Umsetzungen, die im Verlauf der Entwicklung für diese Attribute in Betracht gezogen wurden, werden im Folgenden kurz diskutiert. Als Beispiel dienen dabei die beiden Attribute zur Spezifizierung der Längen eines Arrays (`MAX` und `LEN`), die in Abschnitt 3.1.2 vorgestellt wurden.

Benutzerdefinierte GNU-Attribute Die naheliegendste Lösung – die Einführung neuer, selbstdefinierter GNU-Attribute – erwies sich als impraktikabel, da Clang derzeit keine einfache Möglichkeit zur Definition eigener GNU-Attribute bietet. Stattdessen sind direkte Änderungen an den Quelltexten des C++-Compilers notwendig; für ein Plug-in schlichtweg unmöglich, für die eigenständige Variante zu aufwendig. Auch die fortlaufend nötige Anpassung an Änderungen in neuen Clang-Versionen macht diesen Ansatz unattraktiv.

Kommentare Einfache C-Kommentare (`/* ...*/`) in unmittelbarer Nähe der betroffenen Sprachelemente sind ebenfalls eine praktikable Option zur Umsetzung von Cid-Attributen. In ersten Prototypen zeigte sich aber schnell, dass Clang Kommentare bereits während der Analyse entfernt und daher nicht im AST abbildet. Es wäre also erforderlich alle Kommentare aus dem Quelltext der aktuellen Übersetzungseinheit manuell herauszufiltern, irrelevante Kommentare auszufiltern und abschließend jedes gefundene Cid-Attribut dem korrekten IDL-Element zuzuordnen. Insbesondere der letzte Schritt ist dabei inhärent fehleranfällig.

Kodierung im Namen Ein weiterer möglicher Ansatz besteht darin, die zusätzlichen Informationen über die Namen betroffener Sprachelemente zu kodieren, indem diesen Vor- beziehungsweise Nachsätze angefügt werden. Für den Benutzer scheint das zunächst aufwendig und sehr einschränkend bezüglich der Namensgebung. Die Verwendung von Präprozessormakros gestattet es jedoch, den Mechanismus weitestgehend zu verbergen:

```
#define LEN(num, name) LEN_##num##_##name
#define MAX(num, name) MAX_##num##_##name
```

Allerdings stößt diese Methode schnell an Grenzen, wenn einem Element mehrere Attribute beigefügt werden sollen. Der Präprozessor wertet Makros grundsätzlich nicht mehrstufig aus, so dass Konstruktionen wie

```
... LEN(10, MAX(10, name)) ...
```

nicht korrekt aufgelöst werden. Zu diesem Zweck wären vielmehr Hilfsmakros erforderlich; eines für jede denkbare Attribut-Kombination. Auch bei dieser Umsetzung ist wiederum manuelles Nachparsen nötig; es beschränkt sich allerdings auf die Namen der Sprachelemente und diese werden von Clang innerhalb des AST bereitgestellt.

Annotationen Clang bietet zwar keine Möglichkeit zur Definition eigener GNU-Attribute, unterstützt jedoch das Annotationsattribut (`annotate`), mit dem beliebige Zeichenfolgen an Sprachelemente gebunden werden können. Im weiteren Übersetzungsprozess bleibt diese Verbindung bestehen und derart angefügte Annotationen sind einfach über den AST zugänglich.

Auch bei diesem Ansatz tritt das Problem der augenscheinlich umständlichen Nutzung auf. Wie bereits beschrieben, lassen sich Schreibaufwand und Komplexität durch Präprozessormakros einfach reduzieren:

```
#define LEN(num) __attribute__((annotate("len=" #num)))
#define MAX(num) __attribute__((annotate("max=" #num)))
```

Die Verwendung mehrerer GNU-Attribute mit einem Sprachelement ist zulässig und wird von Clang korrekt im AST abgebildet – die Kombination beliebiger Cid-Attribute stellt folglich kein Problem dar. Das ebenfalls wieder erforderliche, manuelle Parsing beschränkt sich allein auf die neu geschaffenen Cid-Attribute beziehungsweise deren textuelle Repräsentation. Aufgrund ihrer Vorzüge und einfachen Umsetzbarkeit wurde diese Variante letztendlich für die Implementierung ausgewählt.

3.5. Zielsprachen

Ein wesentlicher Vorteil von IDL-Compilern liegt in der Unterstützung verschiedener Zielsprachen. Der Compiler ist in der Regel dazu fähig, aus ein und derselben IDL-Spezifikation angepassten Kommunikationscode in jeder unterstützten Programmiersprache zu erzeugen.

Im Rahmen dieser Arbeit wurde die Codeerzeugung jedoch nur für eine Zielsprache implementiert, um den Entwicklungsaufwand in Grenzen zu halten. Bei der Entscheidung für C++ spielte die Zielplattform eine wesentliche Rolle: L4Re ist ebenfalls in dieser Sprache realisiert. Daher ist die Nutzung der vorhandenen Laufzeitumgebung aus C++-Kommunikationscode heraus direkt und unkompliziert möglich. Weiterhin vereinfacht sich die Übersetzung der sehr C++-nahen Cid-IDL in Quelltext, da Konzepte wie Namensräume und Klassen unmittelbar zur Verfügung stehen und bei der Codeerzeugung nicht aufwendig nachgebildet oder ersetzt werden müssen.

3.6. Marshalling-Methoden

Die Datenübertragung zwischen Client und Server wirkt sich entscheidend auf die Leistungsfähigkeit des generierten Kommunikationscodes aus. Während der zugrunde liegende IPC-Mechanismus des Fiasco.OC-Mikrokerns einer Beeinflussung durch den IDL-Compiler natürlich entzogen ist, bietet die konkrete Verwendung des IPC-Puffers Raum für Optimierungen verschiedener Art. Eine maßgebliche Rolle spielen dabei die Anordnung der zu übertragenden Daten im Puffer sowie die Art und Weise, auf die die Daten in den Puffer gelangen beziehungsweise daraus entnommen werden. Diese Freiheitsgrade beim Marshalling und Unmarshalling (siehe auch Abschnitt 2.2) schlagen sich in verschiedenen Marshalling-Methoden nieder.

Das ursprüngliche Entwicklungsziel für Cid war eine automatisierte Anwendung der in Abschnitt 2.2.2 auf Seite 14 vorgestellten IPC-Streams. Im Verlauf dieser Arbeit erschien es jedoch bald wünschenswert, die Nachteile dieses Mechanismus zu vermeiden; ein unmittelbar auf C-Strukturen (`struct`) basierendes Verfahren kam deshalb hinzu. Großer Wert wurde dabei auf die einfache Austauschbarkeit der beiden Marshalling-Methoden durch den Benutzer gelegt – für einen Wechsel genügt die Neuübersetzung der IDL-Spezifikation durch Cid. Anpassungen des Server- beziehungsweise Client-Codes, den der Benutzer implementiert, sind dagegen in der Regel nicht erforderlich. Selbst große Teile des automatisch erzeugten Kommunikationscodes sowie der grundsätzliche Aufbau des IPC-Puffers während der Übertragung ist bei beiden Verfahren identisch; weitere Details dazu folgen in Abschnitt 3.7 auf Seite 32.

Speicherverwaltung Der von Cid erzeugte Code umfasst grundsätzlich keine Speicherverwaltung für den Client; diese obliegt allein dem Benutzer. Auf Serverseite ist die Situation eine etwas andere: Ausschließlich die Daten, die sich im IPC-Puffer befinden, werden im Rahmen einer IPC übermittelt. Während der Server eine eingegangene Anfrage verarbeitet, müssen vom Client übermittelte Daten in einem lokalen Puffer zwischengespeichert werden, da andernfalls die Gefahr besteht, sie im IPC-Puffer durch eine weitere, geschachtelte IPC zu beschädigen. Für Daten, die vom Server an den Client übertragen werden sollen, gelten ähnliche Überlegungen; auch sie können daher vom Server nicht unmittelbar im IPC-Puffer abgelegt werden, sondern bedürfen einer Zwischenspeicherung.

In seiner Grundkonfiguration ist Cid darauf ausgelegt, zugunsten der Betriebssicherheit gegebenenfalls Kompromisse bei der Ausführungsgeschwindigkeit einzugehen (vgl. Abschnitt 3.3). Daher werden durch den generierten Server-Code passende lokale Puffer bereitgestellt, in denen Kopien der verarbeiteten Daten (eingehend und ausgehend) vorgehalten werden, solange der serverseitige Benutzercode läuft. Eine weitergehende Optimierung kann vom Nutzer explizit angefordert werden; Abschnitt 3.7 auf Seite 33 geht näher auf diese Möglichkeit ein.

Empfang von Capabilitys durch den Server Capabilitys nehmen auch bei der Übertragung mittels IPC eine Sonderstellung ein: Voraussetzung für den Empfang einer Capability unter Fiasco.OC ist es, einen entsprechenden Puffer bereitzustellen *bevor* die

IPC stattfindet, die die Capability überträgt. Für den Kommunikationscode des Clients ist das leicht zu arrangieren – ein passender Puffer wird dem System übergeben, ehe die IPC an den Server ausgelöst wird. Auf Serverseite hingegen besteht das Problem, dass verschiedene IDL-Funktionen unterschiedliche Parameter entgegennehmen; insbesondere die Anzahl eingehender Capabilitys ist nicht konstant. Da weiterhin unbekannt ist, welche IDL-Funktion als nächste von einem Client aufgerufen werden wird, ist eine einfache Lösung analog zum Client unmöglich. Stattdessen wird im Verlauf der statischen Analyse (das heißt während der Übersetzung der IDL-Spezifikation in Kommunikationscode) die maximale Anzahl eingehender Capabilitys über alle Funktionen einer Schnittstelle hinweg bestimmt. Eine entsprechende Anzahl an Capability-Puffern wird vom Server bereitgehalten und pauschal vor jeder Antwort-IPC ans System übergeben.

3.6.1. Marshalling mittels IPC-Stream

Das standardmäßig von Cid eingesetzte Marshalling-Verfahren, im Folgenden *„IPC-Stream-Marshalling“* genannt, nutzt die von L4Re bereitgestellten IPC-Streams (siehe Abschnitt 2.2.2). Folglich entsteht ein ähnlicher Kommunikationscode wie bei der manuellen Verwendung dieses Mechanismus. Für den Benutzer ist der Code entsprechend sehr gut les- und nachvollziehbar.

Zunächst mag der Einsatz eines IDL-Compilers für diese Aufgabe unnötig und übertrieben erscheinen. Die automatische Codeerzeugung kann jedoch einen Teil der Nachteile der IPC-Streams von L4Re kompensieren: Einen fehlerfreien IDL-Compiler vorausgesetzt, sind weder Inkonsistenzen in den Implementierungen auf Client- und Serverseite möglich, noch wird der Benutzer mit der Vergabe und korrekten Anwendung von Opcodes behelligt.

Auch die in Abschnitt 3.7 auf der nächsten Seite noch ausführlich vorgestellten Optimierungen finden natürlich Anwendung und führen zur bestmöglichen Nutzung der IPC-Streams. Die Durchmischung von Arrays und zugehörigen Längenparametern (vgl. Abbildung 3.3 auf Seite 31) bleibt dabei leider ebenso unvermeidlich wie die Beschränkung der Längenparameter auf `unsigned long` als Datentyp. Gegenüber dem Benutzer kann Cid zumindest letztgenannte verbergen: Die IDL-Spezifikation gestattet die Wahl beliebiger vorzeichenloser[3] Typen für die Längenparameter von Arrays. Der erzeugte Kommunikationscode verwendet in solchen Fällen eine zusätzliche Variable für das Un-/Marshalling des Arrays. Ihr Datentyp entspricht dem von IPC-Stream erwarteten `unsigned long` und wird mittels einer einfachen Typwandlung (`static_cast`) in den vom Benutzer gewünschten Typ umgesetzt.

3.6.2. Marshalling mittels Struktur

Die zweite verfügbare Marshalling-Methode, das *„struct-Marshalling“*, setzt direkt auf der IPC-Schnittstelle von Fiasco.OC auf. Die Unzulänglichkeiten der IPC-Streams (siehe Abschnitt 2.2.2 auf Seite 14) bleiben so außen vor. Zur Vereinfachung des Un-/Marshalling greift der von Cid erzeugte Code auf eine Struktur (`struct`) zurück, über

[3] Es ist sinnvoll die Beschränkung auf vorzeichenlose Datentypen beizubehalten, da zulässige Array-Längen ohnehin zwingend ≥ 0 sind.

die der IPC-Puffer angesprochen wird – daher der Name der Marshalling-Methode. Der C++-Compiler, der später den Kommunikationscode übersetzt, sorgt dabei für die korrekte Ausrichtung aller Elemente innerhalb der Struktur und damit auch im IPC-Puffer.

Die Struktur beinhaltet alle Parameter einer IDL-Funktion, deren Größe und Position im IPC-Puffer bereits bekannt sind, wenn die IDL-Spezifikation verarbeitet wird. Dazu gehören grundsätzlich alle einfachen Datenwerte; insbesondere auch die Längenparameter von Arrays (sofern vorhanden). Unter speziellen Umständen sind Arrays und Capabilitys ebenfalls in der Struktur enthalten: Wie in Abschnitt 3.7 auf dieser Seite noch ausführlich erläutert, werden alle Parameter einer IDL-Funktion in der bestmöglichen Reihenfolge verarbeitet und sind in ebendieser Reihenfolge als Elemente der Struktur angelegt. Das erste dabei auftretende Array mit dynamisch spezifizierter Länge und alle ihm nachfolgenden Parameter werden außerhalb der Struktur im IPC-Puffer abgelegt – Abbildung 3.3 auf der nächsten Seite illustriert dieses Schema. Für die korrekte Ausrichtung aller Parameter „hinter der Struktur“ erzeugt Cid eigenen Code, der an die Implementierung der IPC-Streams von L4Re angelehnt ist. Da die konkrete Größe von Arrays mit dynamisch spezifizierter Länge erst während der Laufzeit bekannt ist, verschieben sich die Positionen der Werte, die im IPC-Puffer auf das erste solche Array folgen, auf eine zur Übersetzungszeit[4] nicht ermittelbare Weise. Treten in einer IDL-Funktion hingegen keine Arrays dieser Art auf, werden alle Parameter in die Struktur aufgenommen und das Un-/Marshalling vereinfacht sich weiter.

Besonders im Zusammenhang mit der Verarbeitung großer Parametermengen (egal ob einfache Daten, Arrays oder Capabilitys) ergeben sich wesentliche Vorteile aus der beschriebenen Verwendung einer Struktur zum Un-/Marshalling: Da während der Übersetzung der IDL-Spezifikation in die Zielsprache bereits alle notwendigen Informationen verfügbar sind und der IPC-Puffer durch Cid frei organisiert werden kann, kommen sämtliche Optimierungen voll zum Tragen. Dazu zählen beispielsweise die Vorsortierung aller Parameter oder die Überprüfung auf Pufferüberläufe bereits zur Übersetzungszeit – Abschnitt 3.7 auf dieser Seite wird noch ausführlich auf alle eingesetzten Optimierungsverfahren eingehen. Im Fall von Arrays kommt hinzu, dass Längenparameter (auch im IPC-Puffer) von beliebigem Typ sein können. Bei statisch spezifizierter Länge entfällt deren Übertragung vollständig; sie ist zur Laufzeit unveränderlich und wird deshalb im erzeugten Code als Konstante betrachtet. Im Idealfall verringert sich somit das Datenvolumen der im Hintergrund ausgeführten IPC, was sich wiederum positiv auf deren Laufzeit auswirkt.

3.7. Optimierungen

Für den Kommunikationscode, den ein IDL-Compiler aus der IDL-Spezifikation des Benutzers erzeugt, ist neben Zuverlässigkeit und Fehlerfreiheit auch die Ausführungsgeschwindigkeit von großer Bedeutung. Kann der erzeugte Code diesen Anforderungen nicht gerecht werden, wird der Compiler für Benutzer unattraktiv. Cid führt deshalb

[4] Übersetzungszeit bezieht sich hier sowohl auf die Umsetzung der IDL-Spezifikation durch Cid als auch auf die Verarbeitung des erzeugten Kommunikationscodes durch einen C++-Compiler.

an verschiedenen Stellen des Übersetzungsprozesses Optimierungen durch. Während sich die stets aktiven Basisoptimierungen dabei nicht auf die spätere Ausführung der benutzerimplementierten Serverfunktionen auswirken, gilt dies nicht für die optionalen erweiterten Optimierungen. Ihr Einsatz kann zu unerwarteten Ergebnissen führen, sofern der Benutzer keine geeigneten Maßnahmen ergreift. Aus diesem Grund kommen erweiterte Optimierungen ausschließlich auf explizite Anweisung des Benutzers zur Anwendung.

Vorsortieren der Daten im IPC-Puffer Eine willkürliche Aneinanderreihung der zu übertragenden Daten im IPC-Puffer kann zusätzlichen Platzbedarf mit sich bringen. Wenn größere Datentypen auf kleinere folgen, muss der Größenunterschied durch Füllbytes (*„padding“*) ausgeglichen werden, um eine korrekte Ausrichtung (*„alignment“*) sicherzustellen; siehe auch Abbildung 3.2 auf der vorherigen Seite. Durch die geeignete Vorsortierung aller übertragener Daten – neben den Parametern der IDL-Funktion gehören dazu auch Opcode beziehungsweise Rückgabewert – versucht Cid die Menge der nötigen Füllbytes zu minimieren. Möglich wird diese Art der Optimierung, da der Inhalt des IPC-Puffers bereits zur Übersetzungszeit der IDL-Spezifikation bekannt ist.

Alle Daten werden einem einfachen Schema gemäß angeordnet. Auf einfache Datentypen am Anfang des Puffers folgen Arrays und schließlich Capabilitys. Sowohl einfache Datentypen als auch Arrays sind dabei jeweils absteigend nach ihrer Größe sortiert. Für gleichgroße Parameter wird die Reihenfolge ihres Auftretens in der Signatur der IDL-Funktion beibehalten. Abbildung 3.3 auf der vorherigen Seite veranschaulicht diese speziellen Anordnung:

- Die übertragenen Daten hängen direkt von der jeweils genutzten IDL-Funktion ab – verschiedene, von einem Server angebotene Funktionen arbeiten im Allgemeinen mit unterschiedliche Parametern. Folglich ist auf Serverseite die Kenntnis der vom Client aufgerufenen Funktion nötig, um eingehende Daten korrekt aus dem IPC-Puffer zu entnehmen. Aus diesem Grund wird bei Übertragungen vom Client an den Server stets der Opcode am Anfang des Puffers abgelegt. Er identifiziert die aufgerufene Funktion eindeutig (vgl. Abschnitt 2.2) und gibt damit indirekt Auskunft über die übertragenen Nutzdaten sowie deren Anordnung im IPC-Puffer.

- Eine absteigende Sortierung der Größe nach bringt den geringstmöglichen Bedarf an Füllbytes mit sich. Die korrekte Ausrichtung setzt voraus, dass Variablen an einer Speicheradresse beginnen, die ein Vielfaches der Größe des jeweiligen Datentyps darstellt. Da die Größen typischerweise Zweierpotenzen entsprechen, ist kein Auffüllen erforderlich, sofern ein kleiner Datentyp auf einen größeren folgt. Im umgekehrten Fall dagegen ist die Größendifferenz zwischen den beiden Typen auszugleichen. Abbildung 3.2 auf der vorherigen Seite illustriert diesen Zusammenhang. Wie bereits dargelegt ist im Fall von Opcodes ein Abweichen von diesem Schema zwingend notwendig.

- In den ersten Versuchen wurden Datentypen gleicher Größe willkürlich angeordnet. Jedoch zeigten die in Abschnitt 5.2 auf Seite 48 dokumentierten Leistungsmessungen unerwartete Anomalien, die sich auf diese Umordnung zurückführen ließen. Nach Änderungen, die nun dafür sorgen, dass gleich große Parameter diejenige Reihenfolge

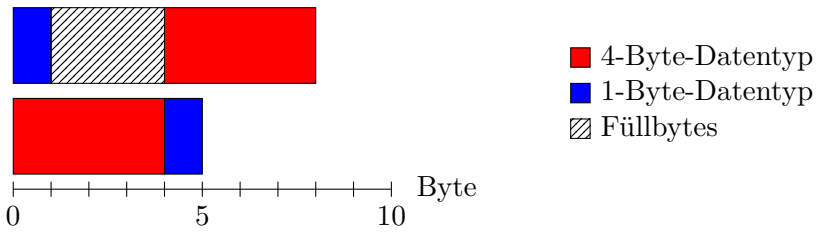


Abbildung 3.2.: Ausrichtung von Datenwerten mit Hilfe von Füllbytes

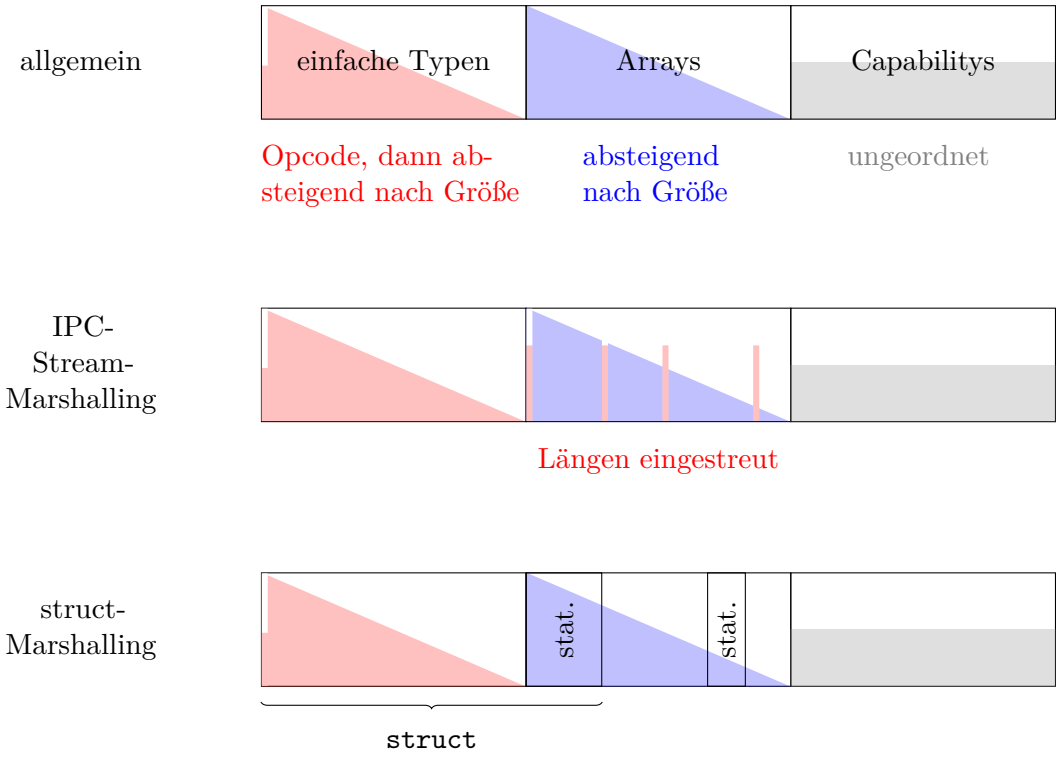


Abbildung 3.3.: Überblick über den Aufbau des IPC-Puffers
Übertragungsrichtung Client → Server, „stat.“ . . . Arrays mit statisch spezifizierter Länge, Füllbytes werden nicht dargestellt

beibehalten, in der sie in der Signatur der IDL-Funktion auftreten, verschwanden die erwähnten Anomalien. Die genaue Ursache wurde aus Zeitgründen nicht näher untersucht, aller Wahrscheinlichkeit nach spielen hier aber Cache-Effekte und/oder Optimierungen des C++-Compilers, der den Kommunikationscode schließlich übersetzt, eine Rolle.

- Die Länge eines jeden Arrays wird vor dem Arrays selbst gespeichert, um so Längenprüfungen korrekt durchführen zu können und ausschließlich auf belegte Bereiche des IPC-Puffers zuzugreifen. Bei Verwendung des in Abschnitt 3.6.1 vorgestellten IPC-Stream-Marshalling bleibt der Nachteil dieser Methode zwangsläufig bestehen: Die Länge *jedes* Arrays wird unmittelbar vor dem Array selbst im IPC-Puffer abgelegt. So kann es zu unnötigen Redundanzen – beispielsweise im Zusammenhang mit Arrays statisch spezifizierter Länge – und zum Einsatz zusätzlicher Füllbytes kommen.

- Ein Großteil der Daten unveränderlicher Größe – vor allem einfache Datentypen, unter günstigen Umständen aber auch Arrays mit statisch spezifizierter Länge sowie Capabilitys – befindet sich nach dem gewählten Schema an festen, bereits zur Übersetzungszeit bekannten Positionen innerhalb des IPC-Puffers. Insbesondere das in Abschnitt 3.6.2 bereits erläuterte struct-Marshalling profitiert dabei von dem Umstand, dass auf diese Parameter im Puffer direkt zugegriffen werden kann und zusätzliche Adressberechnungen zur Laufzeit somit entfallen.

  Der Ansatz, auch Arrays diesem Prinzip zu unterwerfen, und Arrays mit statisch spezifizierte Länge grundsätzlich vor jenen mit dynamisch spezifizierter Länge einzuordnen, liefe dem Prinzip zuwider, alle Parameter der Größe nach anzuordnen und wurde daher nicht umgesetzt.

- Der IPC-Mechanismus von Fiasco.OC erzwingt die Positionierung von Capabilitys am Ende des IPC-Puffers.

statische Prüfung auf Pufferüberläufe Bereits während der Übersetzung der IDL-Spezifikation sind die Signaturen sämtlicher Schnittstellenfunktionen bekannt. Damit stehen insbesondere Informationen zum (maximalen[5]) Umfang der Daten bereit, die im Hintergrund mittels IPC zu übertragen sind. Überläufe des IPC-Puffers werden so bereits zu diesem Zeitpunkt als Fehler erkannt. Der Benutzer erhält eine klare, leicht verständliche Fehlermeldung woraufhin die Übersetzung abgebrochen wird. Dieses in Cid angewandte Konzept des schnellen Abbruches wurde in Abschnitt 3.3 bereits vorgestellt.

Eine Verkürzung der Laufzeit des erzeugten Kommunikationscodes ergibt sich daraus, dass zur Laufzeit keine Prüfung auf Pufferüberläufe mehr nötig ist. Lediglich für Arrays mit dynamisch spezifizierter Länge sind Tests dieser Art weiterhin sinnvoll, um sicherzustellen, dass die in der IDL-Spezifikation vereinbarte maximale Array-Länge nicht

[5] Für Arrays mit dynamisch spezifizierter Länge, deren Länge im Rahmen eines konkreten RPC definitionsgemäß erst zur Laufzeit feststeht, wird die Prüfung anhand ihrer maximalen Länge durchgeführt. Diese wiederum ist Cid als obligatorischer Teil der Metainformationen aus der IDL-Spezifikation bekannt; vgl. Abschnitt 3.1.2.

überschritten wird. Während das struct-Marshalling diesen Vorteil voll ausnutzen kann, ist das im Rahmen des IPC-Stream-Marshalling (beiden Marshalling-Verfahren wurden in Abschnitt 3.6 ausführlich diskutiert) nicht der Fall – die von den IPC-Streams durchgeführten Laufzeitprüfungen sind obligatorisch und können durch Cid nicht deaktiviert werden.

Erweiterte Optimierungen Ausschließlich auf explizite Anweisung des Benutzer (mittels `FAST`-Attribut), führt Cid für IDL-Funktionen auch erweiterte Optimierungen durch. Diesen liegt die Idee zugrunde, auf die sonst genutzte serverseitige Pufferung ein- und ausgehender Daten zu verzichten und Zugriffe stattdessen direkt auf den IPC-Puffer umzuleiten – ein verminderter Kopieraufwand im Zuge des Un-/Marshalling ist die Folge. Für die vom Benutzer implementierten Serverfunktionen hat das zunächst keine unmittelbar spürbaren Auswirkungen. Die Signatur der Funktion wird zwar ebenfalls umgestaltet, um unnötige Kopien zu vermeiden, was jedoch auf eine Art und Weise geschieht, die keine Änderungen am Code erfordert: Anstelle von Wertparametern (*„call by value“*) kommen Referenzparameter (*„call by reference“*) zum Einsatz; Arrays werden durch die semantisch ohnehin sehr ähnlichen Zeiger ersetzt. Sofern der Benutzer die im Folgenden beschriebenen Einschränkungen beachtet, ist ein nachträgliches Zuschalten erweiterter Optimierungen schnell möglich und erfordert an den benutzereigenen Serverfunktionen lediglich minimale Anpassungen der Signaturen.

Sowohl konkrete Umsetzung als auch Umfang der erweiterten Optimierungen hängen maßgeblich von der gewählten Zielsprache und Marshalling-Methode ab. Der Verzicht auf lokale Puffer im automatisch generierten Kommunikationscode des Servers führt jedoch in jedem Fall dazu, dass derart optimierte Funktionen keinerlei Schutz der eingehenden Daten (`IN`, Übertragungsrichtung Client → Server) vor Veränderung oder Zerstörung garantieren können, sobald ausgehende Daten (`OUT`, Übertragungsrichtung Client ← Server) geschrieben oder weitere, geschachtelte IPCs verwendet werden. Dieser Zusammenhang wurde in Abschnitt 3.6 auf Seite 28 bereits angesprochen. Sämtliche Parameter, die sowohl ein- als auch ausgehend sind (`INOUT`, Übertragungsrichtung Client ↔ Server), entziehen sich prinzipbedingt den beschriebenen Anpassungen: In aller Regel wird ein solcher Parameter nicht während beider Übertragungen an derselben Stelle innerhalb des IPC-Puffers abgelegt. Selbst für den Fall, dass die betreffende IDL-Funktion ausschließlich einen einzigen Parameter nutzt, führt der Opcode zur beschriebenen Veränderung der Pufferstruktur, da er lediglich vom Client an den Server übertragen wird. Die Ersetzung lokaler Kopien durch direkte Verweise in den IPC-Puffer scheidet damit aus; Leseoperationen müssten an anderen Speicheradressen erfolgen als Schreiboperationen und C++ bietet kein Sprachelement mit einer derartigen Semantik. Für Parameter dieser speziellen Art kommen folglich auch bei der Anwendung erweiterter Optimierungen serverseitige Puffer zum Einsatz. Der Benutzer kann dies vermeiden, indem er zwei separate Parameter – einen pro Übertragungsrichtung – einsetzt.

Eine letzte Einschränkung ist durch die aktuelle Implementierung der IPC-Streams (vgl. Abschnitt 2.2.2) bedingt. Sie bieten keine Unterstützung für „vorgezogenes Marshalling“, das heißt es ist nicht möglich die endgültige Position eines Wertes im IPC-Puffer zu bestimmen, ohne tatsächlich Daten zu schreiben. Anders gesagt kann kein korrekter

Verweis nach dem oben beschriebene Schema erzeugt werden, sofern der entsprechende Parameter ein ausgehender ist. Für eingehende Arrays dagegen ist das nicht der Fall; hier erlauben auch IPC-Streams die Verwendung von Referenzen.

Einwegfunktionen Das Konzept der Einwegnachrichten wurde bereits im Zusammenhang mit DICE in Abschnitt 2.2.1 kurz angesprochen. Obwohl es sich dabei um keine Optimierung im eigentlich Sinne handelt, führt auch diese Technik zu einer Verbesserung der Laufzeit und soll daher an dieser Stelle erwähnt werden.

In der aktuellen Implementierung unterstützt Cid ausschließlich Einwegnachrichten, die vom Client an den Server übermittelt werden. Ist eine IDL-Funktion mit dem entsprechenden Cid-Attribut (`ONEWAY`) als „Einwegfunktion“ ausgezeichnet, wird die Bearbeitung der Anfrage durch den Server nicht abgewartet. Stattdessen arbeitet der Client nach ihrem Absenden umgehend weiter, während der Server die zugehörige Funktion wie üblich ausführt, abschließend aber keine Antwort sendet. Aus Sicht des Clients verkürzt sich die Laufzeit der IDL-Funktion damit nicht nur um die Zeit, die der Server für die Verarbeitung benötigt, sondern auch um die Übertragungsdauer der Antwort-IPC. Ebenso nimmt das Maß an möglicher paralleler Aktivität im System zu, da Client und Server gleichzeitig arbeiten können.

4. Implementierung

Das folgende Kapitel befasst sich mit Details der aktuellen Implementierung von Cid. In diesem Zusammenhang werden sowohl die Wiederverwendung bestehenden Programmcodes als auch die Erweiterbarkeit des IDL-Compilers näher betrachtet.

4.1. Wiederverwendung

Das zentrale Anliegen der vorliegenden Arbeit ist die möglichst umfassende Einbindung und Nutzung bereits vorhandenen, bewährten Programmcodes – primär dem des C++-Compilers Clang; siehe auch Abschnitt 2.3. Durch diese Wiederverwendung reduzieren sich Entwicklungs- und Wartungsaufwand deutlich, während gleichzeitig die Zuverlässigkeit des neuen Programms verbessert wird. Entsprechend wurde bei der Implementierung von Cid möglichst umfassend auf Code aus LLVM, Clang sowie der C++-STL zurückgegriffen.

4.1.1. Clang-AST

Cid basiert in wesentlichen Teilen auf Clang, dem C++-Front-End von LLVM. Dabei fungiert der AST als Bindeglied zwischen Clangs Infrastruktur auf der einen und dem für Cid neu entwickelten Code auf der anderen Seite.

4.1.1.1. Erzeugung

Die Umwandlung einer IDL-Spezifikation in den äquivalenten AST wird vollständig von den entsprechenden Clang-Komponenten durchgeführt. Dank der großen Ähnlichkeit zwischen IDL und C++ (vgl. Abschnitt 3.1.1) sind dazu keinerlei Anpassungen erforderlich. Auf Entwurf und Implementierung eines eigenen Front-Ends konnte so verzichtet werden. Lediglich die Metainformation-tragenden Cid-Attribute erfordern ein sehr einfaches, zusätzliches Parsing (siehe Abschnitt 3.4 auf Seite 26).

Der von Clang erzeugte AST wird vom neu geschriebenen Code nicht direkt verwendet. Stattdessen kommen Hüllklassen zum Einsatz, die die jeweiligen nativen AST-Objekte kapseln, den Zugriff auf häufig benötigte Daten vereinfachen und, sofern nötig, Zusatzinformationen vorhalten. Das oben erwähnte Parsing der Cid-Attribute beispielsweise wird durch `AttributeNode`, die AST-Hüllklasse für GNU-Attribute, realisiert. Auf Konzept und technischen Umsetzung der Hüllklassen wird Abschnitt 4.2.1 auf Seite 40 noch detailliert eingehen.

4.1.1.2. Ermittlung der Größe von Datentypen

Da sich während einer IPC alle übertragenen Daten im IPC-Puffer befinden müssen (vgl. Abschnitt 2.2), stellt dessen Größe eine natürliche Obergrenze für den Umfang der Datenübertragung dar. Cid ist dazu in der Lage, die Einhaltung der IPC-Puffergrenzen statisch – das heißt bereits im Rahmen der Übersetzung einer IDL-Spezifikation in entsprechenden Kommunikationscode – zu überprüfen; in Abschnitt 3.7 wurde dieser Mechanismus ausführlich vorgestellt.

Bei der Größenbestimmung der Einzelwerte kann wiederum auf Clang zurückgegriffen werden. Sowohl für einfache Datentypen als auch für komplexe Datentypen, die im AST abgebildet sind, ist die jeweilige Größe mittels weniger Methodenaufrufe ermittelbar. Clang selbst nutzt diese Informationen zur Festlegung des Speicherlayouts komplexer Typen und berücksichtigt dabei neben der Zielarchitektur auch die für eine korrekte Ausrichtung gegebenenfalls notwendigen Füllbytes.

4.1.1.3. Codeerzeugung

Mit der `Rewriter`-Klasse [clab] bietet Clang bereits ein nützliches Hilfsmittel für die Erstellung von Anwendungen, die Quellcode analysieren, modifizieren und anschließend wieder ausgeben (*„source to source transformation“*). Dabei sind allerdings nur kleinere Änderungen vorgesehen, durch die sich die Struktur des Quellcodes lediglich lokal verändert; beispielsweise die Auflösung von Programmschleifen (*„loop unwinding“*). Da Cid als IDL-Compiler aber aus einer sehr kompakten, rein deklarativen IDL-Spezifikation gänzlich neuen Kommunikationscode erzeugt, der sich zudem noch über verschiedene Dateien verteilt, erwies sich der Einsatz von `Rewriter` im Rahmen dieser Arbeit als unzweckmäßig.

Des Weiteren hält Clang aber auch Mechanismen bereit, um aus einem vorhandenen AST erneut lauffähigen Quellcode zu erzeugen (*„pretty printing“*). Das ist nur möglich, weil der Compiler in seinem AST außergewöhnlich umfassende und detaillierte Informationen zum Ausgangsquelltext abbildet. Jedoch kann auch diese Funktionalität nicht unmittelbar zur Codeerzeugung herangezogen werden:

- Der von Clang aus einem AST erzeugte Code ist keineswegs perfekt, vereinzelt (beispielsweise im Zusammenhang mit anonymen Namensräumen) nicht einmal gültiger C++-Code. Zur Behebung dieses Mangels wären mehrere, teilweise umfangreiche Anpassungen von Clang erforderlich.

- Mit dem AST als alleinigem Ausgangspunkt für den erzeugten Code ist es unumgänglich, den gesamten Kommunikationscode zunächst vollständig im AST aufzubauen. Insbesondere müssen alle verwendeten Datentypen, Anweisungen (Zuweisungen, Verzweigungen, Schleifen, ...) in dieser Form erzeugt/bereitgestellt werden.

Insgesamt steht der erforderliche Aufwand in keinem angemessen Verhältnis zum erwarteten Nutzen, so dass auch dieser Ansatz schnell verworfen wurde. Stattdessen wurde die gesamte Codeerzeugung von Grund auf neu implementiert. Wo immer möglich und sinnvoll wird dabei jedoch auf die oben beschriebene Fähigkeiten von Clang

zurückgegriffen, um Code direkt aus dem AST zu generieren. Ein Beispiel dafür ist die Deklaration der Datenstrukturen für das struct-Marshalling (siehe auch Abschnitt 3.6.2). Für die statische Prüfung auf IPC-Pufferüberläufe (vgl. Abschnitt 3.7) werden die entsprechenden Daten ohnehin im AST benötigt, so dass an dieser Stelle kein Mehraufwand entsteht.

4.1.2. Datenhaltung

Der überwiegende Teil aller von Cid verwendeten Verwaltungsdatenstrukturen entstammt LLVM beziehungsweise Clang [clab, llvb]. Besonders häufig anzutreffen sind dabei `StringRef` zur effizienten Behandlung von (wiederholt auftretenden) Zeichenketten oder `SmallVector`, eine auf die Speicherung weniger Elemente optimierte Variante der `vector`-Klassenvorlage aus der C++-STL. Neben den bereits erläuterten Vorteilen einer Wiederverwendung bewährten Programmcodes bringen die aus LLVM und Clang übernommenen Datentypen auch eine verbesserte Kompatibilität zum Stammprogramm mit sich, so dass sich die Mitnutzung anderer Funktionen einfacher gestaltet.

4.1.3. Auswertung von Kommandozeilenargumenten

Wie die Cid-Attribute (vgl. Abschnitt 3.4) wurden auch die an Cid übergebenen Kommandozeilenargumente ursprünglich durch sehr einfachen, gänzlich neu entwickelten Code verarbeitet. Im Hinblick auf zukünftige Erweiterungen und um die Wartung zu erleichtern, kommen dafür nun die in Clang vorhandenen Parsing-Mechanismen [clab] zum Einsatz. Nach Übergabe und Verarbeitung aller Kommandozeilenargumente steht ein Objekt des Typs `InputArgList` bereit, das komplexe Auswertungen mittels einfacher Methodenaufrufe erlaubt.

Für Analyse und Vorverarbeitung der Cid-Attribute hingegen wurde das selbst implementierte Parsing beibehalten. Im Gegensatz zu Kommandozeilenargumenten sind diese Attribute sehr einfach und gleichförmig strukturiert, so dass die Anbindung der oben beschriebenen Mechanismen mehr Aufwand als Nutzen mit sich bringt. Des Weiteren existieren im Umfeld von Clang Bestrebungen, die Einbindung selbstdefinierter Attribute zu vereinfachen, was ohnehin jeder manuellen Auswertung vorzuziehen wäre.

4.2. Erweiterbarkeit

Erst im Verlauf der Implementierung von Cid trat die einfache Erweiterbarkeit als weiteres zentrales Entwicklungsziel in den Vordergrund. Eine klare Strukturierung, Modularisierung sowie die Einführung von Abstraktionsebenen erleichtern künftige Erweiterungen und Anpassungen; beispielsweise zur Unterstützung weiterer Zielsprachen, zur Einführung neuer IDL-Sprachelemente oder zur Anpassung an Veränderungen von Clang beziehungsweise L4Re. Die im Folgenden vorgestellten Techniken ermöglichten die vollständige Umsetzung des `ONEWAY`-Attributs in unter einer Stunde, obwohl dieses erst gegen Ende des Entwicklungsprozesses aufgenommen wurde.

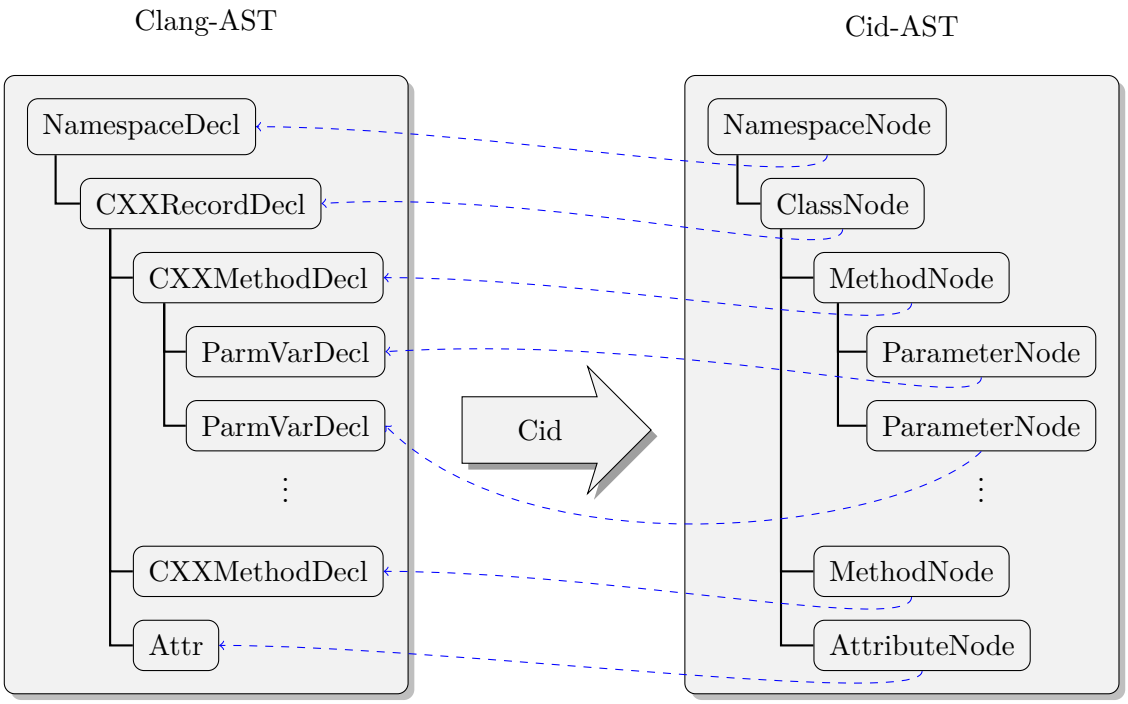


Abbildung 4.1.: Beziehung zwischen Cid-Hüllklassen und Klassen des Clang-AST (gestrichelte Linie ... Referenz auf gekapselte Klasse, siehe auch Abbildung 2.3)

4.2.1. AST-Hüllklassen

Der bewährte Compiler-Aufbau, wie er in Abschnitt 2.3 vorgestellt wurde, liegt auch Cid zugrunde. An die Stelle eines neu entwickelten Front-Ends für die IDL tritt jedoch Clang – siehe dazu auch Abschnitt 3.2.

Wie in Abschnitt 4.1.1 bereits angesprochen, wird der von Clang erzeugte AST für die weitere Verarbeitung nicht direkt, sondern unter Zuhilfenahme von Hüllklassen genutzt, welche die weitere Programmierung stark vereinfachen. Der so entstehende „Cid-AST“ und insbesondere dessen Schnittstelle zum Rest des IDL-Compilers lässt sich – völlig unabhängig von Clang – frei gestalten. Dabei werden Funktionen verborgen, die der Clang-AST zwar anbietet, die von Cid aber nicht benötigt werden. An anderer Stelle kommen neue Funktionen hinzu; beispielsweise die Verwaltung des Opcodes einer IDL-Funktion. Ein willkommener Nebeneffekt der AST-Kapselung ist die Konzentration aller erforderlichen Konsistenzprüfungen der IDL-Spezifikation in den Konstruktoren der Hüllklassen. Zusätzliche Auswertungen – wie die automatische Vergabe von Protokoll-ID und Opcode (vgl. Abschnitt 3.1.2) oder das nötige Parsing der Cid-Attribute (vgl. Abschnitt 3.4) – finden ebenfalls an dieser Stelle statt.

Bei der technischen Umsetzung der Hüllen waren mehrere Anläufe nötig. Erst eine Kombination von einfachen Klassen und Klassenvorlagen (*„class template“*) erwies

sich als geeignet im Sinne des Entwurfsprinzips „DRY“[1] sowie der vollständigen Erhaltung von Typinformationen; zwei Punkten, die wesentlich zur Zuverlässigkeit des fertigen Programms beitragen. Abbildung 4.2 auf der nächsten Seite zeigt die gesamte Klassenhierarchie und ergänzt damit die nachfolgenden Erläuterungen der einzelnen Bestandteile.

Die Basis aller AST-Hüllklassen bildet `BaseNode`, eine einfache Klasse welche sowohl Debug-Ausgaben als auch die damit assoziierte rudimentäre RTTI (*„Runtime Type Information“*) implementiert. Darauf aufbauend ergänzt `GenericNode`, die erste von zwei Klassenvorlagen, die Kapselung des zugehörigen Objektes aus dem nativen Clang-AST – Abbildung 4.1 auf der vorherigen Seite veranschaulicht die dabei auftretenden Zuordnungen. `AttributeNode` auf der nächsten Stufe der Hierarchie kapselt beliebige GNU-Attribute. Darüber hinaus dient die Klasse auch zur Umsetzung der Cid-Attribute: Alle Annotationsattribute werden zunächst näher analysiert. Handelt es sich um ein Cid-Attribut, wird es weiterverarbeitet und ist im Folgenden mit Hilfe der Methoden von `AttributeNode` nutzbar. `NodeWithAttribute` ist eine Verwaltungsklasse, die sämtliche GNU-Attribute/`AttributeNode`s eines AST-Elements vorhält und einfach zugänglich macht. Mangels Strukturinformationen zum Gesamt-AST kann die Vererbung von Cid-Attributen (siehe Abschnitt 3.1.2) an dieser Stelle noch nicht umgesetzt werden. Das geschieht erst innerhalb der zweiten Klassenvorlage, `NamedNode`. Es repräsentiert alle benannten C++-Sprachelemente und bildet ihre Beziehungen untereinander ab, indem Verweise auf den oder die Elternknoten eines AST-Knotens vorgehalten werden. Ähnliche Informationen liegen im Clang-AST bereits vor. Jedoch berücksichtigen die dortigen Verweise die AST-Kapselung natürlich nicht, sondern verknüpfen die nativen AST-Knoten.

Der eigentliche Cid-AST setzt sich schließlich aus den Instanzen derjenigen vier Klassen zusammen, die am Ende der soeben erläuterten Vererbungshierarchie stehen. Sie repräsentieren die verschiedenen Sprachelemente der IDL (vgl. Abschnitt 3.1.2):

`NamespaceNode`	→ Namensraum/Sektion
`ClassNode`	→ IDL-Klasse/Schnittstelle
`MethodNode`	→ IDL-Funktion
`ParameterNode`	→ Parameter einer IDL-Funktion

4.2.2. Modularisierung

Während Clang in praktisch unveränderter Form als Front-End von Cid fungiert, wurde das Back-End, also die Erzeugung des Kommunikationscodes aus dem AST, vollständig neu implementiert. Ein besonderes Augenmerk lag dabei auf einer sauberen Aufteilung und Modularisierung, um so neben der Wartung auch künftige Erweiterungen des IDL-Compilers zu erleichtern. Die Struktur des Programmcodes sowie dessen Ablage im Dateisystem sind dazu hierarchisch organisiert:

[1] *„Don't repeat yourself“* steht für die Vermeidung von Wiederholungen im Programmcode. Auf diese Weise werden Inkonsistenzen zwischen Kopien des Codes an verschiedenen Stellen ausgeschlossen und gleichzeitig die Wartung vereinfacht.

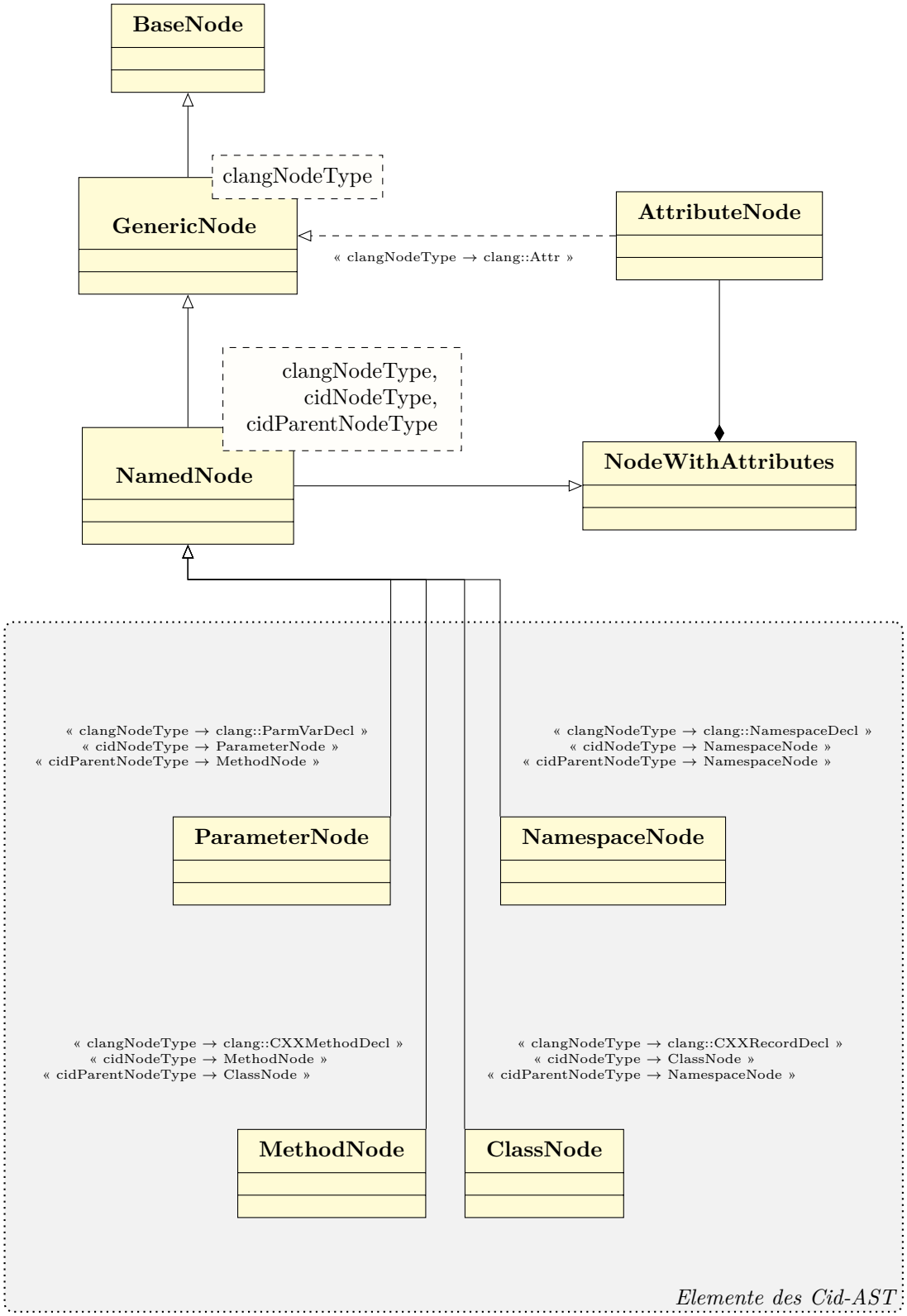


Abbildung 4.2.: Hierarchie der Cid-AST-Klassen (vereinfacht)

- In den oberen Ebenen stehen Implementierungen für häufig anfallende Aufgaben bereit; beispielsweise das Traversieren der Parameter einer IDL-Funktion oder Hilfsfunktionen für die Einbettung von L4Re-Code in den erzeugten Programmcode. Auch die Ausgabe der verschiedenen Quellcodedateien für den erzeugten Kommunikationscode ist hier bereits angelegt.

- Die nachfolgenden Hierarchieebenen halten Spezialisierungen und konkrete Ausgestaltungen der allgemeinen Mechanismen vor – zunächst für die einzelnen Zielsprachen und letztlich auch für die jeweils unterstützten Marshalling-Methoden. Eine neue Zielsprache muss nicht zwingend alle Marshalling-Methoden (sofort) unterstützen.

  In technischer Hinsicht werden dabei neben dem klassischen Überschreiben ererbter Methoden vor allem Rückruffunktionen genutzt: Die allgemeine Implementierung ruft an ausgewählten Stellen virtuelle Methoden auf, die von den spezialisierten Klassen überschrieben werden, um so den jeweils benötigten, spezifischen Code auszuführen.

- Auf welcher Ebene Optimierungen des Kommunikationscodes (siehe Abschnitt 3.7) angesiedelt sind, hängt davon ab, inwieweit sie für verschiedene Zielsprachen beziehungsweise Marshalling-Methoden anwendbar sind – je allgemeiner das Optimierungsverfahren, desto höher in der Hierarchie ist es zu finden. Gegebenenfalls nötige, spezifische Anpassungen werden, wie bereits beschrieben, mittels Überladungen und Rückruffunktionen umgesetzt.

Der Zugriff auf die Infrastruktur von Clang – insbesondere auf komplexe Funktionen, wie die Auflösung von Namen (*„name lookup“*) oder die Ausgabe von Diagnose- und Statusmeldungen – sowie die Einbettung von L4Re-Codefragmenten in den erzeugten Kommunikationscode sind so gut wie möglich in zentralen Klassen zusammengefasst. Auch sie folgen dem oben vorgestellten hierarchischen Aufbau. Die einzige Ausnahme von diesem Prinzip stellen die bereits diskutierten AST-Hüllklassen dar. Sie sind ohnehin sehr eng an den Clang-AST gekoppelt und greifen daher, nicht zuletzt aus Gründen der Übersichtlichkeit, direkt darauf zu.

4.2.3. X-Makros

Clang setzt an verschiedenen Stellen so genannte „X-Makros“ ein und auch Cid bedient sich dieser Technik. Dabei wird der C++-Präprozessor auf eine eher unkonventionelle Art eingesetzt: Eine separate Konfigurationsdatei enthält die gewünschten Informationen in Form von Makroaufrufen, jedoch *keine* Definitionen der verwendeten Makros. Dateien dieser Art tragen bei Cid die Endung `.inc`, um sie besser von normalem Code abzugrenzen.

aus „ast/AstNodes.inc“

```
NODE(NamespaceNode)
NODE(ClassNode)
NODE(MethodNode)
```

```
NODE(ParameterNode)
NODE(AttributeNode)
```

Als Beispiel für die Verwendung von X-Makros dient an dieser Stelle und im Folgenden das einfache RTTI der AST-Hüllklassen. Die soeben gezeigte Konfigurationsdatei definiert zunächst alle Elemente des Cid-AST (vgl. Abschnitt 4.2.1).

Werden die in der Konfigurationsdatei hinterlegten Informationen im eigentlichen Quelltext benötigt, erfolgt dort eine passende Definition der verwendeten Makros sowie die anschließende Einbindung der Konfigurationsdatei mittels `#include`-Direktive. Im folgenden Auszug wird auf diese Art eine Aufzählung aller AST-Hüllklassen definiert.

aus „ast/BaseNode.h“

```
enum NodeType {
  // the node types
  #define NODE(NAME) TYPE_##NAME,
  #include "AstNodes.inc"
  TYPE_Last
  #undef NODE
};
```

Die gezeigte Aufhebung der Makrodefinition ist nicht zwingend nötig, vermeidet aber Fehler bei der mehrfachen Nutzung dieser Technik innerhalb einer Übersetzungseinheit. Der Präprozessor setzt die eingebundenen Makroaufrufe gemäß Definition um und bringt so die zentral abgelegten Informationen in die benötigte Form.

vom Präprozessor erzeugt

```
enum NodeType {
  // the node types
  TYPE_NamespaceNode,
  TYPE_ClassNode,
  TYPE_MethodNode,
  TYPE_ParameterNode,
  TYPE_AttributeNode,
  TYPE_Last
};
```

Ihren eigentlichen Nutzen entfalten X-Makros jedoch erst dadurch, dass sie die zentral vorgehaltenen Informationen an verschiedenen Stellen in jeweils passender Form in den Quellcode einbetten. Die eingangs vorgestellte Konfigurationsdatei dient Cid so auch zur „Umwandlung“ eines AST-Knotentyps in die äquivalente Zeichenkette[2], wie sie unter anderem zur Beschreibung eines AST-Knotens im Rahmen von Debug-Ausgaben benötigt wird.

aus „ast/BaseNode.cpp“

```
llvm::StringRef BaseNode::getNodeTypeAsString(NodeType type) {

  switch(type) {
    #define NODE(NAME) case TYPE_##NAME: \
```

[2] Bei `llvm::StringRef` handelt es sich um einen Datentyp, der von LLVM bereitgestellt wird und im hier gezeigten Kontext annähernd mit `std::string` gleichzusetzen ist.

```
      return #NAME;
    #include "AstNodes.inc"
    #undef NODE
  }
}
```

Das Hinzufügen neuer sowie das Bearbeiten bestehender Einträge in der entsprechenden Konfigurationsdatei hat grundsätzlich globale Auswirkungen; mittels X-Makros erzeugter Code passt sich automatisch an derartige Veränderung an. Er ist daher jederzeit und über das gesamte Programm hinweg konsistent und auf dem aktuellen Stand.

4.2.4. Selbstbeschränkung

Ganz im Sinne der objektorientierten Programmierung setzt Cid konsequent auf das Verbergen von Implementierungsdetails (*„information hiding“*). Zugriffe auf den internen Zustand einer Klasse laufen grundsätzlich über definierte Schnittstellen ab, wobei Schreibzugriffe besonders strikten Beschränkungen unterliegen. Insbesondere die Instanzen der AST-Hüllklassen sind, wie bereits der Clang-AST, zweckmäßigerweise als unveränderbare Objekte konzipiert.

Vereinzelt kommt es dabei zu Konflikten; so beispielsweise bei der automatischen Vergabe von Opcodes (siehe auch Abschnitt 3.1.2): Der Opcode einer IDL-Funktion wird von der zugehörigen Cid-AST-Klasse `MethodNode` verwaltet. Für die automatische Opcode-Vergabe ist jedoch die Kenntnis aller Funktionen (sowie der ihnen manuell zugewiesenen Opcodes) einer Schnittstelle erforderlich[3]. Erst `ClassNode` auf der nächsthöheren Ebene des AST verfügt über diese Informationen. Das Problem besteht nun darin, dass die automatisch zugewiesenen Opcodes zwar in den zugehörigen `MethodNode`-Objekten hinterlegt werden müssen, eine zu diesem Zweck bereitgestellte Zugriffsmethode jedoch allgemein zugänglich wäre und damit beliebigem Code Schreibzugriff auf den Opcode erlauben würde.

```
class MethodNode {
...
public:
void setOpcode(Opcode opcode) {
  this._opCode = opcode;
}
...
};
```

Der `friend`-Mechanismus von C++ bietet sich als mögliche Alternative an. Mit seiner Hilfe kann einer ausgewählten Funktion oder Klasse (hier `ClassNode`) unbeschränkter Zugriff auf die privaten Daten einer anderen Klasse (hier `MethodNode`) gewährt werden. Allerdings wird auch dabei die Datenkapselung unnötig stark beeinträchtigt, da `ClassNode` auf diese Weise nicht nur Zugang zum Opcode, sondern auch auf alle anderen internen Daten von `MethodNode`, erhält.

[3] Zur Erinnerung: Der Benutzer kann IDL-Funktionen wahlweise manuell einen Opcode zuordnen. Allerdings muss jeder Opcode (im Rahmen eines Protokolls) eindeutig sein – vom Benutzer vergebene Werte müssen demzufolge bei der automatischen Opcode-Vergabe ausgespart werden.

```
class MethodNode {
...
friend class ClassNode;
};
```

Cid nutzt an dieser Stelle die neu geschaffene Klasse `OpcodeSetter`. Als innere Klasse kann `OpcodeSetter` auf alle private Daten von `MethodNode` frei zugreifen [ISO03]. Eine Instanziierung von `OpcodeSetter` ist nicht vorgesehen und die Schnittstelle der Klasse umfasst nur eine einzige private, statische Methode zum Ändern des Opcodes. Während `ClassNode` mittels `friend`-Deklaration Zugriff auf diese Methode erhält, bleibt allen anderen Klassen Schreibzugriff auf den Opcode verwehrt und der Eingriff in die Datenkapselung fällt so gering wie möglich aus.

```
class MethodNode {
...
public:
class OpcodeSetter {
  OpcodeSetter(); // NOT IMPLEMENTED

  static void set(MethodNode& mN, Opcode opcode) {
    mN._opCode = opcode;
  }
  friend class ClassNode;
};
...
};
```

Zusicherungen Ein weiteres wichtiges Hilfsmittel bei der Wahrung von Programmkonsistenz und -fehlerfreiheit sind die zahlreich in der Codebasis von Cid enthaltenen Zusicherungen (`assert`). Sie stellen die Einhaltung der impliziten Annahmen sicher, auf denen der nachfolgende Programmcode aufbaut. Beispiele für derartige Voraussetzungen sind die fehlerfreie Initialisierung der Elemente des Cid-AST oder der korrekte Aufbau von Konfigurationsdateien, die durch X-Makros (siehe Abschnitt 4.2.3) weiterverarbeitet werden. Dadurch werden recht freie Änderungen sowie Ergänzungen an der Codebasis von Cid möglich: Unerwünschte Wechselwirkungen mit vorhandenem Code führen bei Tests schnell zum Fehlschlagen von Zusicherungen. Deren Position im Quelltext sowie die in sie integrierte Meldung geben dem Programmierer dann erste Hinweise auf die Fehlerursache.

5. Leistungsbewertung

Dieses Kapitel untersucht den praktischen Nutzen von Cid in seiner aktuellen Entwicklungsstufe. Dazu wird zunächst der Umfang der vorliegenden Implementierung betrachtet, um den Einfluss des IDL-Compilers auf die TCB des Gesamtsystems abzuschätzen. Zur Überprüfung der Leistungsfähigkeit des erzeugten Kommunikationscodes werden anschließend verschiedene Messungen an einem realen System durchgeführt.

5.1. Codeumfang

Erwartungsgemäß fällt der Anteil des für Cid neu erstellten Codes mit weniger als 8.000 SLOC (Details finden sich in Tabelle 5.1 auf dieser Seite) relativ klein aus. Damit bleibt der Umfang des IDL-Compilers deutlich hinter den rund 10.000 SLOC von DICE[1] zurück. Darüber hinaus besteht an dieser Stelle Raum für weitere Verbesserungen: Eine stärkere Modularisierung des Erstellungsprozesses von Cid – beispielsweise durch Nichteinbeziehen von Marshalling-Methoden (siehe Abschnitt 3.6), deren Verwendung nicht vorgesehen ist – kann den Eintrag in die TCB noch reduzieren.

Für eine realistische Abschätzung des Einflusses von Cid auf die TCB des Systems genügt die alleinige Betrachtung des Cid-eigenen Codes jedoch nicht. Zusätzlich ist auch dessen Basis (Clang und LLVM) zu berücksichtigen, wodurch sich der Umfang des TCB-relevanten Codes um beachtliche 800.000 SLOC[2] erhöht. Allerdings dürfte insbesondere der Anteil an tatsächlich genutztem LLVM-Code sehr gering ausfallen, da sowohl Optimierungen als auch Codeerzeugung neu implementiert wurden. Insofern sollten die genannten Werte als pessimistische Abschätzung des tatsächlichen Codeeintrags betrachtet werden.

In Anbetracht einer Vergrößerung der TCB um über 800.000 SLOC erscheint die Nutzung des IDL-Compilers, gerade bei einem Mikrokern-basierten System wie L4Re,

[1] DICE in SVN-Revision 473
[2] Gesamtcodeumfang von LLVM und Clang in der SVN-Revision 139.090

Programmteil	SLOC (gerundet)
ast	2.200
codegen	4.200
support	1.000
startup	500
gesamt	7.900

Tabelle 5.1.: Codeumfang – Cid

zunächst äußerst kritisch. Die Situation ist jedoch eine grundlegend andere, wenn Clang auch als C++-Compiler für das Basissystem selbst verwendet wird. In diesem Fall sind LLVM und Clang ohnehin bereits Teil der TCB und die durch Cid zusätzlich eingebrachte Codemenge beschränkt sich auf die eingangs genannten 8.000 SLOC.

5.2. Laufzeitverhalten

Es wurde eine Reihe verschiedenster Messungen durchgeführt, um die Leistungsfähigkeit des von Cid erzeugten Kommunikationscodes näher zu untersuchen. Während ein Vergleich der beiden zurzeit implementierten Marshalling-Methoden (vorgestellt in Abschnitt 3.6) dabei von besonderem Interesse ist, wurde auf eine Gegenüberstellung mit manuell implementiertem Kommunikationscode bewusst verzichtet. Dieser greift in der Regel ohnehin auf die IPC-Streams von L4Re (vgl. Abschnitt 2.2.2) zurück und gleicht daher praktisch dem Code, den Cid im Rahmen des IPC-Stream-Marshalling erzeugt.

Der für die Messungen verwendete Rechner verfügt über einen Intel Core i5-430M Prozessor (2,26 GHz[3], 3 MB L3-Cache) und führte im Rahmen der Versuche Fiasco.OC und L4Re (beide in SVN-Revision 36) nativ aus. Als C++-Compiler kam g++ in der Version 4.5.2 zum Einsatz. Während der Mikrokern selbst mit Core2-Optimierungen übersetzt wurde, blieben für L4Re aufgrund technischer Probleme Optimierungen deaktiviert.

5.2.1. Messverfahren

Alle Zeitmessungen erfolgten mit Hilfe der `rdtsc`-Anweisung des Prozessors. Für jede Versuchsreihe wurde dabei die durchschnittliche Dauer eines vollständigen RPC (Client → Server → Client) als Mittelwert aus jeweils 100.000 Einzelmessungen bestimmt, wobei der entsprechende Code vor jeder Messung zunächst einmal durchlaufen wird (*„warm up“*). Zunächst kam ein sehr einfachen Messverfahren zum Einsatz, das sich stark am Ping-Pong-Benchmark von L4Re orientierte und ausschließlich die mittlere Laufzeit jedes RPC erfasste. Bereits kurz nach Beginn der Versuche fielen jedoch einige Anomalien in den gemessenen Werten auf – Abschnitt 5.2.3 auf Seite 55 wird darauf noch näher eingehen. Die Suche nach ihrer Ursache war Anlass für die Umstellung auf ein neues, komplexeres Messverfahren, das neben dem Mittelwert auch Minimum, Maximum sowie Standardabweichung für jede Testreihe festhält.

Messabweichung Auf eine exakte Bestimmung der Messabweichung (Verfälschung/Erhöhung der gemessenen Zeiten durch die Messung selbst) wurde verzichtet. Dieser zusätzliche Zeitbedarf geht gleichermaßen in die Messungen beider Marshalling-Methoden ein, hebt sich bei deren Vergleich also ohnehin praktisch auf. Im Zuge der oben erläuterten Umstellung des Messverfahrens nahmen die beobachteten Laufzeiten bei Messungen mit einfachen Datenwerten im Mittel um 10 Takte, bei Messungen mit Arrays um etwa 50 Takte zu. Diese Werten stellen somit eine grobe Abschätzung der Mindestabweichung dar.

[3] Die wiederholte Ermittlung der tatsächlichen Taktrate während der Messungen mittels `l4_get_hz` zeigte, dass die „Turbo Boost“-Funktion des Prozessors trotz Verwendung der Einprozessor-Version von Fiasco.OC nicht aktiv wurde.

5.2.2. Messreihen

Sofern nicht explizit anders angegeben, erfolgte die Übertragung der Nutzdaten bei allen durchgeführten Versuchen stets vom Client an den Server, die als separate Tasks/Prozesse (also insbesondere in getrennten Adressräumen) liefen. Arrays gleicher Länge nutzten in der zugrunde liegenden IDL-Spezifikation einen gemeinsamen Längenparameter. Neben dem eingesetzten Quellcode enthält Anhang A auch die Primärdaten aller Messungen. Die dort verwendeten Versuchsnamen werden in den folgenden Abschnitten, die sich detailliert mit den einzelnen Testreihen befassen, jeweils in Klammern mit aufgeführt.

Zur einfacheren Lesbarkeit wurden alle im Folgenden diskutierten Messwerte auf die Laufzeit eines einfachen Ping-Pong-Laufs normiert. Diese wurde mit dem gleichnamigen L4Re-Benchmark zu rund 1.260 Takten ermittelt. Angegeben ist nun jeweils die „Verlangsamung“ eines von Cid erzeugten RPC gegenüber diesem Basiswert: 0 % entspricht einer gemessenen mittleren Laufzeit von ebenfalls etwa 1.260 Takten, 50 % einer mittleren Laufzeit von etwa 1.900 Takten.

Aufgrund der unterschiedlichen Messverfahren für den L4Re-Ping-Pong-Lauf auf der einen und den von Cid generierten Kommunikationscode auf der anderen Seite, ist es nicht unproblematisch, diese Werte direkt zueinander in Beziehung zu setzen. Im Vordergrund steht jedoch der Vergleich der beiden von Cid unterstützten Marshalling-Methoden und die entsprechenden Messwerte wurden durchweg konsistent erhoben. Der vom L4Re-Ping-Pong-Benchmark gelieferte Basiswert kann insofern als (willkürlich gewählte) Konstante betrachtet werden. Gleichzeitig repräsentiert dieser Wert jedoch näherungsweise auch die Dauer eines minimalen RPC (ohne jede Datenübertragung), so dass die angegebene Verlangsamung einen einfachen Indikator für die Effizienz des erzeugten Kommunikationscodes darstellt.

5.2.2.1. Einfache Datenwerte

Im Verlauf der Testreihe werden, ausgehend von einem reinen Ping-Pong ohne Nutzdatenübertragung (*„PingPong“*), zunehmend größere Datenmengen in Form einzelner `int`-Werte (*„Simple“*) übertragen; die untere Hälfte von Abbildung 5.1 auf der nächsten Seite veranschaulicht die gemessenen Laufzeiten. Bei jeder Client-Anfrage wird der zugehörige Opcode mit übertragen, so dass der IPC-Puffer in jedem Fall beteiligt ist.

Ergebnisse Insbesondere die beobachtete Verlangsamung des Cid-Ping-Pong fällt überraschend stark aus: Obwohl ausschließlich der Opcode übertragen wird, nimmt die Laufzeit unabhängig von der verwendeten Marshalling-Methode um nahezu 60 % zu. Dieser starke Einbruch begründet sich offenbar zum einen in der Verwendung des IPC-Puffers, zum anderen aber auch im zusätzlichen Code, der an jeder RPC-Bearbeitung beteiligt ist: mehrfache Indirektionen, Verwaltung des IPC-Puffers, ... bis hin zur Auswertung von Protokoll-ID und Opcode durch den Server.

Im weiteren Verlauf der Messreihe mit einfachen Datentypen wird deutlich, dass die RPC-Laufzeiten annähernd proportional zur übertragenen Datenmenge zunehmen: Ausgehend von der soeben diskutierten 60-prozentigen Verlangsamung beim Cid-Ping-Pong steigt diese auf bis zu 75 % (struct-Marshalling) beziehungsweise 100 % (IPC-Stream-

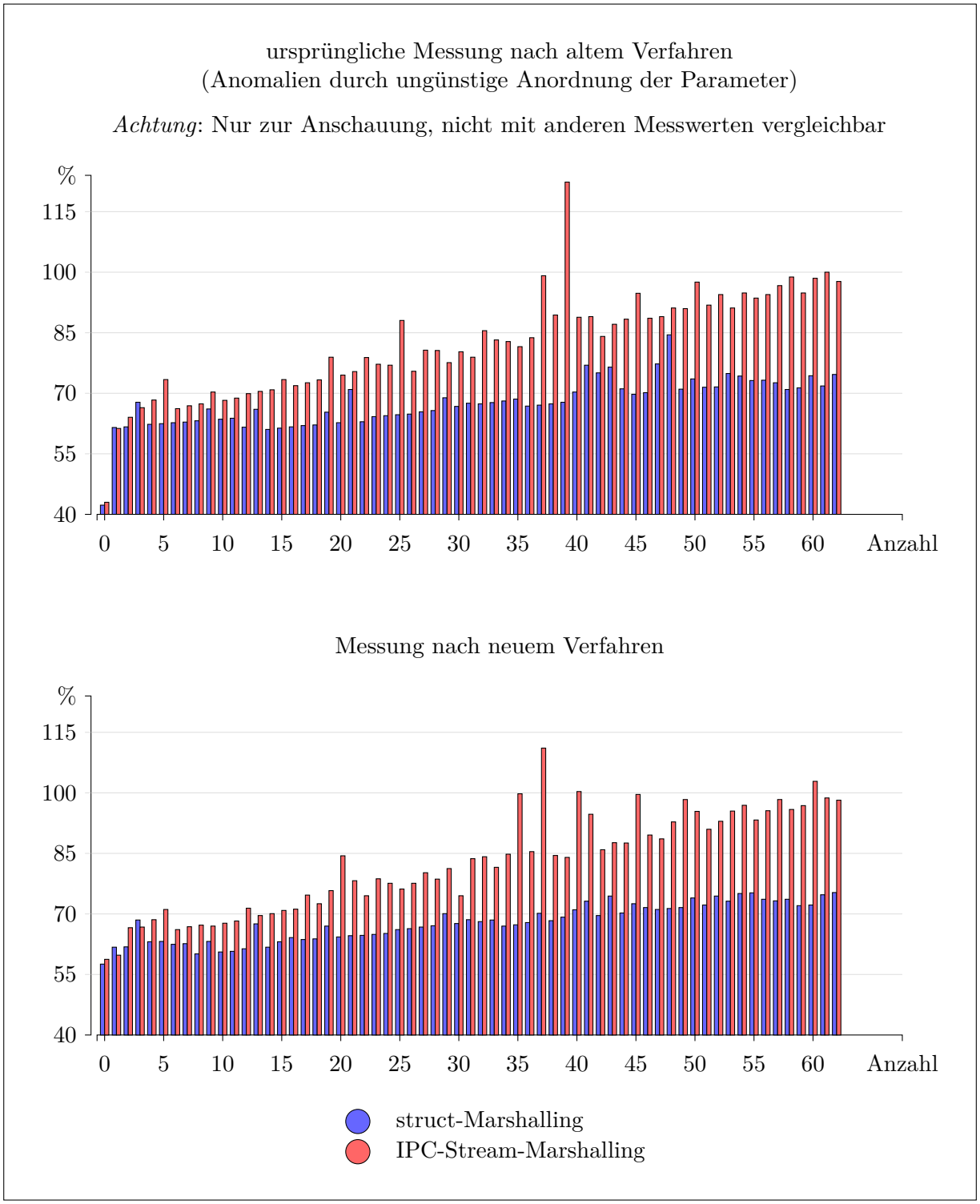


Abbildung 5.1.: Übertragung einfacher Datenwerte
Ordinate ... Anzahl übertragener `int`-Werte,
Abszisse ... Laufzeitverlängerung gegenüber L4Re-Ping-Pong

Marshalling) an. Weiterhin sind einige Anomalien zu beobachten, auf die Abschnitt 5.2.3 auf Seite 55 näher eingehen wird.

Die Messungen belegen einerseits die deutlich höhere Effizienz des struct-Marshalling, offenbaren andererseits aber auch einen unerwartet große Einfluss der beteiligte Infrastruktur auf die Gesamtlaufzeit eines RPC. Der Umfang der übertragenen Nutzdaten spielt dagegen nur eine untergeordnete Rolle.

5.2.2.2. Capabilitys

Die Capability-Übertragung (*„Cap“*) zeigt keinerlei Unterschiede zwischen den Marshalling-Methoden. In Anbetracht der Tatsache, dass Capabilitys von beiden Verfahren intern praktisch gleich verarbeitet werden, kann dieses Ergebnis nicht überraschen. Die Verlangsamung von etwa 100 % – also eine Verdopplung der Laufzeit gegenüber einem reinen L4Re-Ping-Pong und 40 Prozentpunkte über dem oben beschriebenen Basiswert des Cid-Ping-Pong – ergibt sich aus den bereits diskutierten allgemeinen Verlangsamungseffekten sowie der zur Verarbeitung der Capability im Mikrokern selbst notwendigen Zeit.

5.2.2.3. Arrays

Im Rahmen dieser Messreihe werden `int`-Arrays verschiedener Länge übertragen. Anhand der verwendeten Arrays können die Versuche in vier Gruppen unterteilt werden:

- statisch spezifizierte, variable Länge (*„ArrayStatVarLen“*)
- statisch spezifizierte, konstante Länge (*„ArrayStatFixLen“*)
- dynamisch spezifizierte, variable Länge (*„ArrayDynVarLen“*)
- dynamisch spezifizierte, konstante Länge (*„ArrayDynFixLen“*)

Wie ersichtlich, nutzen jeweils zwei Versuchsreihen Arrays mit statisch beziehungsweise dynamisch spezifizierter Länge (siehe auch Abschnitt 3.1.2). In jeder von diesen kommt einerseits eine zunehmende Anzahl von Arrays konstanter Länge (10 Elemente) zum Einsatz, zum anderen eine gleichbleibende Anzahl (10 Stück) kontinuierlich in ihrer Länge anwachsender Arrays.

Im Rahmen der ersten, zweiten, ... Messung aller Gruppen werden jeweils gleiche Mengen von Nutzdaten übertragen. Während die Art der Längenspezifikation (statisch oder dynamisch) diesbezüglich keinen Einfluss hat, ist die Größe der Nutzdaten in korrespondierenden Messungen mit variabel beziehungsweise konstant langen Arrays identisch. In Versuch 4 beispielsweise, werden in beiden Fällen je 40 Elemente übertragen – entweder in Form vierer Arrays mit je zehn Elementen (Arrays konstanter Länge) oder durch zehn Arrays mit je vier Elementen (Arrays variabler Länge).

Die soeben beschriebene Äquivalenz erstreckt sich ausschließlich auf die Nutzdaten, *nicht* auf die tatsächlich übertragenen Daten! Alle Arrays einer IDL-Funktion weisen im Rahmen der Tests stets dieselbe Länge auf, so dass auf Seiten des struct-Marshalling ein großes Optimierungspotential besteht; siehe auch Abschnitt 3.6. Das führt soweit, dass

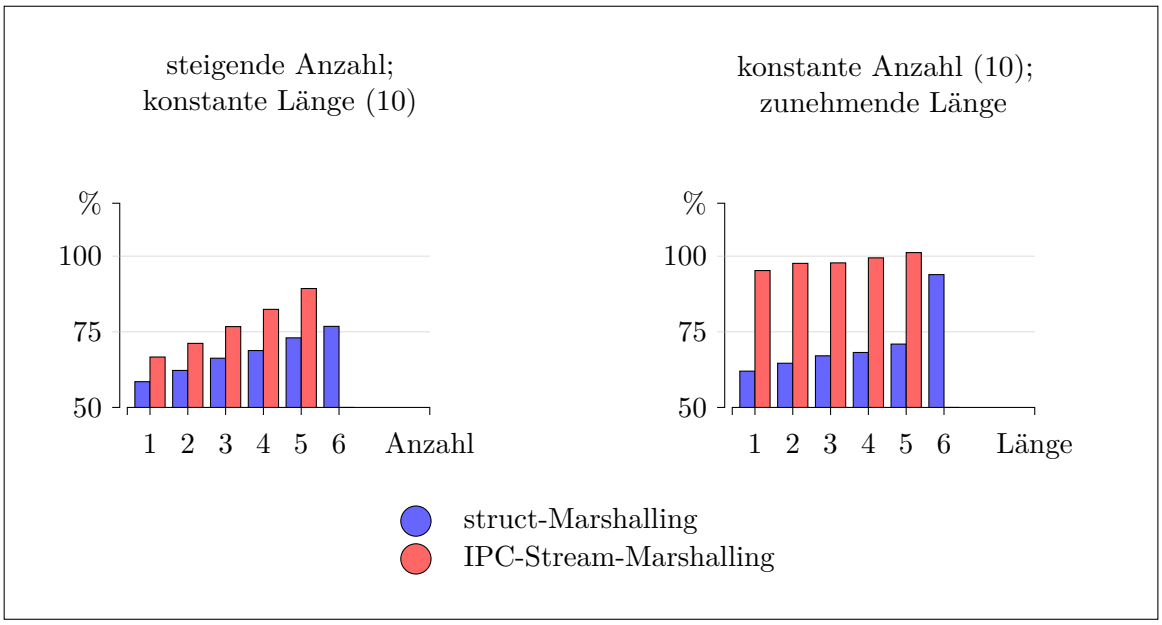


Abbildung 5.2.: Übertragung von `int`-Arrays mit statisch spezifizierter Länge
Ordinate ... Anzahl/Länge übertragener Arrays,
Abszisse ... Laufzeitverlängerung gegenüber L4Re-Ping-Pong

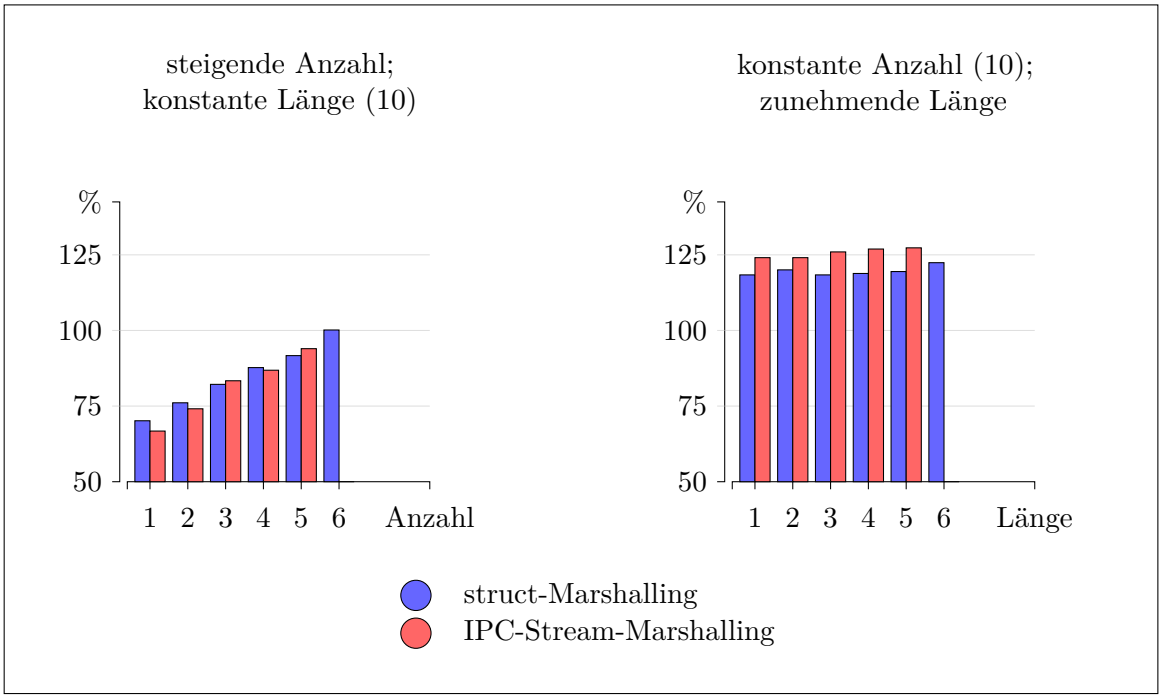


Abbildung 5.3.: Übertragung von `int`-Arrays mit dynamisch spezifizierter Länge
Ordinate ... Anzahl/Länge übertragener Arrays,
Abszisse ... Laufzeitverlängerung gegenüber L4Re-Ping-Pong

der jeweils letzte Versuch jeder Messreihe – eine Übertragung von 60 Datenelementen – ausschließlich unter Verwendung von struct-Marshalling durchführbar ist.

Ergebnisse Die in den Abbildungen 5.2 und 5.3 auf der vorherigen Seite dargestellten Messergebnisse entsprechen in vielen Punkten nicht den Erwartungen. Werden die Array-Längen statisch spezifiziert, zeigt sich noch das vertraute Bild: Das struct-Marshalling erreicht deutlich niedrigere Laufzeiten als das IPC-Stream-Marshalling – bis zu 30 Prozentpunkte beträgt dieser Vorsprung (Abbildung 5.2, rechte Seite).

Unerwarteterweise profitiert jedoch auch das IPC-Stream-Marshalling maßgeblich von einer statischen Spezifikation der Array-Länge, obwohl Cid in diesem Fall keine speziellen Optimierungen einbringen kann. Der Effekt nimmt darüber hinaus mit der Anzahl verarbeiteter Arrays deutlich zu: Bei zehn übertragenen Arrays (Messungen mit konstanter Anzahl; Abbildungen 5.2 und 5.3, jeweils auf der rechten Seite) fällt die Verlangsamung bei Verwendung statisch spezifizierter Array-Längen um bis zu 25 Prozentpunkte geringer aus als mit dynamisch spezifizierten Längen. Offenbar wirken sich hier Code-Optimierungen des C++-Compilers aus, der den Kommunikationscode letztlich verarbeitet.

Die dynamische Spezifikation der Array-Länge führt bei beiden Marshalling-Methoden zu einer signifikanten Zunahme der Laufzeiten. Beim struct-Marshalling ist dieser Effekt jedoch deutlich stärker ausgeprägt, so dass beide Verfahren sich unter diesen Bedingungen als praktisch gleichwertig erweisen. Insgesamt lässt sich beobachten, dass die Laufzeit hier deutlich vom Verarbeitungsaufwand für die einzelnen Arrays (Überprüfung auf Überlauf, Ausrichtung im Puffer) dominiert wird, während das übertragene Datenvolumen nur eine untergeordnete Rolle spielt.

5.2.2.4. Erweiterte Optimierungen

Die Auswirkungen der erweiterten Optimierungen (vgl. Abschnitt 3.7) sollen anhand zweier zusätzlicher Versuche untersucht werden: Zum einen ist natürlich auch an dieser Stelle der Unterschied zwischen beiden Marshalling-Methoden von Interesse, zum anderen aber auch der maximal mögliche Beschleunigungseffekt, der sich aus dem Verzicht auf eine serverseitige Pufferung eingehender Daten ergibt.

Für die Vergleichsmessung (*„Fast“*) fiel die Wahl auf eine Abwandlung der bereits oben beschriebenen Messreihe zu Arrays mit dynamisch spezifizierter Länge. Bei diesem Array-Typ fällt der Laufzeitunterschied zwischen beiden Marshalling-Methoden, der sich aus Cids allgemeinen Array-Optimierungen ergibt, recht klein aus. Um den (hier unerwünschten) Einfluss dieser einfachen Optimierungen weiter abzuschwächen, erhält jeder Array einen eigenen Längenparameter vom Typ `unsigned long`. Auf diese Weise kann im Rahmen des struct-Marshalling die übertragene Datenmenge nicht durch die Nutzung eines einzigen, gemeinsamen Längenparameters verringert werden. Gleichzeitig entfällt beim IPC-Stream-Marshalling der sonst erforderliche Zwischenschritt zur Typanpassung. Beide Effekte wurden in Abschnitt 3.6 bereits ausführlich erläutert.

Ein idealisierter Versuch (*„FastOutArrayStatVarLen“*) dient abschließend dazu, die maximal mögliche Beschleunigung zu ermitteln, die mittels erweiterter Optimierungen unter perfekten Bedingungen erreicht werden kann. Diese sind gegeben, wenn Arrays mit

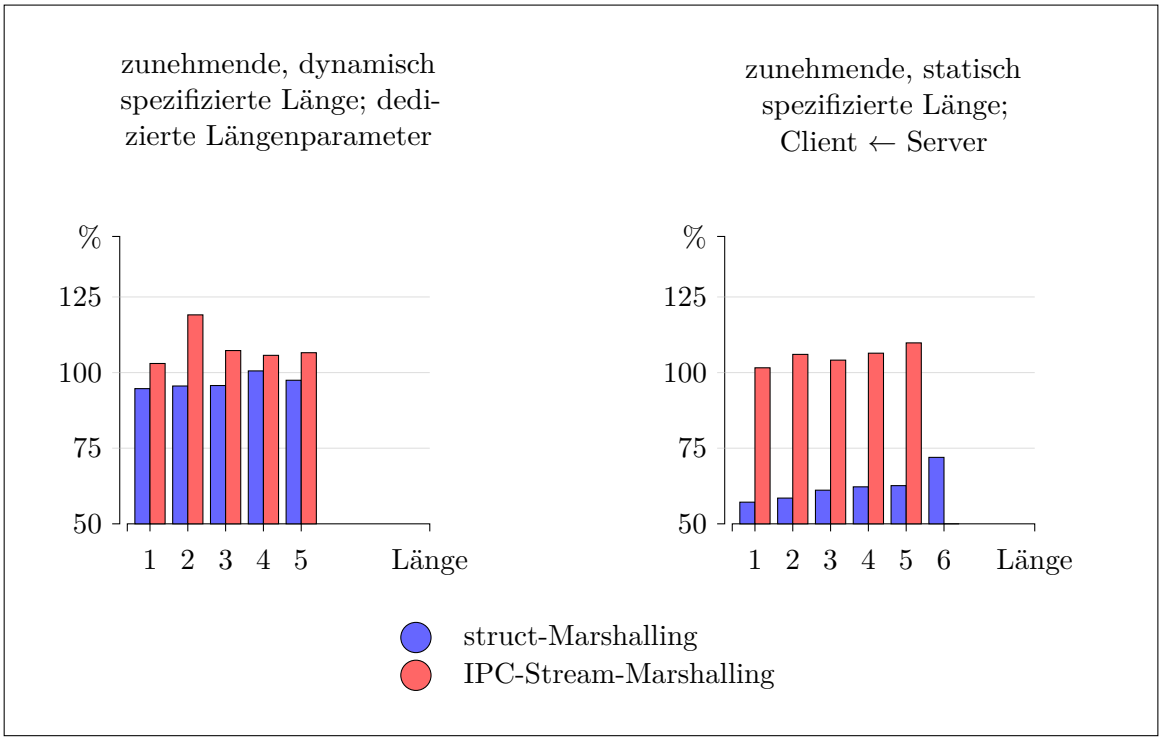


Abbildung 5.4.: Übertragung einer konstanten Anzahl (10) von `int`-Arrays unter Verwendung erweiterter Optimierungen
Ordinate ... Länge der übertragenen Arrays,
Abszisse ... Laufzeitverlängerung gegenüber L4Re-Ping-Pong

statisch spezifizierter Länge vom Server an den Client übertragen werden und struct-Marshalling zum Einsatz kommt: Unter Verzicht auf eine serverseitige Pufferung schreibt der benutzereigene Server-Code die Array-Daten dabei direkt in den IPC-Puffer von L4Re. Das Marshalling im Server reduziert sich dann auf die obligatorischen Überlaufprüfungen, umfasst aber keine Kopieroperationen. Aufgrund des eingesetzten Messverfahrens fließt bei den Versuchen der Kopieraufwand dennoch mit in die Laufzeit ein, so dass der tatsächliche Optimierungsgewinn selbst bei diesem Szenario unterschätzt wird; lediglich die verschlankten Funktionsaufrufe auf Serverseite schlagen sich voll in den gemessenen Werten nieder.

Ergebnisse Bei der Bewertung der Ergebnisse (dargestellt in Abbildung 5.4 auf dieser Seite) ist zu beachten, dass beide Versuche sich nicht nur durch die zusätzlichen Optimierungen von ihren jeweiligen Vorbildern unterscheiden: Im ersten Szenario wurden die Längenparameter verändert (Typ und gemeinsame Nutzung), im zweiten die Übertragungsrichtung der Nutzdaten. Letzteres führt dazu, dass nun Daten sowohl vom Client an den Server (Opcode) als auch in der Gegenrichtung (Array-Daten) übertragen werden.

Wie bereits erwartet, führt der Einsatz von Cids erweiterten Optimierungen im Allge-

meinen zu kürzeren RPC-Laufzeiten als bei den Messungen ohne diese Optimierungen. Für das IPC-Stream-Marshalling fällt die Laufzeitverkürzung mit rund 20 Prozentpunkten etwa genauso hoch aus, wie die bereits diskutierte Verbesserung durch Verwendung statischer Längenspezifikation. Im idealisierten Szenario dagegen kommt es teilweise sogar zu Verlangsamungen, die vermutlich auf die geänderte Übertragungsrichtung zurückgehen: Die von Cid durchgeführten erweiterten Optimierungen sind für IPC-Streams nicht anwendbar, so dass nur die zusätzliche Verwendung des IPC-Puffers zum Tragen kommt.

Noch deutlich bessere Ergebnisse lassen sich mittels struct-Marshalling erzielen: Im ersten Szenario beträgt die Verbesserung 25 Prozentpunkte. Das zweite, idealisierte Szenario zeigt eine Verbesserung von nahezu 10 Prozentpunkten gegenüber den ohnehin bereits guten Laufzeiten im Rahmen der Messungen zu Arrays mit statisch spezifizierter Länge. Wie zuvor festgestellt, wird hier die Laufzeit maßgeblich vom Nutzdatenvolumen bestimmt, so dass der Beschleunigungseffekt mit zunehmender Array-Länge stärker zum Tragen kommt.

5.2.3. Anomalien

Wie in Abschnitt 5.2.1 erwähnt, zeigten sich bereits kurz nach Beginn der Messungen zahlreiche Anomalien in den Werten. Zur Verdeutlichung zeigt die obere Hälfte von Abbildung 5.1 auf Seite 50 diese ersten Ergebnisse beispielhaft für die Messungen mit einfachen Datenwerten. Da zu dieser Zeit noch das ursprüngliche, sehr einfach Messverfahren zum Einsatz kam, sind diese Werte *nicht* mit den anderen Messwerten vergleichbar!

Sofort fällt der extrem starke Anstieg im zweiten Versuch (Übertragung eines einzigen `int`-Wertes) gegenüber dem ersten (keine Nutzdatenübertragung) auf. Ein derart sprunghafter Zuwachs um etwa 20 Prozentpunkte ist ebenso wenig erklärbar wie die Spitzen im weiteren Verlauf der Messreihe (Versuche mit 21, 25, 37, 39 sowie 47 und 48 Datenwerten).

Mit Hilfe des erweiterten Messverfahrens und zusätzlichen Versuche ließ sich ein Teil der Anomalien auf die Anordnung der übertragenen Daten im IPC-Puffer zurückführen: Weicht die Verarbeitungsreihenfolge der Parameter einer IDL-Funktion von der Reihenfolge ihres Auftretens in der Funktionssignatur ab, hat dies offenbar in einigen Fällen eine erhebliche Laufzeitverlängerung zur Folge; siehe dazu auch Abschnitt 3.7.

Entsprechenden Anpassungen der Kommunikationscode-Erzeugung führten zwar insgesamt zu einer Verringerung des Effekts, jedoch treten in mehreren Versuchen weiterhin unerklärliche Laufzeitspitzen und/oder -einbrüche auf. Auffällig sind insbesondere die Messungen mit 20, 35, 37, 40, 41 und 45 einfachen Datenwerten (untere Hälfte von Abbildung 5.1 auf Seite 50), Versuch 6 zu Arrays mit konstanter, statisch spezifizierter Länge (rechte Hälfte von Abbildung 5.2 auf Seite 52) sowie Versuch 2 zu erweiterten Optimierungen (rechte Hälfte von Abbildung 5.4 auf der vorherigen Seite). Außer den teilweise stark erhöhten Laufzeiten zeigen sie allerdings keine Auffälligkeiten – die ermittelte Standardabweichung unterscheidet sich nicht signifikant von der anderer Messungen und Wiederholungen der abweichenden Versuche führen zu den gleichen Ergebnissen. Für die Analysen in den vorangegangenen Abschnitten wurden diese fehlerbehafteten

Werte nicht mit herangezogen.

Die Ursachen der verbleibenden Anomalien konnten im Rahmen dieser Arbeit nicht abschließend ermittelt werden. Jedoch tritt ein ähnliches Phänomen auch bei den UTCB-IPC-Tests des L4Re-Ping-Pong-Benchmarks auf, so dass die Ursache allem Anschein nach im IPC-Mechanismus von Fiasco.OC selbst zu suchen ist.

5.2.4. Bewertung

Die vorliegenden Messergebnisse belegen, dass die Verwendung von Cid unter L4Re durchaus sinnvoll ist. Natürlich schreckt die mindestens 50-prozentige Verlangsamung eines RPC gegenüber dem reinen L4Re-Ping-Pong-Benchmark zunächst ab. Allerdings geht dieser Effekt mit der Infrastruktur einher, die unter L4Re im Allgemeinen für RPCs genutzt wird. Er tritt mit manuell erstelltem IPC-Stream-Code in ähnlichem Maße auf und spricht daher nicht gegen den Einsatz eines IDL-Compilers.

Gleichzeitig ermöglicht Cid die praktische Nutzung von Verfahren wie struct-Marshalling: Während die manuelle Erstellung des entsprechenden Kommunikationscodes sehr aufwendig ist, sind teilweise beachtliche Verbesserungen der RPC-Laufzeit möglich. Insbesondere bei der Übertragung einfacher Datenwerte oder Arrays mit statisch spezifizierter Länge werden die Vorteile deutlich sichtbar. Die von Cid bereitgestellten erweiterten Optimierungen führen unter günstigen Umständen zu weiteren Verbesserung.

6. Zusammenfassung und Ausblick

Die vorliegende Arbeit zeigt, dass die Entwicklung und Implementierung eines IDL-Compilers durch Wiederverwendung eines bereits vorhandenen Hochsprachen-Compilers sowohl vereinfacht als auch beschleunigt werden kann (Abschnitt 4.1). Trotz seiner verhältnismäßig kurzen Entwicklungszeit erfüllt Cid dabei alle wesentlichen Anforderungen an einen IDL-Compiler für L4Re:

- Die Bedienung erfolgt vollständig über die Kommandozeile und ohne Interaktion mit dem Benutzer. Der IDL-Compiler fügt sich damit nahtlos in den für L4Re üblichen skriptgesteuerten Übersetzungsvorgang ein.

- Die Ähnlichkeit zwischen der weitverbreiteten Programmiersprache C++ und der neu geschaffenen IDL erleichtert neuen Benutzern des IDL-Compilers die Einarbeitung. Auch die Erzeugung von Kommunikationscode ist darauf ausgelegt, eine zuverlässige Ausführung grundsätzlich der bestmöglichen Laufzeit vorzuziehen.

- Die von Cid verarbeitete IDL (Abschnitt 3.1.2) ist flexibel genug, um alle Kommunikationsmodelle abzubilden, die in einem Client-Server-Umfeld wie L4Re regelmäßig auftreten. Mechanismen wie die Vererbung von Cid-Attributen oder die direkte Einbindung von Typdeklarationen aus vorhandenem C-/C++-Code unterstützen den Benutzer zusätzlich.

- Bei der Zuweisung von Protokoll-IDs und Opcodes ist neben der manuellen auch eine vollautomatische Vergabe vorgesehen (Abschnitt 3.1.2) – im Regelfall bleibt dem Benutzer diese Routineaufgabe somit erspart.

- Der aus einer IDL-Spezifikation erzeugte Kommunikationscode ist manuell geschriebenem Code in puncto Leistungsfähigkeit ebenbürtig beziehungsweise überlegen (Abschnitt 5.2.4). Dabei lassen die beiden von Cid unterstützten Marshalling-Methoden dem Benutzer die Wahl zwischen verbesserter Laufzeit (struct-Marshalling) und vertrautem, einfach lesbarem Quellcode (IPC-Stream-Marshalling). Optionale, erweiterte Optimierungen (Abschnitt 3.7) können die Ausführungsgeschwindigkeit noch weiter verbessern.

- Da Cid sowohl in Form eines Clang-Plug-ins als auch in einer eigenständigen Version bereitgestellt wird (Abschnitt 3.2), kann diejenige Variante gewählt werden, die für das jeweilige Umfeld am besten geeignet ist. Die konsistente Bedienung beider Programmversionen ermöglicht dabei einen einfachen Wechsel zwischen ihnen.

6.1. Ausblick

Der aktuelle Entwicklungsstand von Cid bietet zahlreiche Anknüpfpunkte für weiterführende Arbeiten. Neben der Unterstützung neuer Zielsprachen, erscheint vor allem die Ergänzung zusätzlicher Marshalling-Methoden (beispielsweise die Nutzung gemeinsamen Speichers zur Datenübermittlung) oder Optimierungstechniken lohnenswert. Der IDL-Compiler kann so gegebenenfalls leistungsfähigeren Kommunikationscode erzeugen und darüber hinaus an spezifische Anforderungen angepasst werden.

Des Weiteren kommt eine Weiterentwicklung der IDL selbst in Frage. Diese kann von der Einführung neuer Cid-Attribute bis hin zur ursprünglich angedachten, vollständigen Einbettung der IDL-Spezifikation in den normalen Programmcode reichen.

Letztlich bietet auch der erzeugte Kommunikationscode Raum für Erweiterungen. So ist zum Beispiel eine Berücksichtigung von Ausnahmen im benutzereigenen Servercode denkbar, die dann an den Client übermittelt und dort erneut ausgelöst werden. Durch diese für den Benutzer transparente Weiterleitung tritt die Client-Server-Struktur eines auf L4Re-aufbauenden Systems noch weiter in den Hintergrund; die Programmerstellung vereinfacht sich.

Abgesehen von Verbesserungen am IDL-Compiler bieten die Laufzeitanomalien, die im Rahmen der Laufzeitmessungen (Abschnitt 5.2.3) festgestellt wurden, Anlass für eingehende Untersuchungen. Nach dem aktuellen Kenntnisstand werden sie vom Fiasco.OC-Mikrokern beziehungsweise L4Re verursacht.

Glossar

AST
: *„Abstract Syntax Tree“* (abstrakter Syntaxbaum) – Repräsentation der syntaktischen Struktur einer Übersetzungseinheit (in der Regel eine Quellcodedatei, sowie per Präprozessor eingebundene Dateien) innerhalb eines Compilers; dient der Datenübergabe von Front-End an Back-End und damit deren Entkopplung.

Back-End
: Teil eines Compilers, der Optimierung und Erzeugung des Zielcodes umfasst; Ausgangspunkt ist eine interne Repräsentation des Quellcodes (AST), das Ergebnis liegt in der Zielsprache vor.

Cid
: *„Clang-basierter IDL-Compiler“* – IDL-Compiler für L4Re, der im Rahmen dieser Arbeit entwickelt wurde.

Clang
: Eigenständig entwickeltes Front-End für LLVM zur Verarbeitung von C, C++ und Objective-C.

Front-End
: Teil eines Compilers, der lexikalische und semantische Analyse des Quelltextes umfasst; erzeugt typischerweise eine interne Repräsentation (AST) des Ausgangsmaterials.

IDL
: *„Interface Definition Language“* (Schnittstellenbeschreibungssprache) – Spezielle Sprache zur Spezifikation der Schnittstellen einer Softwarekomponente; von einem IDL-Compiler verarbeitet, um Kommunikationscode in einer oder mehreren Hochsprache(n) zu erzeugen.

IPC
: *„Inter-Process Communication“* (Inter-Prozess-Kommunikation) – Durch das Betriebssystemkern bereitgestellter Mechanismus zum Datenaustausch zwischen den Threads eines beziehungsweise mehrerer Prozesse.

Marshalling
: Vorbereiten der zu übertragenden Daten und geeignetes Ablegen im Transportpuffer des verwendeten Kommunikationssystems; siehe auch Unmarshalling und RPC.

Opcode
: Eindeutige Kennzeichnung einer Funktion innerhalb der Schnittstelle eines Servers; in aller Regel eine einfache Ganzzahl.

RPC
: *„Remote Procedure Call“* (Fernaufruf einer Prozedur/Funktion) – Mechanismus zur Nutzung einer Funktion, die durch einen Server bereitgestellt wird; erforderliche Datenübertragungen zwischen Client und Server sind Teil des RPC.

SLOC
: *„Source Lines of Code“* (Quellcodezeilen) – Maß für Umfang/Komplexität eines Programms; alle Angaben im Rahmen dieser Arbeit erhoben mit David A. Wheelers „SLOCCount“, `http://www.dwheeler.com/sloccount/`.

TCB
: *„Trusted Computing Base“* (vertrauenswürdige Basis) – Gesamtheit aller Systemkomponenten (sowohl Hard- als auch Software), deren fehlerfreie Funktion gewährleistet sein muss, um die Sicherheit des Systems als Ganzes zu garantieren.

Unmarshalling
: Auslesen des Transportpuffer und Rekonstruktion der ursprünglichen Struktur der enthaltenen Daten; siehe auch Marshalling und RPC.

Akronyme

DICE
„Dresden IDL Compiler“.

DROPS
„Dresden Real-Time Operating System“.

GCC
„GNU Compiler Collection“.

L4Re
„L4 Runtime Environment“.

LLVM
„Low Level Virtual Machine“.

RTTI
„Runtime Type Information“.

UTCB
„User Level Thread Control Block“.

Literaturverzeichnis

[Aig01] Aigner, Ronald: *Development of an IDL Compiler for Micro-Kernel based Components*, TU Dresden, Diplomarbeit, 2001. – http://os.inf.tu-dresden.de/papers_ps/aigner-diplom.pdf

[Bie] Bierbaum, Jan: *Cid – User Manual*

[claa] *Clang: a C language family frontend for LLVM.* – http://clang.llvm.org/

[clab] *Clang API documentation.* – http://clang.llvm.org/doxygen/

[clac] *Clang mailing list.* – http://lists.cs.uiuc.edu/pipermail/cfe-dev/

[CYC+01] Chou, Andy ; Yang, Junfeng ; Chelf, Benjamin ; Hallem, Seth ; Engler, Dawson: An empirical study of operating systems errors. In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, S. 73–88. – http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.21.6489&rep=rep1&type=pdf

[dic] *DICE.* – http://os.inf.tu-dresden.de/dice/

[Fes07] Feske, Norman: A case study on the cost and benefit of dynamic RPC marshalling for low-level system components. In: *SIGOPS Oper. Syst. Rev.* 41 (2007), July, S. 40–48. – ISSN 0163–5980. – http://os.inf.tu-dresden.de/papers_ps/feske07-dynrpc.pdf

[fia] *The Fiasco.OC μ-kernel.* – http://os.inf.tu-dresden.de/fiasco/

[gcca] *GCC, the GNU Compiler Collection.* – http://gcc.gnu.org/

[gccb] *GCC documentation.* – http://gcc.gnu.org/onlinedocs/

[HHL+97] Härtig, Hermann ; Hohmuth, Michael ; Liedtke, Jochen ; Wolter, Jean ; Schönberg, Sebastian: The performance of μ-kernel-based systems. In: *Proceedings of the 16th ACM symposium on Operating systems principles*, 1997 (SOSP '97), S. 66–77. – http://os.inf.tu-dresden.de/papers_ps/sosp97.pdf

[ISO03] ISO: *ISO/IEC 14882:2003: Programming languages — C++.* 2003. – 757 S.

[KEH+09] Klein, Gerwin ; Elphinstone, Kevin ; Heiser, Gernot ; Andronick, June ; Cock, David ; Derrin, Philip ; Elkaduwe, Dhammika ; Engelhardt, Kai ; Kolanski, Rafal ; Norrish, Michael ; Sewell, Thomas ; Tuch, Harvey ;

Winwood, Simon: seL4: formal verification of an OS kernel. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ACM, 2009, S. 207–220. – http://www.cse.unsw.edu.au/~kleing/papers/sosp09.pdf

[Lie95] Liedtke, Jochen: On μ-kernel construction. In: *Symposium on Operating System Principles*, ACM, 1995. – http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.26.4581&rep=rep1&type=pdf

[Lie96a] Liedtke, Jochen: μ-Kernels Must And Can Be Small. In: *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, IEEE, 1996, S. 152–155. – http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.2355&rep=rep1&type=pdf

[Lie96b] Liedtke, Jochen: *Toward Real Microkernels*. 1996. – http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.25.9120&rep=rep1&type=pdf

[llva] *The LLVM Compiler Infrastructure*. – http://llvm.org/

[llvb] *LLVM API Documentation*. – http://llvm.org/doxygen/

[LW09] Lackorzynski, Adam ; Warg, Alexander: Taming subsystems: capabilities as universal resource access control in L4. In: *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, 2009 (IIES '09), S. 25–30. – http://os.inf.tu-dresden.de/papers_ps/lackorzynskiwarg09_iies09-taming-subsys.pdf

A. Leistungsmessungen

Dieser Teil des Anhangs enthält weiterführendes Material zu den Leistungsmessungen, die in Kapitel 5 vorgestellt wurden. Die hier bereitgestellten Informationen ermöglichen weitere Auswertungen der bereits vorliegenden Ergebnisse und erleichtern eine Wiederholung der durchgeführten Versuche.

A.1. Quellcode

Allen Messungen liegt eine IDL-Spezifikation der verwendeten Schnittstellen zugrunde, die aus Platzgründen im Folgenden nur auszugsweise wiedergegeben wird: Zur Verdeutlichung des Schemas, dem jede Messreihe folgt, sind (sofern vorhanden) jeweils die ersten beiden Versuche enthalten. Es folgt der Code zur clientseitigen Zeiterfassung; ebenfalls auf die wesentlichen Teile reduziert.

Die vollständige IDL-Spezifikation ist, wie auch Client- und Server-Quellcode, in die PDF-Version dieser Arbeit eingebettet. Sie stehen damit als Ausgangspunkt für weitere Versuche direkt zur Verfügung.

IDL-Spezifikation (verkürzt)

```
1  #include <l4/re/dataspace>
2  #include <l4/sys/capability>
3
4  namespace IDL IN {
5
6  struct PingPong {
7    void go();
8  };
9
10 struct Simple {
11   void go_1(int val_1);
12   void go_2(int val_1,int val_2);
13   [...]
14 };
15
16 struct Cap {
17   void go(L4::Cap<L4Re::Dataspace> CAP val);
18 };
19
20 struct ArrayStatFixLen {
21   void go_1(int val_1 [] MAX(10) LEN(10));
22   void go_2(int val_1 [] MAX(10) LEN(10),int val_2 [] MAX(10) LEN(10));
23   [...]
24 };
25
26 struct ArrayStatVarLen {
```

```
27    void go_1(int val_1 [] MAX(1) LEN(1),int val_2 [] MAX(1) LEN(1),int
         val_3 [] MAX(1) LEN(1),int val_4 [] MAX(1) LEN(1),int val_5 [] MAX
         (1) LEN(1),int val_6 [] MAX(1) LEN(1),int val_7 [] MAX(1) LEN(1),
         int val_8 [] MAX(1) LEN(1),int val_9 [] MAX(1) LEN(1),int val_10 []
          MAX(1) LEN(1));
28    void go_2(int val_1 [] MAX(2) LEN(2),int val_2 [] MAX(2) LEN(2),int
         val_3 [] MAX(2) LEN(2),int val_4 [] MAX(2) LEN(2),int val_5 [] MAX
         (2) LEN(2),int val_6 [] MAX(2) LEN(2),int val_7 [] MAX(2) LEN(2),
         int val_8 [] MAX(2) LEN(2),int val_9 [] MAX(2) LEN(2),int val_10 []
          MAX(2) LEN(2));
29    [...]
30  };
31
32  struct FastOutArrayStatVarLen {
33    void go_1(int val_1 [] MAX(1) LEN(1),int val_2 [] MAX(1) LEN(1),int
         val_3 [] MAX(1) LEN(1),int val_4 [] MAX(1) LEN(1),int val_5 [] MAX
         (1) LEN(1),int val_6 [] MAX(1) LEN(1),int val_7 [] MAX(1) LEN(1),
         int val_8 [] MAX(1) LEN(1),int val_9 [] MAX(1) LEN(1),int val_10 []
          MAX(1) LEN(1));
34    void go_2(int val_1 [] MAX(2) LEN(2),int val_2 [] MAX(2) LEN(2),int
         val_3 [] MAX(2) LEN(2),int val_4 [] MAX(2) LEN(2),int val_5 [] MAX
         (2) LEN(2),int val_6 [] MAX(2) LEN(2),int val_7 [] MAX(2) LEN(2),
         int val_8 [] MAX(2) LEN(2),int val_9 [] MAX(2) LEN(2),int val_10 []
          MAX(2) LEN(2));
35    [...]
36  } OUT FAST;
37
38  struct ArrayDynVarLen {
39    void go_1(int val_1 [] MAX(1) LEN(len),int val_2 [] MAX(1) LEN(len),
         int val_3 [] MAX(1) LEN(len),int val_4 [] MAX(1) LEN(len),int val_5
          [] MAX(1) LEN(len),int val_6 [] MAX(1) LEN(len),int val_7 [] MAX
         (1) LEN(len),int val_8 [] MAX(1) LEN(len),int val_9 [] MAX(1) LEN(
         len),int val_10 [] MAX(1) LEN(len), unsigned len = 1);
40    void go_2(int val_1 [] MAX(2) LEN(len),int val_2 [] MAX(2) LEN(len),
         int val_3 [] MAX(2) LEN(len),int val_4 [] MAX(2) LEN(len),int val_5
          [] MAX(2) LEN(len),int val_6 [] MAX(2) LEN(len),int val_7 [] MAX
         (2) LEN(len),int val_8 [] MAX(2) LEN(len),int val_9 [] MAX(2) LEN(
         len),int val_10 [] MAX(2) LEN(len), unsigned len = 2);
41    [...]
42  };
43
44  struct Fast {
45    void go_1(int val_1 [] MAX(1) LEN(len_1),int val_2 [] MAX(1) LEN(len_2
         ),int val_3 [] MAX(1) LEN(len_3),int val_4 [] MAX(1) LEN(len_4),int
          val_5 [] MAX(1) LEN(len_5),int val_6 [] MAX(1) LEN(len_6),int
         val_7 [] MAX(1) LEN(len_7),int val_8 [] MAX(1) LEN(len_8),int val_9
          [] MAX(1) LEN(len_9),int val_10 [] MAX(1) LEN(len_10),unsigned
         long len_1 = 1,unsigned long len_2 = 1,unsigned long len_3 = 1,
         unsigned long len_4 = 1,unsigned long len_5 = 1,unsigned long len_6
          = 1,unsigned long len_7 = 1,unsigned long len_8 = 1,unsigned long
         len_9 = 1,unsigned long len_10 = 1);
46    void go_2(int val_1 [] MAX(2) LEN(len_1),int val_2 [] MAX(2) LEN(len_2
         ),int val_3 [] MAX(2) LEN(len_3),int val_4 [] MAX(2) LEN(len_4),int
          val_5 [] MAX(2) LEN(len_5),int val_6 [] MAX(2) LEN(len_6),int
         val_7 [] MAX(2) LEN(len_7),int val_8 [] MAX(2) LEN(len_8),int val_9
```

```
      [] MAX(2) LEN(len_9),int val_10 [] MAX(2) LEN(len_10),unsigned
     long len_1 = 2,unsigned long len_2 = 2,unsigned long len_3 = 2,
     unsigned long len_4 = 2,unsigned long len_5 = 2,unsigned long len_6
      = 2,unsigned long len_7 = 2,unsigned long len_8 = 2,unsigned long
     len_9 = 2,unsigned long len_10 = 2);
47   [...]
48 } FAST;
49 }
```

Code zur Messung der Laufzeit der IDL-Funktion `func`

```
1  l4_uint32_t start, end, dur, min, max;
2  double M, S, oldM;
3  int m_n;
4
5  [...]
6
7  M = S = 0;
8  min = 1e6;
9  max = 0;
10
11 for(m_n = 1; m_n < 100001; ++m_n) {
12   func(...); /* warm up */
13
14   start = l4_rdtsc_32();
15   func(...);
16   end = l4_rdtsc_32();
17
18   if (end < start) {m_n--; continue;} /* catch overflows */
19   dur = end - start;
20   if (dur > max) max = dur;
21   if (dur < min) min = dur;
22
23   /* calculate mean and variance using Welford's method */
24   oldM = M;
25   M += (dur - oldM) / m_n;
26   S += (dur - oldM) * (dur - M);
27 }
28 printf("mean=%.0f\tvar=%.0f\tmin=%u\tmax=%u\n", M, S / (m_n - 1), min,
     max)
```

A.2. Messwerte

Die Tabellen A.1 und A.2 enthalten alle Daten, die im Verlauf der Leistungsmessungen erhoben wurden. Da es sich um Rohdaten handelt, sind alle Laufzeiten in Taktzyklen angegeben – zum Vergleich: Ein vollständiger Lauf des Ping-Pong-Benchmarks von L4Re benötigte auf dem Testsystem im Mittel 1.263 Takte. Weitere Informationen zu Testsystem und Messungen sind in Kapitel 5 zu finden.

Der Tabelleneintrag „IPC-Worte“ bezeichnet die Anzahl der Elemente des IPC-Puffers, die im Rahmen des jeweiligen Versuchs zur Datenübertragung genutzt wurden. Die Spaltenbezeichnung geht darauf zurück, dass die Pufferelemente von Fiasco.OC durch vorzeichenbehaftete Maschinenworte realisiert werden.

Zum besseren Verständnis sind für alle Array-Messungen (Tabelle A.2 auf Seite xii) darüber hinaus auch Anzahl und Länge aller übertragenen Arrays angegeben. Der Eintrag „—“ verweist auf Messungen, die technisch nicht durchführbar waren, da die Kapazität des IPC-Puffers von maximal 63 Elementen für sie nicht ausreicht.

Die in den Tabellen auftretenden Bezeichnungen für die verschiedenen Messreihen entsprechen dem Namen der jeweiligen Schnittstelle/IDL-Klasse in der zugrundeliegenden IDL-Spezifikation (siehe Anhang A.1). Sie wurden in Abschnitt 5.2.2 im Zusammenhang mit der Beschreibung der einzelnen Versuche vorgestellt.

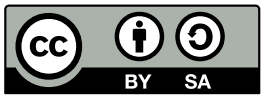
	Marshalling via struct					Marshalling via IPC-Stream				
Versuch	IPC-Worte	μ [T]	σ [T]	Min [T]	Max [T]	IPC-Worte	μ [T]	σ [T]	Min [T]	Max [T]
Pingpong	1	1.990	48,98	1.978	5.050	1	2.005	47,67	1.987	3.805
Capability	1	2.521	57,18	2.516	6.416	1	2.521	56,12	2.516	6.617
Simple_1	2	2.043	51,04	2.040	4.970	2	2.018	57,69	2.008	4.902
Simple_2	3	2.044	47,06	2.040	3.737	3	2.104	53,16	2.091	5.945
Simple_3	4	2.128	53,96	2.123	5.050	4	2.106	50,15	2.094	5.050
Simple_4	5	2.060	47,98	2.055	3.905	5	2.129	59,72	2.114	5.018
Simple_5	6	2.061	51,60	2.055	5.120	6	2.161	51,29	2.141	5.067
Simple_6	7	2.052	51,81	2.046	5.934	7	2.098	49,09	2.085	5.008
Simple_7	8	2.054	55,74	2.049	5.064	8	2.107	74,48	2.091	4.964
Simple_8	9	2.022	50,92	2.011	4.990	9	2.112	5,66	2.111	3.660
Simple_9	10	2.061	51,94	2.055	4.932	10	2.109	51,58	2.082	5.041
Simple_10	11	2.028	53,36	2.017	4.946	11	2.118	55,00	2.088	5.174
Simple_11	12	2.030	63,91	2.017	4.941	12	2.125	65,25	2.094	5.052
Simple_12	13	2.038	48,39	2.026	3.935	13	2.165	55,05	2.141	5.998
Simple_13	14	2.116	50,93	2.108	4.982	14	2.142	53,43	2.123	5.003
Simple_14	15	2.043	51,13	2.040	5.002	15	2.148	51,83	2.126	5.023
Simple_15	16	2.060	52,27	2.055	5.129	16	2.158	52,15	2.135	5.195
Simple_16	17	2.073	54,18	2.064	5.159	17	2.162	54,91	2.138	5.074
Simple_17	18	2.067	50,47	2.061	5.136	18	2.206	19,92	2.194	5.822
Simple_18	19	2.069	50,23	2.061	3.994	19	2.179	56,45	2.153	5.446
Simple_19	20	2.109	51,07	2.094	5.130	20	2.220	55,79	2.206	5.215
Simple_20	21	2.075	47,44	2.067	3.743	21	2.329	78,47	2.321	5.221
Simple_21	22	2.079	71,99	2.070	5.088	22	2.251	65,64	2.224	4.997
Simple_22	23	2.080	49,58	2.070	3.849	23	2.204	55,95	2.185	5.076
Simple_23	24	2.083	52,22	2.073	5.150	24	2.257	53,44	2.247	5.330
Simple_24	25	2.086	49,68	2.076	4.991	25	2.243	54,37	2.218	5.928
Simple_25	26	2.098	54,29	2.082	4.990	26	2.225	53,94	2.203	5.141
Simple_26	27	2.101	52,15	2.088	4.979	27	2.243	55,99	2.215	5.304
Simple_27	28	2.106	47,10	2.094	3.965	28	2.276	55,39	2.256	5.360
Simple_28	29	2.110	53,94	2.094	5.209	29	2.256	53,42	2.241	5.138
Simple_29	30	2.148	50,76	2.138	5.227	30	2.289	55,29	2.268	6.965
Simple_30	31	2.117	49,62	2.114	5.195	31	2.204	54,52	2.262	5.307
Simple_31	32	2.129	68,27	2.123	5.032	32	2.320	71,72	2.289	12.055
Simple_32	33	2.123	55,56	2.117	5.046	33	2.326	76,66	2.321	5.245
Simple_33	34	2.128	54,79	2.123	6.599	34	2.293	55,17	2.271	5.168
Simple_34	35	2.109	53,91	2.094	4.973	35	2.334	58,22	2.327	6.865
Simple_35	36	2.113	50,44	2.111	3.844	36	2.523	51,56	2.496	4.290
Simple_36	37	2.120	51,64	2.117	5.189	37	2.342	55,50	2.318	6.718
Simple_37	38	2.149	54,97	2.141	4.018	38	2.666	56,99	2.655	4.411
Simple_38	39	2.126	50,88	2.120	5.014	39	2.330	55,38	2.321	5.419
Simple_39	40	2.137	49,83	2.129	5.026	40	2.324	53,72	2.306	5.230
Simple_40	41	2.160	54,35	2.150	5.062	41	2.530	54,43	2.502	6.910
Simple_41	42	2.187	70,92	2.182	5.934	42	2.459	75,56	2.434	11.820
Simple_42	43	2.142	52,35	2.135	5.038	43	2.348	53,86	2.327	5.257
Simple_43	44	2.203	60,00	2.197	5.271	44	2.370	58,81	2.357	5.298
Simple_44	45	2.150	53,16	2.141	5.067	45	2.369	55,54	2.348	5.387
Simple_45	46	2.179	56,68	2.176	5.121	46	2.521	56,79	2.483	6.211
Simple_46	47	2.167	54,82	2.156	5.204	47	2.394	55,02	2.389	5.366
Simple_47	48	2.161	55,99	2.150	5.168	48	2.382	52,96	2.357	5.251
Simple_48	49	2.164	56,28	2.153	5.331	49	2.435	58,36	2.422	7.155
Simple_49	50	2.167	52,21	2.156	4.142	50	2.505	55,35	2.493	4.456
Simple_50	51	2.197	48,17	2.191	5.813	51	2.468	55,38	2.460	5.605
Simple_51	52	2.175	72,95	2.162	5.088	52	2.412	65,35	2.392	5.508
Simple_52	53	2.203	50,63	2.197	3.929	53	2.437	57,45	2.425	5.496
Simple_53	54	2.187	51,43	2.182	5.124	54	2.469	57,30	2.463	7.143
Simple_54	55	2.211	52,44	2.203	5.283	55	2.487	56,86	2.478	6.371
Simple_55	56	2.213	11,49	2.206	5.265	56	2.441	55,47	2.416	4.304
Simple_56	57	2.193	57,75	2.188	5.263	57	2.470	56,40	2.463	5.564
Simple_57	58	2.188	52,76	2.185	5.271	58	2.505	53,06	2.493	6.203
Simple_58	59	2.193	54,01	2.182	4.151	59	2.474	58,92	2.466	6.176
Simple_59	60	2.173	57,53	2.153	5.073	60	2.486	57,47	2.478	5.612
Simple_60	61	2.175	49,68	2.156	5.106	61	2.562	54,69	2.540	5.440
Simple_61	62	2.207	82,82	2.191	11.730	62	2.510	74,55	2.496	10.316
Simple_62	63	2.214	51,19	2.206	5.106	63	2.503	51,20	2.490	5.567

Tabelle A.1.: Rohdaten der Messungen mit Einzelwerten
(T ... Taktzyklen, μ ... Mittelwert, σ ... Standardabweichung)

			Marshalling via struct					Marshalling via IPC-Stream				
Versuch	Arrays	Elemente	IPC-Worte	μ [T]	σ [T]	Min [T]	Max [T]	IPC-Worte	μ [T]	σ [T]	Min [T]	Max [T]
static_array_fixed_length_1	1	10	11	2.002	54,97	1.987	4.910	12	2.105	54,91	2.091	5.295
static_array_fixed_length_2	2	10	21	2.049	51,71	2.046	5.239	23	2.162	52,27	2.150	5.171
static_array_fixed_length_3	3	10	31	2.100	53,21	2.082	5.242	34	2.232	51,13	2.221	3.923
static_array_fixed_length_4	4	10	41	2.132	50,00	2.120	5.032	45	2.304	52,67	2.292	4.154
static_array_fixed_length_5	5	10	51	2.185	57,16	2.176	5.218	56	2.391	56,75	2.386	5.304
static_array_fixed_length_6	6	10	61	2.233	53,32	2.215	5.115	—	—	—	—	—
static_array_variable_length_1	10	1	11	2.046	63,38	2.026	5.860	21	2.466	72,61	2.460	11.169
static_array_variable_length_2	10	2	21	2.079	54,71	2.070	5.304	31	2.496	60,12	2.484	5.402
static_array_variable_length_3	10	3	31	2.110	49,33	2.091	3.967	41	2.498	55,33	2.487	4.207
static_array_variable_length_4	10	4	41	2.124	53,33	2.114	4.991	51	2.519	60,41	2.499	5.588
static_array_variable_length_5	10	5	51	2.159	50,75	2.147	5.073	61	2.541	57,49	2.534	5.605
static_array_variable_length_6	10	6	61	2.449	57,73	2.434	5.602	—	—	—	—	—
fast_out_static_array_variable_length_1	10	1	11	1.985	44,63	1.981	4.893	21	2.546	58,86	2.531	6.360
fast_out_static_array_variable_length_2	10	2	21	2.002	50,65	1.993	4.919	31	2.602	58,81	2.596	5.709
fast_out_static_array_variable_length_3	10	3	31	2.035	50,10	2.020	4.964	41	2.578	63,34	2.564	5.659
fast_out_static_array_variable_length_4	10	4	41	2.049	50,79	2.037	4.973	51	2.607	58,03	2.599	5.638
fast_out_static_array_variable_length_5	10	5	51	2.054	47,12	2.049	3.894	61	2.650	70,19	2.635	5.738
fast_out_static_array_variable_length_6	10	6	61	2.172	53,84	2.159	5.106	—	—	—	—	—
dynamic_array_fixed_length_1	1	10	12	2.149	68,97	2.138	9.740	12	2.106	71,36	2.091	10.492
dynamic_array_fixed_length_2	2	10	22	2.224	56,76	2.209	5.144	23	2.199	55,08	2.194	5.242
dynamic_array_fixed_length_3	3	10	32	2.301	53,66	2.283	5.171	34	2.316	61,81	2.312	5.550
dynamic_array_fixed_length_4	4	10	42	2.371	53,27	2.360	4.263	45	2.360	57,15	2.351	5.605
dynamic_array_fixed_length_5	5	10	52	2.421	55,82	2.407	5.307	56	2.450	57,60	2.434	5.564
dynamic_array_fixed_length_6	6	10	62	2.528	60,38	2.516	5.688	—	—	—	—	—
dynamic_array_variable_length_1	10	1	12	2.758	59,82	2.750	5.942	21	2.830	62,44	2.818	5.948
dynamic_array_variable_length_2	10	1	22	2.779	58,28	2.768	5.984	31	2.830	64,10	2.821	6.014
dynamic_array_variable_length_3	10	3	32	2.758	60,65	2.747	5.951	41	2.854	56,87	2.839	6.052
dynamic_array_variable_length_4	10	4	42	2.764	58,11	2.747	5.857	51	2.866	61,41	2.862	6.120
dynamic_array_variable_length_5	10	5	52	2.772	58,68	2.756	5.975	61	2.871	63,28	2.865	5.742
dynamic_array_variable_length_6	10	6	62	2.809	58,63	2.803	5.901	—	—	—	—	—
fast_dynamic_array_variable_length_1	10	1	21	2.459	76,03	2.448	10.481	21	2.564	78,59	2.552	11.464
fast_dynamic_array_variable_length_2	10	2	31	2.470	58,99	2.463	5.561	31	2.767	58,31	2.750	4.488
fast_dynamic_array_variable_length_3	10	3	41	2.472	61,68	2.466	5.496	41	2.618	60,12	2.608	5.535
fast_dynamic_array_variable_length_4	10	4	51	2.533	57,42	2.525	5.612	51	2.598	57,37	2.593	4.367
fast_dynamic_array_variable_length_5	10	5	61	2.494	58,92	2.484	5.407	61	2.609	54,96	2.602	5.624

Tabelle A.2.: Rohdaten der Messungen mit Arrays
(T . . . Taktzyklen, μ . . . Mittelwert, σ . . . Standardabweichung)

B. Lizenz



Die LaTeX-Quelldateien der vorliegenden Arbeit sind vollständig in deren PDF-Version eingebettet. Sie können mit einem geeigneten Programm ausgelesen und weiterverwendet werden.