Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Assembler-Instrumentierung für die Leistungsanalyse paralleler Systeme

Dietmar Jansen

Jül-3653

Kurzfassung

Bei der Analyse des Laufzeitverhaltens paralleler Programme hat sich insbesondere bei Message-Passing-Systemen die Untersuchung von Ereignisspuren als eine geeignete Methode herausgestellt. Zur Erzeugung dieser Spuren müssen die Programme instrumentiert, d. h. zusätzliche Anweisungen in diese integriert werden. Im wesentlichen existieren zwei Ansätze zur Instrumentierung, die Quellcode- und die Objektcode-Instrumentierung.

In dieser Arbeit wird ein neuer Ansatz zur Instrumentierung von Programmen untersucht, bei dem die notwendigen Änderungen am Assemblercode der Programme vorgenommen werden. Zu diesem Zweck wurde das System SALTO durch eine Bibliothek namens ASINST erweitert, deren Funktionen eine systemunabhängige Instrumentierung von Assembler-Quellcodes erlauben.

Exemplarisch wurde diese Bibliothek genutzt, um ein Instrumentierungswerkzeug zu implementieren, mit dem Ereignisspuren für das Analysewerkzeug VAMPIR generiert werden können.

Zur benutzerfreundlichen Nutzung dieses Ansatzes wurde ein System zur Erzeugung von instrumentierten Programmen entwickelt, das dem Benutzer die notwendigen Arbeitsschritte weitgehend abnimmt. Mit dem Einsatz von Instrumentierungs-Kontrollskripten und einer graphischen Benutzeroberfläche lassen sich zusätzlich die zu instrumentierenden Programmelemente komfortabel auswählen.

Abstract

The use of event traces for the analysis of the runtime behavior of parallel programs on message passing systems proved to be an effective technique. In order to create these traces, the programs have to be instrumented, i. e. new instructions must be integrated. Basically, there are two approaches, the source code and the object code instrumentation.

In this thesis, a new approach for instrumenting programs is investigated. The necessary alternations are carried out in the assembly sources of the programs. For this purpose, the system SALTO was extended by a library called ASINST. Its functions allow to instrument assembly sources easily in a system independent manner.

This extension was used to implement an instrumentation tool in order to generate event traces for the analysis tool VAMPIR.

To improve ease of use, a system was developed that takes over all necessary steps to create instrumented programs. Furthermore, the user can conveniently select the program elements to be instrumented with the help of instrumentation control scripts and a graphic user interface.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Für wissenschaftliches Rechnen und komplexe industrielle Anwendungen ist die Nutzung von Großrechnern und Supercomputern unerläßlich. Auch wenn in den letzten Jahren die Rechenleistung von Personal-Computern und Workstations beachtlich gesteigert werden konnte, reicht diese für die Simulation komplexer Systeme, wie z. B. für die Klimaforschung, nicht aus. Dies gilt ebenfalls für Anwendungen aus anderen Bereichen, beispielsweise der Bildverarbeitung oder der Kryptologie. Obwohl die Leistungsfähigkeit von Großrechnern und Supercomputern ebenfalls erheblich verbessert werden konnte, stellt die verfügbare Rechenleistung immer noch einen Engpaß dar. Einerseits sind mit der Steigerung der Rechenleistung auch die Ansprüche der Anwender gestiegen, so daß häufig mehr Daten in eine Simulation einfließen und somit auch die Komplexität der Berechnungen zunimmt, oder aber der Anwender erwartet einfach kürzere Antwortzeiten. Andererseits ermöglicht diese hohe Rechenleistung neue Anwendungen, die sich aufgrund ihrer Komplexität bisher der rechnergestützten Betrachtung entzogen.

Die Leistungsfähigkeit einzelner, heute erhältlicher Prozessoren reicht bei weitem nicht aus, um die gewünschte Rechenleistung bereitzustellen. Da sich an dieser Situation in absehbarer Zeit nichts ändern wird, ist eine signifikante Leistungssteigerung nur durch Kopplung und gleichzeitiger Benutzung mehrerer Prozessoren erreichbar. Dies führt zur Nutzung von Parallelrechnern oder von durch Netzen verbundenen Rechnersystemen.

Das gleichzeitige Verwenden von mehreren Prozessoren setzt voraus, daß geeignete Hardware zur Verfügung steht und ein Programmiermodell gewählt wird, das nicht mehr die sequentielle, sondern die parallele Verarbeitung von Daten und Instruktionen vorsieht. Trotz beachtlicher Fortschritte auf diesen Gebieten ist die effiziente Implementierung eines parallelen Programms häufig eine komplexe Aufgabe und mit zahlreichen Problemen verbunden. Neben den Programmierfehlern, die aus der sequentiellen Programmierung bekannt sind, gibt es in der parallelen Programmierung neue Fehlerklassen.

Dies sind zum einen Fehler, die durch ein unkoordiniertes und somit fehlerhaftes

Verhalten der verschiedenen Befehlsströme der einzelnen Prozessoren gekennzeichnet sind und deshalb zu falschen Ergebnissen führen. Zum anderen sind es Designfehler, die dazu führen, daß das Programm zwar die korrekten Ergebnisse liefert, dabei aber wesentlich mehr Zeit als erwartet benötigt. Vor allem bei der Portierung von sequentiellen Programmen auf Parallelrechnern lassen sich solche Designfehler häufig beobachten. Die Folgen einer Designentscheidung sind auch für erfahrene Programmierer mit guten Kenntnissen der Maschinenarchitektur und des Programmiermodells aufgrund der Komplexität der zugrundeliegenden parallelen Hardware oft erst nach Ausführung des Programms erkennbar.

Diese Fehler lassen sich in der Regel auf das schlecht vorhersagbare Laufzeitverhalten paralleler Programme zurückführen, da im Rechner mehrere Befehlsströme und Datensätze, bis auf eventuell vorhandene Synchronisationspunkte, gleichzeitig und unabhängig voneinander verarbeitet werden. Dies hat zur Folge, daß die von der sequentiellen Programmierung bekannte Methode des sukzessiven Einkreisens von Fehlern meist nicht zum Ziel führt. Des weiteren ist das klassische Debugging, also das Anhalten und schrittweise Ausführen von Programmen nur dann verwendbar, wenn sich der Fehler nicht in der Kommunikation der Prozessoren untereinander befindet.

Oft werden Fehler erst nach der Ausführung eines Programms erkannt, so daß Korrekturen vorgenommen werden müssen, was in der Regel mit erheblichem Aufwand verbunden ist. Deshalb ist der Entwicklungsprozeß paralleler Programme gekennzeichnet vom ständigen Wechsel zwischen Beobachtung des lauffähigen Programms und Modifikation des zugrundeliegenden Quellcodes.

Eine Vielzahl von Analysewerkzeugen wurde in den letzten Jahren entwickelt, um den Prozeß der Software-Entwicklung paralleler Programme zu verbessern. Diese Werkzeuge analysieren zumeist Laufzeitdaten, die während der Ausführung eines Porgramms ermittelt werden. Insbesondere für Message-Passing-Systeme hat sich das *Tracing* von Programmen als hilfreiche Methode zur Generierung von Laufzeitdaten erwiesen. Dabei werden alle relevanten Ereignisse während der Ausführung des Programms in einer *Ereignisspur* aufgezeichnet, um diese nach der Beendigung des Programms untersuchen zu können.

Zur Erzeugung einer Ereignisspur wird in der Regel in das Programm eingegriffen, d. h. es werden zusätzliche Anweisungen in das Programm eingefügt, die dann während der Laufzeit des Programms die Ereignisspur generieren. Dieser Eingriff wird als Instrumentierung bezeichnet. Die Instrumentierung eines Programms kann natürlich von Hand am Quellcode des Programms vorgenommen werden, doch ist diese Methode weder benutzerfreundlich noch effektiv, da bei großen Programmen die Instrumentierung recht aufwendig werden kann. Sinnvollerweise sollte dieser Prozeß deshalb durch ein Instrumentierungswerkzeug übernommen werden.

Im wesentlichen gibt es zwei Ansatzpunkte zur Implementierung eines Instrumentierers; die Möglichkeit den Quellcode des Programms zu verändern oder die Änderungen am Objektcode des Programms vorzunehmen. Beide Ansätze haben ihre Vor- und Nachteile, auf die in Kapitel 2 eingegangen wird.

Zur Zeit gibt es eine ganze Reihe von zumeist graphischen Analysewerkzeugen, wie z. B. VAMPIR [2, 29] und Pablo [11, 28], aber auch andere Werkzeuge, beispielsweise EARL [16, 17], das die einfache Erstellung neuer Analysewerkzeuge durch das Schreiben von Skripten ermöglicht. Eine Gemeinsamkeit dieser Werkzeuge ist, daß sie zuvor erzeugte Ereignisspuren benötigen, um diese dann graphisch darzustellen oder weiterzuverarbeiten. Dabei verwenden die meisten Werkzeuge ihr eigenes Ereignisspurformat.

Im Rahmen dieser Diplomarbeit wurde ein Instrumentierungswerkzeug entwickelt, das weder den Quellcode noch den Objektcode, sondern den Assemblercode des entsprechenden Programms instrumentiert. Der Vorteil dieses neuen Ansatzes ist es, daß der Instrumentierer unabhängig von der Programmiersprache arbeitet, in der das zu instrumentierende Programm geschrieben ist. Zusätzlich ist das Instrumentierungswerkzeug, durch die Verwendung von SALTO [14, 15], leicht zu portieren. Exemplarisch wird dieser Ansatz zur Generierung von VAMPIR-Ereignisspuren genutzt. Ferner wurde ein System zur Automatisierung der Instrumentierung entwickelt, das sowohl mit Hilfe von Instrumentierungs-Kontrollskripten als auch durch eine graphische Benutzerschnittstelle gesteuert werden kann.

In Kapitel 2 wird zunächst ein Überblick über die bisherigen Ansätze zur Generierung von Ereignisspuren gegeben. Das Message-Passing-Programmiermodell wird beschrieben, und schließlich werden einige Analysewerkzeuge, insbesondere VAMPIR, kurz vorgestellt. Kapitel 3 beschreibt das System SALTO und in Kapitel 4 wird auf die Eigenschaften und die Programmierschnittstelle der VAMPIR-Tracebibliothek eingegangen, die für das Fortran77 Instrumentierungswerkzeug `vampir.inst` [26] entwickelt wurde. Diese Bibliothek kann allerdings auch ohne das Werkzeug `vampir.inst` genutzt werden. Anschließend wird in Kapitel 5 das Instrumentierungswerkzeug für Assemblercodes vorgestellt, dabei auf Besonderheiten und Probleme der Assembler-Instrumentierung eingegangen, das System zur automatischen Instrumentierung von Programmen beschrieben und am Ende diese Kapitels die Steuerung der Instrumentierung durch die Instrumentierungs-Kontrollskripte und die graphische Benutzeroberfläche erläutert. Im letzten Kapitel werden die Ergebnisse dieser Arbeit zusammengefaßt und ein Ausblick auf Einsatzmöglichkeiten und Erweiterungen gegeben.

Kapitel 2

Grundlagen

In diesem Kapitel werden sowohl einige notwendige Begriffe geklärt, als auch die derzeitige Situation auf dem Gebiet der Analyse von Message-Passing-Programmen beschrieben.

Zu Beginn werden die verschiedenen Möglichkeiten der Gewinnung von Laufzeitdaten erläutert. Da sich diese Arbeit mit der Erzeugung von Ereignisspuren befaßt, werden die gängigen Ansätze zur Implementierung eines Instrumentierungswerkzeuges behandelt und einige typische Vertreter vorgestellt.

Weiterhin wird speziell auf die Analyse von Message-Passing-Programmen eingegangen. Dazu wird das Message-Passing-Programmiermodell und seine derzeitig bedeutenste Implementierung MPI [27] behandelt. Des weiteren wird auf die Besonderheiten bei der Erzeugung von Ereignisspuren für Message-Passing-Programme eingegangen. Zum Abschluß dieses Kapitels werden einige Analysewerkzeuge kurz vorgestellt, insbesondere VAMPIR[2, 29], da der im Rahmen dieser Arbeit verfolgte Ansatz exemplarisch zur Generierung von VAMPIR-Ereignisspuren genutzt wird.

2.1 Ansätze zur Generierung von Laufzeitdaten

Wie schon in der Einleitung beschrieben, ist das Beobachten von parallelen Programmen während ihrer Laufzeit von besonderem Interesse. Es müssen daher Methoden verwendet werden, die zwar einen möglichst detaillierten Einblick in die dynamischen Vorgänge zur Laufzeit eines Programms gewähren, aber den Ablauf des Programms möglichst gering beinflussen. Dies läßt sich erreichen, indem während der Laufzeit des Programms vorher festgelegte Ereignisse protokolliert, also in eine Datei gespeichert und diese Daten erst nach Beendigung des Programms mit Hilfe geeigneter Werkzeuge ausgewertet werden.

Im wesentlichen gibt es drei Techniken, um die gewünschten Daten während der Laufzeit eines Programms zu ermitteln:

1. Sampling: In regelmäßigen Abständen wird das parallele Programm unterbrochen und die gegenwärtige Programmregion durch das Abfragen des Programmzählers und mit Hilfe der Symboltabelle des Programms bestimmt. So ist es möglich, z. B. die Aufenthaltsdauer und die Aufenthaltshäufigkeit in den einzelnen Programmregionen zu schätzen. Sampling wird vor allem zur Generierung von *Profiles* genutzt und kann auf verschiedenen Stufen eingesetzt werden. Möglich ist die Beobachtung des gesamten Programms, aber auch einzelner Funktionen, einzelner Basisblöcke oder gar einzelner Instruktionen. Ein wesentlicher Vorteil des Sampling ist, daß die Gewinnung der Daten über das Programm seine Ausführungszeit nur geringfügig verändert, allerdings ist auch die Anzahl der gesammelten Daten gering. Auch wenn das Sampling keine genaue Methode ist, kann sie bei der Suche nach Programmteilen, die einen Großteil der Zeit oder anderer Resourcen verbrauchen und somit eventuell verbessert werden sollten, hilfreich sein.

2. Counting: Unter Counting versteht man das Zählen bestimmter Ereignisse, wie beispielsweise das Auftreten von Seitenfehlern, die Abarbeitung bestimmter Instruktionen, wie Fließkomma-, Integer-, Speicher- und Lade-Operationen oder aber auch das Versenden und das Empfangen von Nachrichten. Die verwendeten Zähler können als Software oder als Hardware implementiert sein. Die Verwendung hardwareseitiger Zähler hat den Vorteil, daß das Programm im Prinzip nicht beeinflußt wird. Insgesamt ist die erhaltene Datenmenge gering, womit auch die Aussagekraft der Daten beschränkt ist.

3. Tracing: Zur Laufzeit wird beim Tracing zu jedem relevanten Ereignis, z. B. der Aufruf bzw. das Verlassen einer Funktion oder das Senden bzw. das Empfangen einer Nachricht, ein Ereignis-Record angelegt. Diese Datensätze enthalten in der Regel einen Zeitstempel, eine Kennung die dem Ereignistyp zugeordnet ist, sowie eine vom Typ abhängige Liste weiterer Parameter. All diese Ereignis-Records werden in eine Protokolldatei gespeichert, die *Ereignisspur* bzw. *Event Trace* genannt wird. Es wird in der Regel eine Protokolldatei für jeden beteiligten Prozessor bzw. Prozeß erstellt. Diese müssen nach Beendigung des Programms zu einem chronologischen Ablaufprotokoll zusammengefaßt werden, um schließlich das parallele Programm analysieren zu können (Abb. 2.1). Obwohl in der Regel die Ereignis-Records im lokalen Speicher des jeweiligen Prozessors geschrieben und erst nach Überlauf auf externe Speicher ausgelagert werden, ist die Beeinflussung des zu analysierenden Programms recht groß. Ein wesentlicher Nachteil des Tracing sind die erzeugten Datenmengen. Unter der Annahme, daß ein Programm auf 100 Prozessoren ausgeführt wird, ein Ereignis-Record 20 Byte lang ist und während der Laufzeit des Programms alle 10 Millisekunden auf jedem Prozessor ein Ereignis aufgezeichnet wird, wäre die erzeugte Ereignisspur, bei einer Ausführungszeit von 5 Minuten, bereits ca. 57 MB groß. Dies stellt vor allem ein Problem dar, wenn das Programm das erste Mal analysiert und meist das komplette Programm instrumentiert wird, da noch nicht bekannt ist, wo sich kritische Stellen im Programm befinden könnten. Deshalb werden flexible und mächtige Instru-

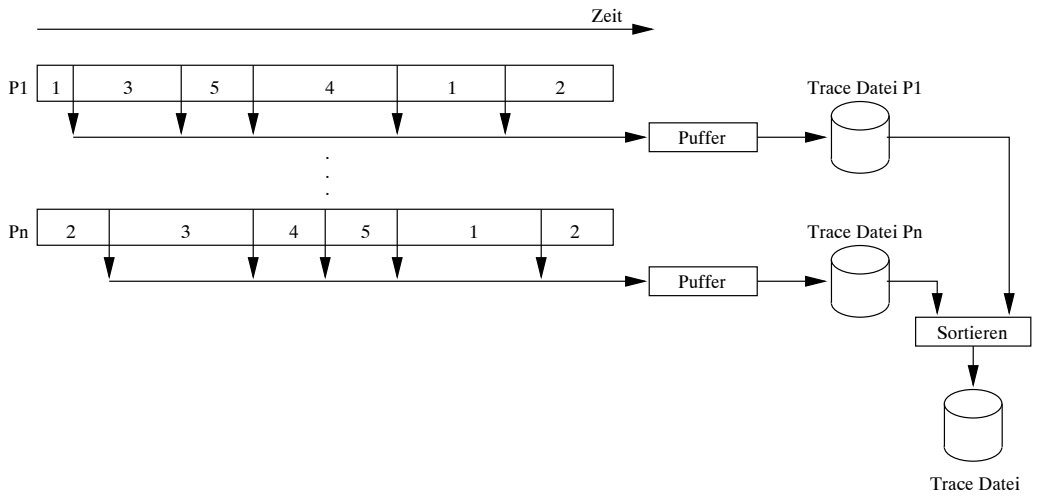


Abbildung 2.1: Tracing von parallelen Programmen

mentierungswerkzeuge benötigt, damit die Eingriffe ins Programm durch den Benutzer möglichst einfach und effizient vorgenommen werden können, um nur die Stellen des Programms zu instrumentieren, die für den Benutzer von Bedeutung sind und untersucht werden sollen.

2.2 Ansätze der Instrumentierung

Die Erzeugung von Ereignisspuren, aber häufig auch die Generierug von Profiles setzt voraus, daß das zu untersuchende Programm instrumentiert wird, um die notwendigen Daten aufzeichnen zu können. In der Regel werden dabei neue Befehle mit ins Programm aufgenommen. Meist handelt es sich dabei um Aufrufe von Bibliotheksfunktionen, die dann die eigentliche Arbeit leisten. Die einfachste Form der Instrumentierung ist, den Quellcode des Programms per Hand zu instrumentieren, also direkt in der jeweiligen Programmiersprache die Funktionsaufrufe in den Quellcode hineinzuschreiben. Diese Methode ist sicherlich nicht benutzerfreundlich und bei größeren Programmen mit erheblichem Arbeitsaufwand verbunden. Deshalb ist man bestrebt, diese Arbeit weitgehend zu automatisieren. Dabei soll der Benutzer lediglich auf eine möglichst einfache Weise bestimmen, welche Programmteile instrumentiert werden sollen.

2.2.1 Quellcode-Instrumentierung

Ein weit verbreiteter Ansatz zur automatischen Instrumentierung ist die Transformation des Quellcodes eines Programms. Dabei werden an geeigneter Stelle Befehle in den Quellcode des Programms eingefügt. Um diese Stellen zu finden, muß der Quellcode einer Syntaxanalyse unterzogen werden. Die einzufügenden Befehle müssen der Syntax der im Quellcode genutzten Programmiersprache entsprechen.

Diese Instrumentierer können als separates Werkzeug implementiert oder als Vorverarbeitungsschritt in einem Compiler integriert werden (Abb. 2.2).

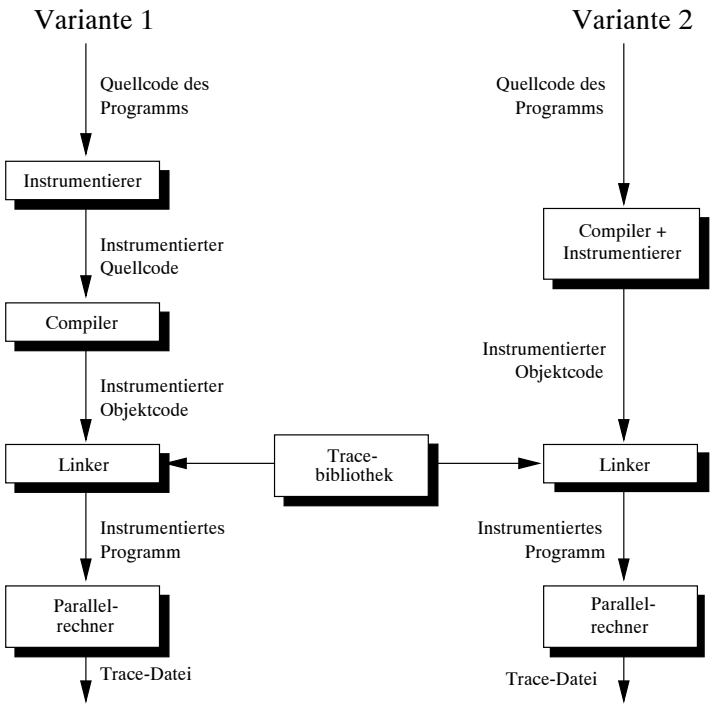


Abbildung 2.2: Instrumentierung des Quellcodes

Typische Vertreter der ersten Variante sind z. B. `vampir.inst` [26] zur Instrumentierung von Fortran77-Quellcode und SvPablo [23].

Das Werkzeug `vampir.inst` wird über die Kommandozeile mit einer Liste von Fortran77-Quellcode-Dateien und einem Zielverzeichnis gestartet, in dem die modifizierten Dateien abgelegt werden. Durch eine Symboldatei kann die Auswahl der zu protokollierenden Ereignisse beeinflußt werden. Die modifizierten Fortran77-Quellcode-Dateien können nun mit einem Fortran77-Compiler übersetzt werden.

SvPablo ist ein Werkzeug mit graphischer Benutzeroberfläche zur Instrumentierung von ANSI C-Quellcodes und zur strukturierten Darstellung von Laufzeitleistungsdaten (Abb. 2.3). C-Programme können interaktiv instrumentiert werden, indem SvPablo jeden C-Quellcode des Programms analysiert und dabei relevante Ereignisse identifiziert. Der Benutzer kann diese Ereignisse (Funktionsaufrufe und äußere Schleifen) selektieren, die dann von SvPablo entsprechend instrumentiert werden, indem eine neue Version des Quellcodes mit den Aufrufen der entsprechenden Funktionen aus der Tracebibliothek generiert wird. Die modifizierten Quellcodes müssen anschließend mit einem C-Compiler übersetzt werden.

Um ein lauffähiges Programm zu erhalten, müssen jeweils noch die entsprechenden Tracebibliotheken hinzugebunden werden. Wird dieses Programm nun gestartet, so erhält man pro Prozessor bzw. Prozeß eine Datei mit den jeweils aufgezeichneten Ereignis-Records. Diese Dateien müssen noch zu einer Datei zusammengefaßt und

dabei die Ereignisse in eine chronologische Reihenfolge gebracht werden. Dies übernimmt bei `vampir.inst` das Programm vptmerge und bei SvPablo das Programm SvPabloCombine. Nach diesen Schritten steht die Ereignisspur im jeweiligen Format zur Verfügung.

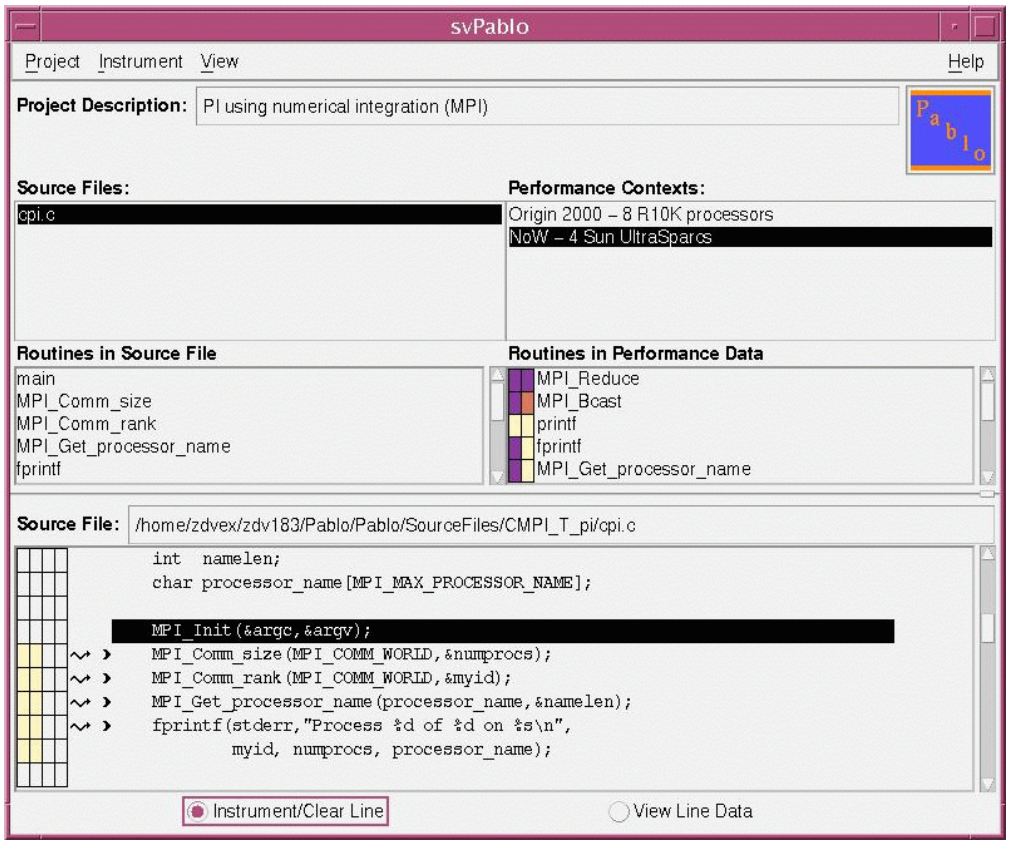


Abbildung 2.3: Instrumentierung mit SvPablo

Instrumentierer der zweiten Variante werden mit in den Compiler integriert, wie dies z. B. in den Compilern der Firma Cray Research Inc. für das Analysewerkzeug Apprentice [10] oder im HPF[1]-Compiler der Firma Portland Group [1] für VAMPIR-Ereignisspuren der Fall ist. Um einen Quellcode zu instrumentieren, muß lediglich eine Option des entsprechenden Compilers genutzt werden. Auch bei dieser Variante ist es notwendig, die jeweilige Tracebibliothek hinzuzubinden, um ein lauffähiges Programm zu erhalten.

Da diese Variante von Instrumentierern direkt im Compiler integriert ist, ist es möglich, die Instrumentierung nach den Optimierungsschritten des Compilers auszuführen, so daß die Auswirkung auf die Ausführungszeit des Programms gering gehalten werden kann. Des weiteren ist eine erneute Syntaxanlayse nicht notwendig, da diese elementarer Bestandteil eines Compilers ist. Dies sind wesentliche Vorteile dieser Variante, doch werden zumeist nur herstellereigene Analysewerkzeuge bzw.

[1] High Performance Fortran

Ereignisspurformate unterstützt, so daß diese Variante für eine große Anzahl von Analysewerkzeugen nicht genutzt werden kann.

Ein großer Vorteil der Quellcode-Instrumentierung ist, daß sich die gesammelten Informationen leicht wieder dem zugrundeliegenden Quellcode zuordnen lassen und somit die eventuell zu verbessernden Stellen leicht zu finden sind. Zusätzlich ist es möglich, Quellcode-Instrumentierungswerkzeuge so zu implementieren, daß sie leicht zu portieren und somit auf verschiedenen Computersystemen eingesetzt werden können. Insgesamt können aber auch drei große Nachteile dieser Art der Instrumentierung angeführt werden: Programmteile, die nur als Objektcode zur Verfügung stehen, wie z. B. spezielle Bibliotheken, können nicht instrumentiert werden, und die Instrumentierung von Programmen, deren Quellcodes in mehreren Programmiersprachen verfaßt sind, ist nur schwer durchführbar. Zweitens ist die Implementierung für komplexe moderne Programmiersprachen, wie C++ oder Fortran90, schwierig, falls das Werkzeug nicht in einen Compiler integriert wird, da die Syntaxanalyse solcher Sprachen nicht trivial ist. Schließlich muß nach jeder Änderung der Instrumentierung das Programm neu übersetzt werden, was bei großen Programmen zum Teil mit erheblichem Zeitaufwand verbunden ist.

2.2.2 Objektcode-Instrumentierung

Der zweite Ansatz zur automatisierten Instrumentierung von Programmen ist, die notwendigen Änderungen im Objektcode bzw. Binärcode des Programms vorzunehmen.

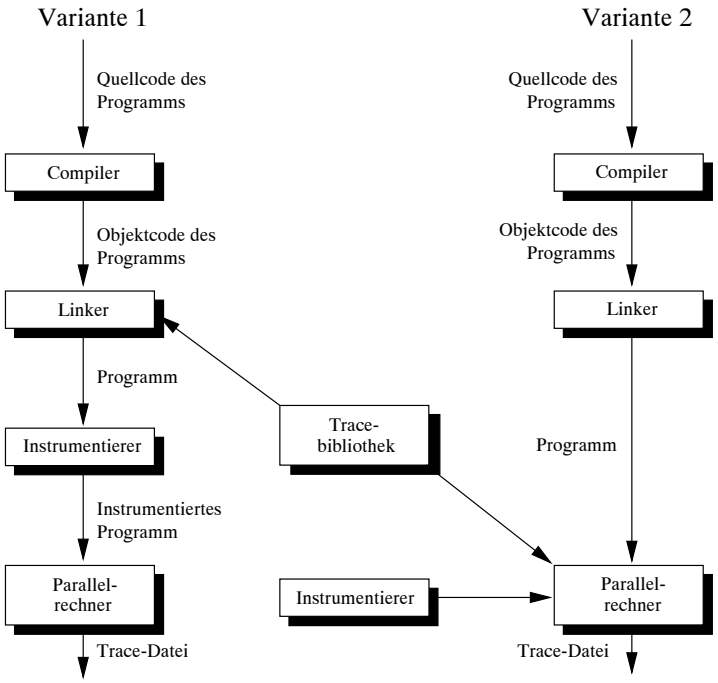


Abbildung 2.4: Instrumentierung des Objektcodes

Wie bei der Quellcode-Instrumentierung werden in der Regel Aufrufe von Bibliotheksroutinen in das zu untersuchende Programm eingefügt, so daß die entsprechende Tracebibliothek hinzugebunden werden muß. Die Änderungen werden bei der Objektcode-Instrumentierung häufig nicht an den Objektcode-Dateien vorgenommen, sondern vielmehr an dem ausführbaren Programm. Dies hat den Vorteil, daß das Programm nur einmal neu gebunden werden muß. Eine weitere Variante ist, den Binärcode während der Ausführung des Programms zu instrumentieren (Abb. 2.4). Bei dieser Variante werden die Programmdateien nicht verändert bzw. neue erzeugt, sondern nur der Binärcode des Programms verändert, der sich zur Laufzeit des Programms im Speicher des Rechners befindet.

Ein Vertreter der ersten Variante ist die Objektcode-Instrumentierungsfunktion von PAT (Performance Analysis Tool) [20]. PAT ist ein Analysewerkzeug der Firma Cray Research Inc. für das massiv parallele System CRAY T3E. Es bietet dem Benutzer einige Möglichkeiten, Leistungsdaten zu ermitteln und diese Informationen anzuzeigen. Insbesondere bietet PAT die Möglichkeit Funktionsaufrufe in ein ausführbares Programm zu integrieren. PAT fügt vor und nach dem Aufruf einer Benutzer- oder Bibliotheksfunktion Aufrufe zu Eintritts- (entry) und Austritts- (exit) funktionen ein. Diese Eintritts- und Austrittsfunktionen können sowohl vom Benutzer als Objekt- oder Bibliotheksdatei als auch durch die generischen Routinen der zu PAT gehörenden Bibliothek bereitgestellt werden.

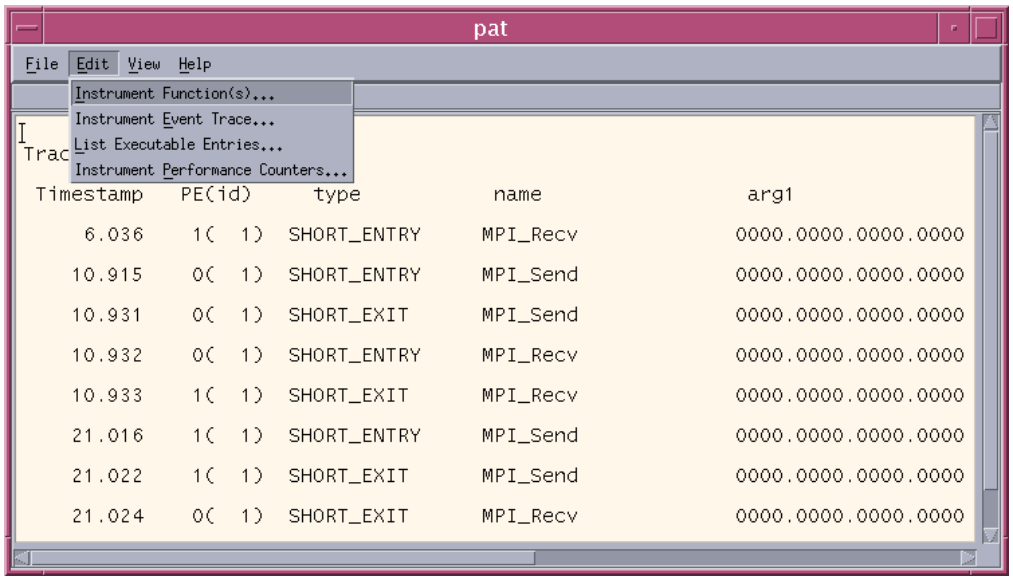


Abbildung 2.5: Instrumentierung mit PAT

Zur Instrumentierung eines Porgramms muß dessen Objektcode zusammen mit der Bibliothek von PAT *libpat.a*, falls vorhanden mit den vom Benutzer definierten Eintritts- und Austrittsfunktionen, und mit einer speziellen Datei *pat.cld*, die Direktiven für den Linker enthält, gebunden werden. Mit dem ausführbaren Programm kann nun PAT gestartet werden. Die eigentliche Instrumentierung wird also am ausführbaren Programm und nicht an den Objektcode-Dateien vorgenommen.

PAT verfügt sowohl über eine graphische Benutzeroberfläche als auch über einen Kommandozeilen-Modus. In beiden Modi können nun Aufrufe von Benutzer- oder Bibliotheksfunktion instrumentiert werden (Abb. 2.5). Eine weitere Möglichkeit ist, PAT eine Datei mit den zu instrumentierenden Funktionsnamen zu übergeben. PAT generiert ein neues lauffähiges Programm unter dem Namen *a.out*, das nun gestartet werden muß, um eine *PIF-Datei*[2] zu erzeugen, in der die gesammelten Daten gespeichert werden. Wird PAT danach erneut aufgerufen, stehen die Daten zur Verfügung, die durch die Instrumentierung erzeugt wurden.

Das Werkzeug ATOM [13] der Firma Digital Equipment Corporation für DEC Alpha Workstations bietet ebenfalls die Möglichkeit den Objektcode eines Programms zu manipulieren. Anders als bei PAT werden die zu instrumentierenden Programmelemente mit Hilfe einer Instrumentierungsroutine bestimmt, die in der Programmiersprache C zu implementieren ist. In dieser Routine müssen zunächst die Funktionen angemeldet werden, zu denen Aufrufe in das zu instrumentierende Programm eingefügt werden sollen. Mit den Funktionen der ATOM-Bibliothek können nun die Prozeduren, Basisblöcke und Instruktionen des Programms aufgefunden und an geeigneten Stellen Funktionsaufrufe integriert werden. Ferner existiert eine Funktion, die es ermöglicht bereits vorhandene Funktionsaufrufe zu ersetzen. ATOM nutzt diese Instrumentierungsroutine, um eine instrumentierte Version des zu analysierenden Programms zu erzeugen. Zur Generierung eines ausführbaren Programms werden noch die Funktionen hinzugebunden, zu denen Funktionsaufrufe in das Programm aufgenommen wurden.

Ein anderer Ansatz wird bei der Instrumentierung für das Leistungsanalyse-Werkzeug Paradyn [4] verfolgt. Die Instrumentierung wird erst nach dem Starten, also zur Laufzeit des zu analysierenden Programms, vorgenommen. Diese Methode wird dynamische Instrumentierung [21] genannt und hat unter anderem den Vorteil, daß das Programm nicht neu gebunden werden muß. Zur Nutzung muß nur ein Präprozessor gestartet werden, der die Tracebibliothek von Paradyn an das zu untersuchende Programm hängt. Somit kann die Analyse zu einem beliebigen Zeitpunkt während der Laufzeit des Programms beginnen. Des weiteren kann die Instrumentierung während der Laufzeit des Programms beliebig geändert werden. Dies hat zur Folge, daß die Eingriffe im Programm gering gehalten werden können, da nur die Teile des Programms zu instrumentieren sind, die zu diesem Zeitpunkt analysiert werden sollen. Die ermittelten Informationen werden nicht in einer Datei gesammelt, um sie nach Ausführung des Programms zu analysieren, sondern dem Benutzer direkt über eine graphische Oberfläche zur Verfügung gestellt.

Die Vorteile der Objektcode-Instrumentierung sind im wesentlichen die Möglichkeit, auch Programmteile zu instrumentieren, für die kein Quellcode zur Verfügung steht und daß die zu untersuchenden Programme nicht neu übersetzt werden müssen. Zusätzlich können auch Programme leicht instrumentiert werden, deren Quellcodes in verschiedenen Programmiersprachen verfaßt sind. Ein großer Nachteil ist jedoch, daß sich die gesammelten Daten häufig nicht oder nur schwer den entsprechenden

[2] pat information file

Quellcodes zuordnen lassen. Des weiteren ist die Instrumentierung von Objektcodes bzw. Binärcodes besonders systemspezifisch und deshalb schwer portierbar, so daß in der Regel Teile des Instrumentierungswerkzeuges für jedes zu unterstützende System neu implementiert werden müssen.

2.3 Analyse von Message-Passing-Programmen

Eines der am weitesten verbreiteten, parallelen Programmiermodelle ist das Message-Passing, da es zum einen sehr gut auf massiv parallelen Systemen als auch Workstation-Clustern einsetzbar und zum anderen zur Lösung einer Vielzahl von Problemen geeignet ist.

Die Analyse von Message-Passing-Programmen unterliegt einigen Besonderheiten, die auf das verwendete Programmiermodell zurückzuführen sind. Insbesondere ist die Kommunikation unter den einzelnen parallelen Prozessen von besonderem Interesse, die sich anhand von Ereignisspuren besonders gut beobachten läßt. Deshalb sind neben der verbrachten Zeit häufig auch die Art der Kommunikation, der Empfänger, der Absender und die Länge der Nachricht von Interesse, um das dynamische Verhalten eines Message-Passing-Programms zu verstehen.

2.3.1 Das Message-Passing-Programmiermodell

Das Message-Passing-Programmiermodell basiert auf der Annahme, daß eine Menge von Prozessen existiert, die jeweils über einen lokalen Speicher verfügen und durch das Versenden und Empfangen von Nachrichten miteinander kommunizieren. Ein wesentliches Merkmal dieses Programmiermodells ist das *zweiseitige* Kommunikationsschema. Dabei stehen auf beiden Seiten einer Kommunikation sowohl Operationen zum Versenden als auch Operationen zum Empfangen von Nachrichten zur Verfügung, um Daten von dem jeweiligen lokalen Speicher in den Speicher der Kommunikationspartner zu transferieren.

Unterschieden wird zum einen zwischen *synchroner* und *asynchroner* Kommunikation und zum andern zwischen vier Arten der Kommunikation, die sich aus der Anzahl und Verteilung der an dem Datenaustausch beteiligten Prozesse ergeben.

Bei der synchronen Kommunikation müssen die beteiligten Prozesse gleichzeitig am Nachrichtenaustausch teilnehmen. Wenn Prozesse, die an dem Nachrichtenaustausch beteiligt sind, noch nicht zur Kommunikation bereit sind, werden alle anderen Prozesse blockiert, bis alle Prozesse bereit sind. Ein Nachteil dieser Art des Datenaustauschs ist, daß Wartezeiten entstehen können, oder es gar zu Verklemmungen kommen kann, wenn z. B. ein Prozeß **A** auf eine Nachricht eines anderen Prozesses **B** wartet und der Prozeß **B** auf eine Nachricht des Prozesses **A**. Allerdings wird bei diesem Verfahren kein Puffer benötigt, um die Nachrichten zwischenzuspeichern.

Die asynchrone Kommunikation hat den Vorteil, daß Prozesse Nachrichten versenden können, ohne auf die Bereitschaft der Kommunikationspartner, Nachrichten zu empfangen, warten zu müssen. Außerdem können umgekehrt Prozesse jederzeit ihre Bereitschaft zum Empfang von Nachrichten signalisieren, ohne blockiert zu werden. Mit Hilfe von speziellen Operationen kann geprüft werden, ob die Kommunikation stattgefunden hat.

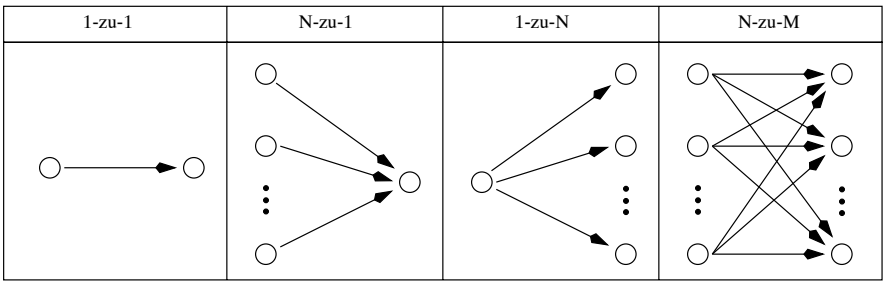


Abbildung 2.6: Arten der Kommunikation

Die vier verschiedenen Arten des Nachrichtenaustauschs (Abb. 2.6) lassen sich generell in Punkt-zu-Punkt und kollektive Kommunikationen unterscheiden. Bei der Punkt-zu-Punkt oder auch 1-zu-1 Kommunikation kann ein Prozeß eine Nachricht zu genau einem anderen Prozeß senden. Die kollektive Kommunikation ermöglicht einer Gruppe von Prozessen kollektive Datenaustauschsoperationen zu benutzen, um Operationen durchzuführen, wie beispielsweise das Verteilen von Daten von einem auf mehrere Prozesse. Es lassen sich drei Arten der kollektiven Kommunikation unterscheiden:

- Die 1-zu-N Kommunikation, bei der ein Prozeß die Nachricht an N Prozesse versendet.
- Die N-zu-1 Kommunikation, bei der ein Prozeß die Nachrichten von N Prozessen aufsammelt.
- Die N-zu-M Kommunikation, bei der N Prozesse Nachrichten an M Prozesse versenden.

Kollektive Kommunikationsereignisse werden jedoch von allen gängigen Instrumentierungssystemen auf Punkt-zu-Punkt Komunikationsereignisse abgebildet.

Eine weitere Besonderheit des Message-Passing-Programmiermodells ist, daß es nicht-deteministisch ist. Die Empfangsreihenfolge von Nachrichten, die z. B. von zwei Prozessen **A** und **B** an einen dritten Prozeß **C** gesendet werden, ist nicht definiert. Allerdings bieten die meisten Implementierungen dieses Programmiermodells eine Möglichkeit, den Determinismus zu garantieren.

Zwei Implementierungen dieses Modells sind PVM[3] [3] und MPI[4] [27]. Diese Varianten des Message-Passing-Programmiermodells sind als Bibliotheken für eine Vielzahl von Systemen verfügbar, wobei MPI gegenwärtig die am weitesten verbreitete Message-Passing-Bibliothek ist. Seit 1997 existiert die Definition der Version 2.0, aber zur Zeit kommt in der Regel nur die Version 1.2 zum Einsatz. Aufgrund der erlangten Bedeutung dieser Bibliothek wird sie in der Version 1.2 im folgenden kurz vorgestellt.

MPI

MPI ist ein komplexes System, das mehr als 120 Funktionen umfaßt. Allerdings reichen schon einige Basis-Funktionen aus, um komplexe parallele Programme zu erstellen [18]. Diese Basis-Funktionen sind :

- `MPI_INIT(int *arc, char **argv)`
  Initialisierung der Kommunikation zwischen Prozessen, argc und argv werden nur für die C-Variante benötigt.
- `MPI_FINALIZE()`
  Beendigung der Kommunikation.
- `MPI_COMM_SIZE(comm, size)`
  Bestimmt die Anzahl der Prozesse und speichert diese Information in `size` ab.
- `MPI_COMM_RANK(comm, pid)`
  Bestimmt den Bezeichner bzw. den *Rang* des Prozesses.
- `MPI_SEND(buf, count, datatype, dest, tag, comm)`
  Versenden einer Nachricht mit `count` Werten des Typs `datatype`, die sich ab der Speicheradresse `buf` befinden, an den Prozeß `dest` mit der *Kennung* `tag`.
- `MPI_RECV(buf, count, datatype, source, tag, comm, status)`
  Empfangen einer Nachricht mit `count` Werten des Typs `datatype` in den Speicher ab Adresse `buf` vom Prozeß `source` mit der *Kennung* `tag`. In `status` befinden sich nach Ausführung der Funktion Informationen über die empfangene Nachricht.

An den Funktionsdefinitionen ist abzulesen, daß Nachrichten durch eine *Kennung* gekennzeichnet werden. Diese wird als Argument an Sende- und Empfangsfunktionen übergeben, um den Operationen die gewünschte Nachricht zuzuordnen.

Prozesse werden bei MPI in Gruppen eingeteilt. Zu Beginn eines Programms wird bei vielen MPI Implementierungen eine feste Anzahl von Prozessen mit genau einem

[3] parallel virtual machine
[4] message passing interface

Prozeß pro Prozessor erzeugt. Diese Prozesse befinden sich alle in einer Gruppe und sind von 0 bis $n-1$ durchnumeriert, diese Nummer wird auch *Rang* des Prozesses genannt. Teilgruppen können während der Laufzeit des Programms erstellt werden.

Alle oben aufgeführten Funktionen, bis auf die ersten beiden, haben als Argument einen *Kommunikator* `comm`. Ein Kommunikator spezifiziert die Prozeßgruppe und den Kontext, also den Gültigkeitsbereich für Nachrichtenkennungen, in der die Operation ausgeführt werden soll. Dies ermöglicht die Entwicklung von modularen Programmen, da mit diesem Mechanismus Nachrichten innerhalb von Modulen oder Bibliotheken versendet werden können, ohne die Möglichkeit den Nachrichtenaustausch außerhalb der Module bzw. Bibliotheken zu beinflussen.

Während die Funktion `MPI_SEND` je nach Implementierung blockierend aber auch nicht blockierend sein kann, ist die Funktion `MPI_RECV` stets blockierend. In allen MPI Implementierungen stehen aber auch Sende- und Empfangsoperationen in synchroner und asynchroner Form zur Verfügung.

Ferner beinhaltet das MPI-System neben Funktionen zur Punkt-zu-Punkt Kommunikation auch Funktionen zur kollektiven Kommunikation, wie beispielsweise `MPI_GATHER`, die eine N-zu-1 Kommunikationsoperation realisiert, wobei alle Prozesse innerhalb des Kommunikators ihre Daten an den einen Prozeß senden oder das Gegenstück, die Funktion `MPI_SCATTER`, eine 1-zu-N Kommunikationsoperation, wobei der eine Prozeß seine Daten auf alle anderen Prozesse innerhalb des Kommunikators verteilt.

2.3.2 Erzeugung von Ereignisspuren

Da bei der Analyse von Message-Passing-Programmen nicht nur die verbrauchte Zeit, sondern insbesondere die Kommunikation der Prozesse untereinander von Interesse ist, reicht es nicht aus, generische Trace-Funktionen zu nutzen, die die benötigte Zeit und eventuell den Namen einer instrumentierten Funktion protokollieren.

Beim Aufruf von Message-Passing-Operationen möchte der Anwender in der Regel zusätzliche Informationen über Ziel, Länge, Art der Kommunikation, Zeitpunkt des Sendens und des Empfangs erhalten. Um diese Informationen liefern zu können, werden sogenannte *Wrapper-Funktionen* verwendet.

Während der Instrumentierung werden bestimmte Funktionsaufrufe, insbesondere Aufrufe von Kommunikationsfunktionen, durch Aufrufe der entsprechenden Wrapper-Funktion ersetzt. Deshalb müssen Wrapper-Funktionen dieselbe Parameterliste besitzen wie die durch sie zu ersetzenden Funktionen. Für jede Funktion, die ersetzt werden soll, muß eine Wrapper-Funktion implementiert werden. Die Wrapper-Funktion protokolliert alle relevanten Informationen und ruft die ersetzte Funktion auf (Abb. 2.7).

So ist es möglich, Ereignis-Records, die das Versenden oder das Empfangen einer Nachricht anzeigen, mit speziell auf die verwendete Kommunikationsfunktion

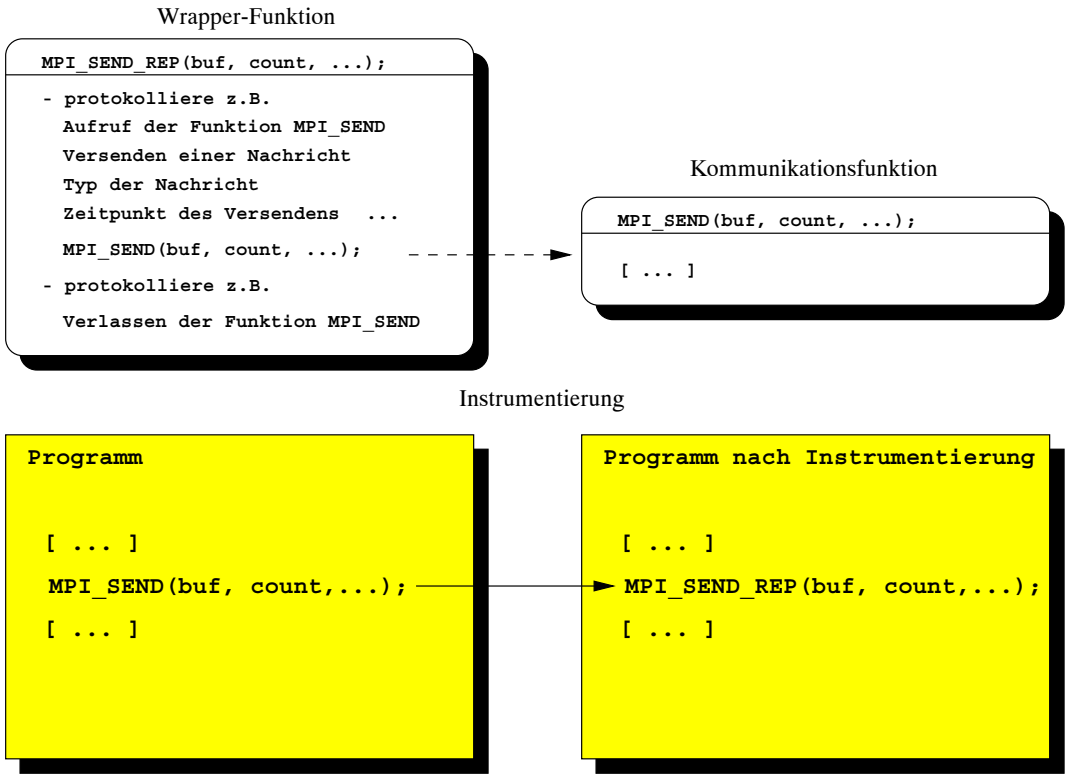


Abbildung 2.7: Verwendung von Wrapper-Funktionen

zugeschnittenen Informationen zu generieren. Trotz der Nutzung von Wrapper-Funktionen kann zumeist nicht der genaue Zeitpunkt des Versendens bzw. des Empfangs einer Nachricht bestimmt und muß deshalb eventuell geschätzt werden. Dies gilt vor allem bei asynchroner Kommunikation.

Für Message-Passing-Bibliotheken wird häufig eine eigene Bibliothek angelegt, die dann für alle Funktionen entsprechende Wrapper-Funktionen bereitstellt. Da es zur Zeit keinen Standard für Ereignisspuren gibt, verwenden die meisten Analysewerkzeuge ihr spezielles Format. Dies hat zur Folge, daß die Wrapper-Funktionen nicht nur für jede in der Bibliothek vorkommende Funktion, sondern auch für alle möglichen Formate implementiert werden müssen.

2.3.3 Analyse und Visualisierung der Ereignisspuren

Ereignisspuren dienen als Grundlage zur Untersuchung von parallelen Programmen. Da Ereignisspuren in den meisten Fällen sehr groß werden, müssen die in ihnen enthaltenen Daten in einer vernünftigen Form aufbereitet werden, um sie analysieren zu können. Häufig wird bei Analysewerkzeugen auf die Visualisierung der Ereignisspuren besonderen Wert gelegt, damit dem Benutzer die Daten in graphischer Form zur Verfügung gestellt werden können. Zu diesen Werkzeugen gehören z. B. VAMPIR [2, 29] und Pablo [11, 28].

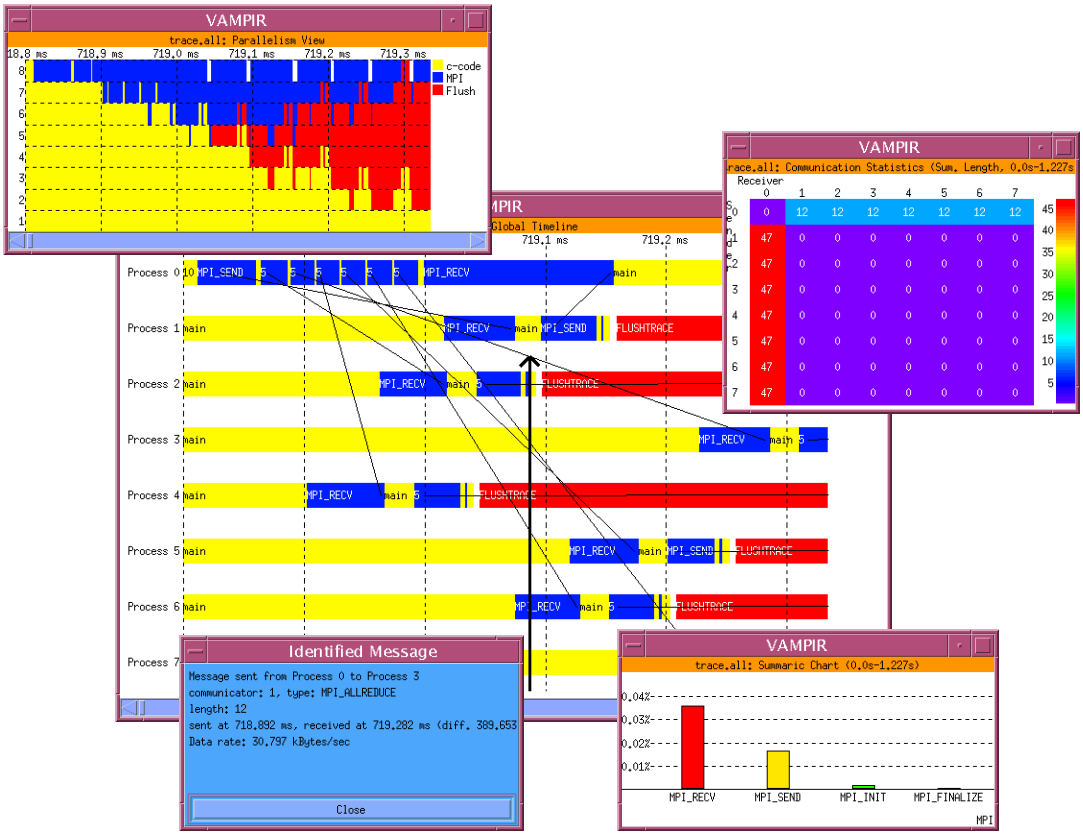


Abbildung 2.8: Analyse mit VAMPIR

VAMPIR (Visualization and Analysis of MPI Resources) liefert eine große Anzahl von graphischen Ansichten zur Analyse von Ereignisspuren (Abb. 2.8). Das dynamische Verhalten kann durch ein Zeitliniendiagramm kenntlich gemacht werden, wobei zu Beginn die gesamte Ereignisspur visualisiert wird. Der Benutzer kann jedoch die Ansicht verfeinern, indem er beliebige Ausschnitte der Ereignisspur selektiert, die dann vergrößert dargestellt werden. Zusätzlich verfügt VAMPIR über Ansichten zum parallelen Verhalten von Programmen und über statistische Ansichten, die beispielsweise Auskunft über die Häufigkeit von bestimmten Funktionsaufrufen oder über die Aufenthaltsdauer in bestimmten Funktionen geben. Des weiteren ist es möglich, Nachrichten zu identifizieren und Informationen über sie abzufragen.

Pablo hingegen stellt Module bereit, die durch einen Analysegraphen verbunden werden können, um Ereignisspuren zu transformieren und zu visualisieren, wobei die Module selbst individuell konfigurierbar sind. Es existieren Module, die z. B. Funktionsaufrufe zählen, die verbrauchte Zeit in einer Funktion akkumulieren und diese berechneten Daten beispielsweise in Balkendiagrammen visualisieren. Während die Ereignisspur gelesen wird, werden die gespeicherten Informationen anhand des Analysegraphen animiert und visualisiert.

Im Gegensatz zu Werkzeugen, die den Schwerpunkt auf die Visualisierung setzen, gibt es eine Reihe von Werkzeugen, die zur Transformation von Ereignisspuren

gedacht sind. Meist sind diese Werkzeuge programmierbar und bieten somit eine Möglichkeit, die vom Benutzer gewünschten Informationen aus einer Ereignisspur zu extrahieren und diese Abläufe zu automatisieren. Ein Beispiel für ein solches Analysewerkzeug ist EARL [16, 17].

EARL (Event Analysis and Recognition Language) ist eine Umgebung zur einfachen Erstellung neuer Werkzeuge für die Analyse von Message-Passing-Ereignisspuren. Die Werkzeuge werden dazu in Form von Skripten angelegt, die in der EARL-Spuranalysesprache, eine Erweiterung der Skriptsprache Tcl, geschrieben sind. Diese Skripte werden vom EARL-Interpreter ausgeführt und somit die Ereignisspur manipuliert. Als programmierbares Werkzeug eignet sich EARL z. B. zur automatischen Identifikation von Leistungsengpässen, zur Berechnung von benutzerdefinierten Leistungsindizes oder auch zur Durchführung von Meßreihen.

Kapitel 3

SALTO

SALTO (System for Assembly Language Transformation and Optimization) [14, 15] ist ein System zur Entwicklung von Werkzeugen, die auf der Manipulation von Assembler-Quellcodes basieren und wurde am IRISA (Institut de Recherche en Informatique et Systèmes Aléatoires) entwickelt. Das Ziel von SALTO ist es, dem Benutzer eine Umgebung zur Verfügung zu stellen, um Werkzeuge zur Leistungssteigerung von Programmen auf Assemblercode-Ebene unabhängig vom zugrundeliegenden Computersystem zu erstellen, wie beispielsweise Instrumentierer oder Assembler-Quellcode-Optimierer. SALTO beinhaltet selbst keine Algorithmen zur Instrumentierung oder Optimierung, sondern übernimmt vielmehr automatisiert einen großen Teil der Arbeit, die anfällt, wenn auf Assemblercode-Ebene gearbeitet wird, wie z. B. die Quellcodeanalyse oder die Konstruktion eines Abhängigkeitsgraphen.

Zur Nutzung von SALTO steht dem Benutzer eine objekt-orientierte Programmierschnittstelle zur Verfügung, mit deren Hilfe der Assembler-Quellcode manipuliert werden kann. Die Objekte beinhalten den kompletten Kontrollflußgraphen [35] und eine abstrakte Darstellung der benutzten Rechner-Architektur. Mit den Funktionen der Programmierschnittstelle kann auf diese Objekte leicht zugegriffen werden. Dies bietet eine komfortable Möglichkeit, Instrumentierungs- und Optimierungsalgorithmen zu implementieren.

Im folgenden soll nun SALTO vorgestellt werden. Dazu wird zunächst ein Überblick über das System gegeben und danach auf die wesentlichen Konzepte von SALTO eingegangen. Am Ende dieses Kapitels werden die Programmierschnittstelle und die verfügbaren Versionen beschrieben.

3.1 Systemüberblick

SALTO gliedert sich in drei Teile: einen Kern, einer Maschinenbeschreibung und einen Optimierungs- oder Instrumentierungsalgorithmus (Abb. 3.1).

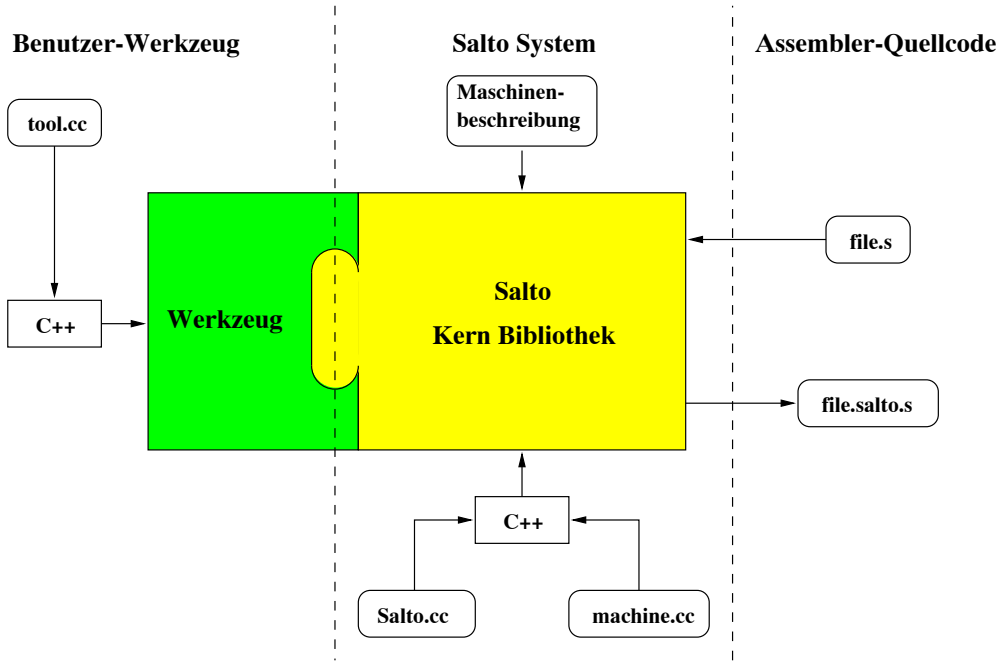


Abbildung 3.1: Systemüberblick

- Der Kern übernimmt Aufgaben wie die Analyse des Assemblercodes und der Maschinenbeschreibung. Ferner konstruiert er die interne Darstellung, die dem Benutzer schließlich über die Programmierschnittstelle zur Verfügung gestellt wird. Er besteht aus zwei Bibliotheken: zum einen eine Bibliothek, in der die vom Zielsystem unabhängigen Teile von SALTO, und zum anderen eine Bibliothek, in der die zielsystemspezifischen Programmteile implementiert sind. Dies hat zur Folge, daß die Werkzeuge, die auf SALTO aufsetzen, für jedes auf dem genutzten Basissystem gewünschte Zielsystem neu übersetzt werden müssen. Sollen beispielsweise auf einem Linux-System auf Intel-Basis Assemblercodes für Sparc und DEC Alpha Systeme bearbeitet werden, so muß das Programm für beide Zielarchitekturen separat übersetzt werden.

- Die Maschinenbeschreibung gliedert sich in vier Dateien, in denen die Systemkonfiguration und eine komplette Beschreibung des Befehlssatzes der Maschine mit Reservierungstabellen enthalten sind. Der Befehlssatz wird durch die Instruktionen der entsprechenden Assemblersprache definiert, während in den Reservierungstabellen die genutzten Resourcen der jeweiligen Instruktion aufgeführt sind.

- Der Optimierungs- oder Instrumentierungsalgorithmus wird vom Benutzer geliefert. Zur Anbindung der Algorithmen stellt das SALTO-System zwei Einstiegspunkte zur Verfügung: die Hauptfunktion `Salto_hook` und die Initialisierungsfunktion `Salto_init_hook`. Die Initialisierungsfunktion wird aufgerufen, nachdem SALTO die Argumente von der Kommandozeile analysiert hat. Das System liest danach die Maschinenbeschreibung und den Assembler-Quellcode.

Falls die interne Darstellung erfolgreich aufgebaut werden konnte, wird die Kontrolle an den Quellcode des Benutzers übergeben, indem die Funktion `Salto_hook` aufgerufen wird.

3.2 Konzepte und Eigenschaften von SALTO

In diesem Abschnitt soll auf die Konzepte und Eigenschaften von SALTO eingegangen werden. Dazu werden die in SALTO verwendeten Datenstrukturen vorgestellt, die sich in Abhängigkeit ihrer Aufgaben in zwei Gruppen unterteilen lassen. Zum einen werden Datenstrukturen zur Repräsentation des Assembler-Quellcodes benötigt und zum anderen zur Beschreibung der Maschine, insbesondere der genutzten Resourcen und Datenabhängigkeit zwischen Instruktionen.

Zur Darstellung der Assembler-Quellcodes wird in SALTO das Konzept der Kontrollflußgraphen verwendet, die dazu in eine geeignete Datenstruktur überführt werden.

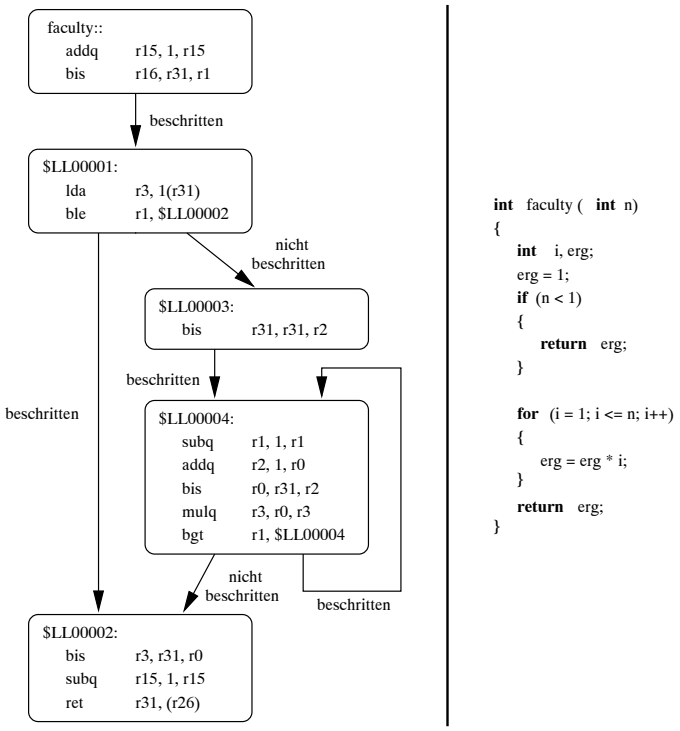


Abbildung 3.2: Kontrollflußgraph einer einfachen Prozedur

Die Maschinenbeschreibungen bestehen aus mehreren Dateien, in der die Architektur, die Syntax und lexikalische Struktur der Assemblersprache und die Semantik der Assemblerbefehle der jeweiligen Zielmaschinen beschrieben wird.

3.2.1 Kontrollflußgraphen

Ein in Assembler geschriebenes Programm kann als Ansammlung von Prozeduren angesehen werden, die eine Liste von Instruktionen beinhalten.

Innerhalb einer Prozedur können die Instruktionen zu *Basisblöcken* zusammengefaßt werden. Ein Basisblock ist eine sequentielle Abfolge von Instruktionen, d. h. er enthält höchstens einen Sprung- bzw. Verzweigungsbefehl und endet gegebenenfalls mit diesem.

So ergibt sich eine Einteilung des Assemblercodes in Prozeduren, Basisblöcken und Instruktionen. Zu jeder Prozedur, aber auch für das gesamte Programm, kann ein Kontrollflußgraph aufgestellt werden. In Abb. 3.2 ist dies für eine einfache Prozedur dargestellt, die in der Programmiersprache C geschrieben ist und die Fakultät zu einer Zahl n berechnet. Die Knoten des Graphen sind die Basisblöcke, während die Kanten die Ausführungsreihenfolge der Blöcke kennzeichnen. Die Beschriftung der Kanten gibt Auskunft über den Kontrollfluß des Programms, indem sie anzeigt, ob der Pfad, entsprechend der Verzweigungsinstruktion, beschritten bzw. nicht beschritten wird. Falls der Basisblock nicht mit einer Verzweigungsinstruktion endet, wird die Kante stets mit „beschritten“ beschriftet.

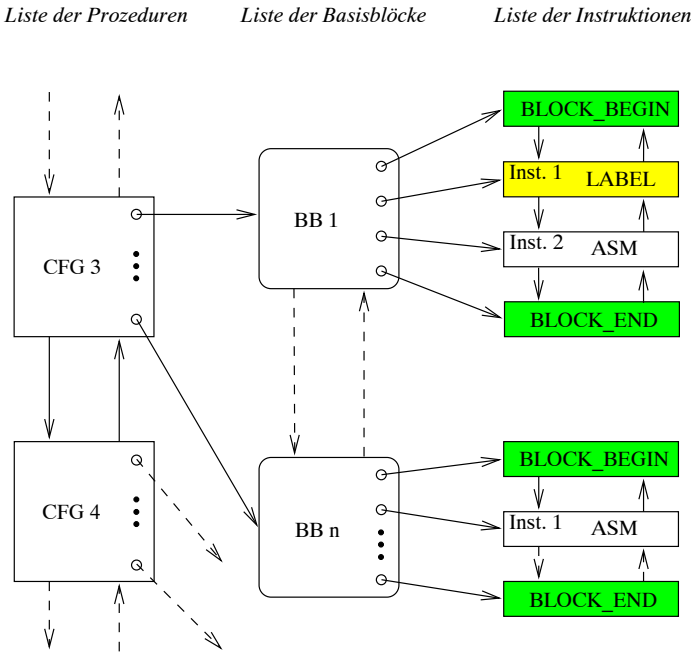


Abbildung 3.3: Organisation der Objektstruktur

Während SALTO den Assemblercode analysiert, wird eine Liste der Prozeduren generiert. Jeder Prozedur wird eine Liste der entsprechenden Basisblöcke zugeordnet, und zu jedem Basisblock wird die Liste seiner Instruktionen gespeichert. In Abb. 3.3 ist diese Objektstruktur skizziert. Die interne Darstellung der Objekte ist hinter der Programmierschnittstelle verborgen. Zusätzlich werden Markierer eingefügt, die den

Beginn bzw. das Ende eines Basisblocks anzeigen. Die Instruktionen innerhalb der Basisblöcke sind und bleiben im Syntax der Assemblersprache der jeweiligen Zielmaschine.

Nach der Syntaxanalyse des Assembler-Quellcodes kreiert SALTO einen Kontrollflußgraphen für jede Prozedur.

Eine bekannte Einschränkung der statischen Analyse von Assemblercodes ist, daß der Zielblock unbestimmbar ist, falls die Adresse einer Verzweigung von einem zu berechnenden Registerwert abhängt. In solchen Fällen läßt SALTO das Sprungziel offen.

3.2.2 Maschinenbeschreibungen

Die zweite Art von Datenstrukturen, die von SALTO bereitgestellt werden enthalten Informationen über die Resourcen, die eine Instruktion zu ihrer Ausführung benötigt. Eine Resource ist üblicherweise ein Register, eine funktionale Einheit oder der Speicher, aber auch ein beliebiges Stück Hardware, das zur Beschreibung des Verhaltens der Maschine notwendig ist. Jede Instruktion benötigt Resourcen für eine bestimmte Anzahl von Zyklen mit einem bestimmten Zugriffsmodus: lesen, schreiben oder benutzen.

Jede Instruktion wird durch eine Reservierungstabelle beschrieben, in der die benötigten Resourcen, der Zugriffsmodus und die benötigten Zyklen gespeichert sind. Diese Informationen werden gebraucht, um den Typ eines Datenflusses zwischen zwei Instruktionen zu bestimmen: RAW[1](Lesen-nach-Schreiben), WAW[2] (Schreiben-nach-Schreiben) und WAR[3] (Schreiben-nach-Lesen).

Der Speicher wird als eine einzige Resource angesehen, und alle Speicherzugriffe werden so behandelt, als ob sie auf den gleichen Speicherplatz zugreifen würden. Wenn zwei Speicherzugriffe vorliegen und einer von diesen schreibend ist, führt das in der im SALTO System verwendeten Methode zu einer Ahhängigkeit zwischen diesen beiden Instruktionen.

Da SALTO ein maschinenunabhängiges System sein soll, müssen die maschinenspezifischen Informationen in einer flexiblen Art beschrieben werden. Die Maschinenbeschreibungen sind deshalb in einer LISP ähnlichen Sprache verfaßt, die sich an RTL (register transfer language) [31] anlehnt.

Diese Beschreibungen enthalten Informationen über die Architektur und die Software-Umgebung, dies sind:

- Die Syntax und lexikalische Struktur der Assemblersprache

[1] read after write
[2] write after write
[3] write after read

- Die Liste der Resourcen der Architektur wie Register, Speicher, funktionale Einheiten, usw.
- Die Beschreibung der Instruktionen der Architektur, einschließlich vordefinierter Makros, die die Assemblersprache zur Verfügung stellt
- Informationen über die Semantik der Instruktionen
- Assembler-Direktiven (Pseudo-Instruktionen der Assemblerspache)

Die Basis-Einträge in der Beschreibung der Assemblersprache sind:

- Kommentare, einschließlich Kommentar Start- und Endmarkierungen
- Register-Bezeichnungen
- Operanden-Bezeichnungen
- Repräsentation der Adressierungsarten
- Mnemotechnische Bezeichnungen der Instruktionen, Makros und Assembler-Direktiven

Die *Resourcen*, die in einer Maschinenbeschreibung enthalten sind, sind Modell für die eventuell relevanten Hardware-Komponenten eines Optimierungprozesses. Deshalb sollten alle Resourcen, die auf die Ausführungszeit eines Quellcodes Einfluß haben, vertreten sein, um ihre Beschreibungen in einem Optimierungsalgorithmus nutzen zu können.

Resourcen mit gleichen Eigenschaften, wie allgemeine Register oder gleichartige funktionale Einheiten werden in *Resourceklassen* zusammengefaßt. Diese Klassen können genutzt werden, um „generische Resourcen“ zu bezeichnen, so daß z. B. ein beliebiges Register ausgewählt werden kann.

Referenzen auf Resourcen oder Resourceklassen werden bei der Beschreibung der *Reservierungstabellen* der individuellen Instruktionen und Operationen genutzt. Eine Reservierungstabelle beschreibt, welche Resourcen von einer Instruktion benötigt werden, sowie die Art der Nutzung und die Zyklen, in der die Instruktion die jeweiligen Resourcen verwendet.

Die Reservierungstabellen bilden die Basis für die Optimierung, indem sie das Auffinden von Konflikten zwischen Instruktionen und die Auswahl alternativer Resourcen ermöglichen. Jeder Instruktion wird in der Beschreibung des von SALTO zu erkennenden Instruktionssatzes eine Reservierungstabelle zugeordnet.

Instruktionssatzbeschreibungen enthalten den mnemotechnischen Namen jeder Instruktion oder Operation, ihr Format und die Reservierungstabelle, in der alle Resourcen aufgeführt sind, die zur Ausführung benötigt werden. Die Beschreibung jeder Makroinstruktion beinhaltet die entsprechende Sequenz der Basisinstruktionen bzw.

Operationen und die Regeln zur Konstruktion ihrer Operanden aus den Operanden der Makroinstruktion.

Die Instruktionssatzbeschreibung wird durch semantische Informationen zu den Instruktionen komplettiert. Es werden drei Arten von semantischen Randbedingungen[4] unterstützt : *verzögertes Laden* (delayed load), *verzögertes Verzweigen* (delayed branch) und *umgehendes Schreiben* (write by-pass).

Schließlich erlaubt die Beschreibung der Assembler-Direktiven die Analyse von Datendeklarationen, Daten- und Code-Verschachtelungen und Ausrichtungsinformationen.

Bevor SALTO die Maschinenbeschreibung analysiert, wird sie durch den Präprozessor `cpp` [30] geleitet. Jede der von `cpp` unterstützten Direktiven kann dabei genutzt werden, um die Beschreibung zu vereinfachen. Zumeist werden die Direktiven `#define` und `#include`, zur Verwendung von Makrodefinitionen und zum Aufbau eine hierarchische Ordnung, verwendet.

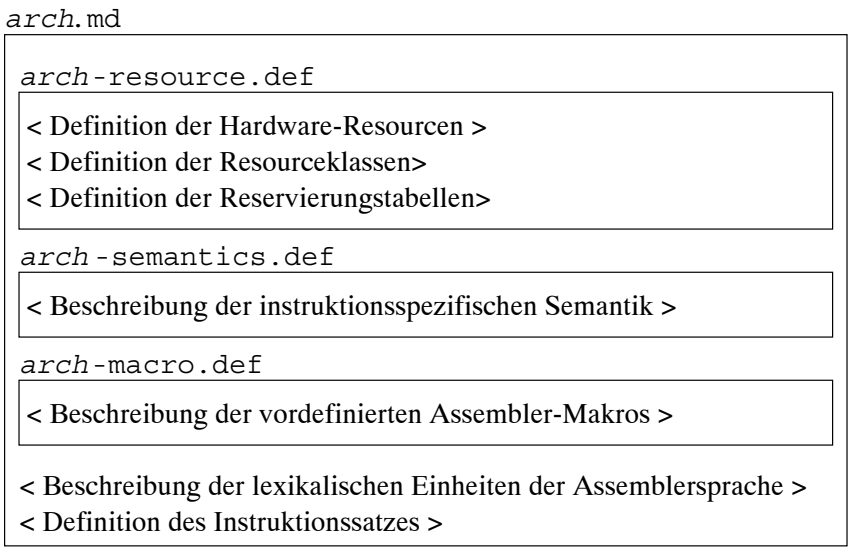


Abbildung 3.4: Organisation der Maschinenbeschreibungen

Die Beschreibung einer Architektur *arch* gliedert sich in vier Dateien (Abb. 3.4). Die Hauptdatei *arch*`.md` beinhaltet Inklusionsdirektiven, um die drei weiteren Dateien *arch*`-Resource.def`, *arch*`-macro.def` und *arch*`-semantic.def` einzubinden. Zusätzlich enthält sie die Beschreibung der lexikalischen Struktur der Assemblersprache sowie die Definition des Instruktionssatzes, der von SALTO erkannt werden soll. Die Datei *arch*`-Resource.def` beinhaltet die Definition der Hardware-Resourcen, Resourceklassen und Reservierungstabellen, während die Datei *arch*`-macro.def` Expansionsregeln für die vordefinierten Assembler-Makros bereitstellt. In der Datei *arch*`-semantics.def` sind die instruktionsspezifischen, semantischen Randbedingungen definiert.

[4] constraints

3.3 Programmierschnittstelle

Zum Zugriff auf die Datenstrukturen bzw. Objektstrukturen wird von SALTO eine umfangreiche, objekt-orientierte Programmierschnittstelle bereitgestellt. Die Programmierschnittstelle besteht aus einigen globalen Funktionen, Objektklassen und einigen SALTO-spezifischen Datentypen.

Die globalen Funktionen dienen vor allem zum Einstieg in die Objektstrukturen von SALTO und zur Manipulation der Liste der Prozeduren, die in dem Assembler-Quellcode vorhanden sind. Es existieren z. B. Funktionen, die das Auffinden der Position eines Objektes (Prozedur bzw. Kontrollflußgraph, Basisblock, Instruktion) innerhalb seines Containers (Assembler-Quellcode, Prozedur bzw. Kontrollflußgraph, Basisblock) ermöglichen oder Funktionen, die die Anzahl der Instruktionen oder Prozeduren innerhalb des Assembler-Quellcodes ermitteln. Eine wichtige Funktion ist `CFG *getCFG(unsigned int pos)`. Diese Funktion liefert einen Zeiger auf das Kontrollflußgraphen-Objekt der *pos*-ten Prozedur innerhalb des Quellcodes und ermöglicht so den Zugriff auf den Kontrollflußgraphen dieser Prozedur.

Die Klasse `CFG` wird genutzt, um die Prozeduren als Kontrollflußgraphen zu repräsentieren. In ihr sind Funktionen enthalten, um Kontrollflußgraphen zu manipulieren. Unter anderem können der Name der Prozedur abgefragt, die Anzahl der enthaltenen Basisblöcke ermittelt und Basisblöcke sowie Kanten zwischen diesen angelegt und gelöscht werden. Um Zugriff auf einen Basisblock zu erhalten, wird die Funktion `BB *CFG::getBB(unsigned int pos)` genutzt. Sie liefert einen Zeiger auf das Objekt des *pos*-ten Basisblocks zurück und erlaubt somit den Zugriff auf in diesem Basisblock enthaltene Instruktionen.

Die Objekte der Klasse `BB` repräsentieren die Basisblöcke des Assembler-Quellcodes. In diesen Basisblöcken sind die Instruktionen des Quellcodes enthalten und können durch die Funktionen manipuliert werden, die in dieser Klasse vorhanden sind. Es stehen z. B. Funktionen zur Bestimmung der Anzahl der in diesem Basisblock enthaltenen Instruktionen, zum Löschen und Einfügen von Instruktionen, aber auch zum Vertauschen der Position von Instruktionen innerhalb des Basisblocks zur Verfügung. Eine wichtige Funktion ist `INST *BB::getInstruction(unsigned int pos)`, die einen Zeiger auf das Objekt der *pos*-ten Instruktion innerhalb des Basisblocks zurückliefert und so Veränderungen an dieser Instruktion ermöglicht.

Die Klasse `INST` repräsentiert die Instruktionen des Assembler-Quellcodes. In ihr sind Funktionen enthalten, mit denen es möglich ist, diese Instruktionen zu verändern oder auszuwerten. Vorhanden sind beispielsweise Funktionen, mit denen es möglich ist, den Typ (Sprungmarke, Pseudo-Instruktion, Verzweigungsinstruktion usw.) der Instruktion zu bestimmen und Funktionen, um neue Assemblerbefehle zu kreieren. Zusätzlich enthält diese Klasse Funktionen, die Informationen über die Instruktion liefern, wie z. B. ihre mnemotechnische Bezeichnung, die Anzahl ihrer Operanden, die Anzahl der genutzten Resourcen und die Art des Zugriffs auf diese Resourcen. Die Funktion `INST *newAsm(char opcode, unsigned int numOps = 0,`

. . .) dient zur Erzeugung neuer Assembler-Instruktionen, wobei *opcode* die mnemotechnische Bezeichnung der Instruktion ist, die von der jeweiligen Assemblersprache vorgegeben und somit systemspezifisch ist, und *numOps* die Anzahl der Operanden. Weitere wichtige Funktionen sind `OperandInfo &INST::getOperand(unsigned int pos)`, die Informationen über die Operanden einer Instruktion extrahiert und die Funktionen, um Zugriff auf die genutzten Resourcen der Instruktion zu erhalten, wie beispielsweise `res_ref *INST::getUse(int pos)`.

Die Klassen `OperandInfo` und `res_ref` beinhalten wiederum Funktionen, um die jeweiligen Einträge zu verändern und Informationen über diese abzufragen. Dabei sind in der Klasse `OperandInfo` die Operanden einer Instruktion modelliert, während die Objekte der Klasse `res_ref` Informationen über die Resourcen enthalten, auf die bei der Ausführung einer Instruktion zugegriffen wird.

Weitere Klassen, die von SALTO bereitgestellt werden, sind `reserv_table1` und `SaltoAttribute`. Während mit der Klasse `reserv_table1` die Reservierungstabellen verwaltet werden, dient die Klasse `SaltoAttribute` dazu, Basisblöcke und Instruktionen mit beliebigen Informationen zu kommentieren.

Neben den globalen Funktionen und den Objektklassen werden fünf SALTO-spezifische Datentypen genutzt, um bestimmte Eigenschaften der Objekte zu repräsentieren.

Der Aufzählungstyp `enum cft_type` zeigt die Art des Kontrollflusses einer bedingten Verzweigung an. Es existieren nur zwei Werte `TAKEN` und `NOT_TAKEN`, die dem Erfolg bzw. Mißerfolg der Bedingung entsprechen.

Der Typ `enum xNode_Type` dient zur Repräsentation der Kategorie, zu der eine Instruktion gehört. Die möglichen Werte dieses Aufzählungstyps sind in Tabelle 3.1 aufgeführt.

Wert	Beschreibung
`X_ASM_TYPE`	Assembler-Instruktion
`X_MACRO_TYPE`	Assembler-Makro
`X_LABLE_TYPE`	Sprungmarke
`X_EQUAL_TYPE`	Konstanten-Deklaration
`X_PSEUDO_TYPE`	Assembler-Pseudo-Instruktion
`X_INFO_TYPE`	von SALTO generierte Anmerkung

Tabelle 3.1: Der Aufzählungstyp `enum xNode_Type`

Der Aufzählungstyp `enum dependence` charakterisiert die Art der Abhängigkeit zwischen zwei Instruktionen. Dieser Typ kann vier Werte annehmen: `None` (keine gemeinsamen Daten), `RAW` (Lese-nach-Schreibe-Abhängigkeit), `WAR` (Schreibe-nach-Lese-Abhängigkeit) und `WAW` (Schreibe-nach-Schreibe-Abhängigkeit).

Während der Aufzählungstyp `enum res_ref_type` die Art der Referenzierung, wie z. B. Anforderung einer Basis-Resource oder eines Blocks von Registern, wider-

spiegelt, dient der Aufzählungstyp `enum res_type` zur Bezeichnung der Hardware-Resourcen und kann drei mögliche Werte annehmen: `REGISTER_RTYPE` (Hardware Register), `FUNCT_UNIT_RTYPE` (funktionale Einheit) und `MEMORY_RTYPE` (Speicher).

3.4 Versionen und Verfügbarkeit

SALTO liegt zur Zeit in der Version 1.3 vor. Um das SALTO-System zu installieren, wird neben dem erforderlichen Speicherplatz lediglich ein moderner C++ Compiler benötigt, so daß SALTO auf allen aktuellen Rechnern und Betriebssystemen lauffähig sein sollte. Zur Installation auf das System CRAY T3E müssen einige Anpassungen vorgenommen werden, die in Anhang A beschrieben sind.

Erfolgreich getestet wurde SALTO auf:

UltraSparc - Solaris 2.5.1
SparcStation5 - SunOS 4.14
SparcStation20 - SunOS 5.6
Pentium75 - Linux 2.0.30 (Redhat 4.2)
UltraSparc - Linux 2.1.89 (UltraPenguin 1.0)
HP 9000 - HPUX B.10.20
Pentium - Windows NT 4.0
CRAY T3E - Unicos/mk 2.0.3.24

In der aktuellen Version existieren noch einige Probleme mit dem GNU C++-Compiler g++ in der Version 2.8.1, so daß auf eine ältere Version bzw. andere Compiler ausgewichen werden muß.

Maschinenbeschreibungen liegen zur Zeit für die folgenden Architekturen bzw. Prozessoren vor:

Mips, Sparc, DEC Alpha und CRAY T3E.

Diese Beschreibungen befinden sich in verschiedenen Stadien ihrer Entwicklung. Während beispielsweise die Beschreibung der Sparc-Architektur sehr fortgeschritten und stabil ist, ist die Maschinenbeschreibung der CRAY T3E eher in einem experimentellen Stadium, obwohl sie im Rahmen dieser Diplomarbeit entscheidend weiterentwickelt wurde.

Gleiches gilt für die maschinenabhängigen Programmteile des SALTO-Systems, die sehr eng mit den Maschinenbeschreibungen verknüpft sind.

Ein weiteres Problem der Maschinenbeschreibungen ist, daß in ihnen neben der Maschine auch die entsprechende Assemblersprache definiert ist. Da es jedoch verschiedene Assemblersprachen auf ein und derselben Maschine gibt, lassen sich ohne Schwierigkeiten nur Assembler-Quellcodes der definierten Assemblersprache

mit SALTO analysieren. In den Maschinenbeschreibungen wird zumeist die GNU-Assemblersprache der entsprechenden Architektur definiert, so daß sich nur von GNU-Compilern erzeugte Assembler-Quellcodes problemlos nutzen lassen.

Es gibt zwei Möglichkeiten, um einen Assembler-Quellcode, der von anderen Compilern erzeugt wurde, mit SALTO zu nutzen. Zum einen kann der Assembler-Quellcode auf die Form der GNU-Assemblersprache transformiert werden, zum anderen kann für jede weitere genutzte Assemblersprache eine neue Maschinenbeschreibung erstellt und ein entsprechender systemabhängiger Programmteil implementiert werden.

So sind die beiden Varianten der Maschinenbeschreibung für die Sparc-Architektur zu erklären. Eine Variante ist für die von GNU-Compilern erzeugten Assembler-Quellcodes bestimmt, während die andere Variante für Compiler der Firma Sun Microsystems Inc. erstellt wurde.

Diese Methode, das Problem mit den verschiedenen Assemblersprachen zu umgehen, hat einen wesentlichen Nachteil: Das Werkzeug, das auf SALTO aufbaut, muß auf einem Basissystem nicht nur für alle Zielsysteme übersetzt werden, sondern auch noch für alle verwendeten Assemblersprachen bzw. Compiler.

Kapitel 4

VAMPIR-Tracebibliothek

Die VAMPIR-Tracebibliothek *libtrace.a* wurde für das Instrumentierungswerkzeug `vampir.inst` [26] entwickelt. Diese Bibliothek kann jedoch auch zur manuellen Quellcode-Instrumentierung genutzt werden.

Die Bibliothek liefert neben Wrapper-Funktionen, die Massage-Passing-Funktionsaufrufe ersetzen, Funktionen, die zur Instrumentierung von Benutzerfunktionen dienen, um spezielle Ereignis-Records in die Ereignisspur zu schreiben.

Des weiteren ist es möglich, das Verhalten der Bibliothek durch einige Umgebungsvariablen zu steuern.

4.1 Eigenschaften der Bibliothek

Um die VAMPIR-Tracebibliothek nutzen zu können, müssen bestimmte Voraussetzungen erfüllt sein. Die Bibliothek benötigt eine Symboldatei, in der eine Beziehung zwischen den Funktionsnamen der Funktionen, die instrumentiert werden sollen, einer Funktionsnummer und dem Namen der Gruppe, zu der die Funktion gehören soll, hergestellt wird. Diese Symboldatei besitzt den Namen „`instrc`“ und beinhaltet pro Zeile die Beschreibung einer einzigen Benutzerfunktion. Eine solche Zeile hat den Aufbau:

<Nummer> <Funktionsname> <Gruppenname> [weitere Direktiven]

In der ersten Spalte muß eine Nummer angegeben werden, die eindeutig der Funktion zugeordnet ist. Die zweite Spalte enthält den Funktionsnamen, während in der dritten Spalte der Name der Gruppe steht, zu der diese Funktion gehören soll. Der Vorteil, Funktionen in Gruppen einzuteilen, ist, daß mit VAMPIR diesen Gruppen unterschiedliche Farben zugeordnet werden können und somit eine übersichtliche Darstellung der Ereignisspuren ermöglicht wird. Die einzelnen Spalten müssen durch mindestens ein Leerzeichen getrennt werden.

Die vierte Spalte dient zur Steuerung der Erzeugung der Ereignisspur während der Laufzeit des instrumentierten Programms. Wird in dieser Spalte die Direktive „OFF“ eingetragen, so werden für die entsprechende Funktion keine Ereignis-Records in die Ereignisspur geschrieben. Dazu muß das Programm nicht erneut instrumentiert und übersetzt werden.

Für jede Wrapper-Funktion, die in der VAMPIR-Tracebibliothek zu finden ist, sollte ebenfalls ein Eintrag in der Symboldatei „`instrc`“ vorhanden sein. Diese Einträge haben einen leicht modifizierten Aufbau. Es wird in der zweiten Spalte der Name der Wrapper-Funktion eingetragen. Als Direktive wird „REPLACE <Original-Funktionsname>“ genutzt, um zu kennzeichnen, daß die Funktionsaufrufe durch Aufrufe der Wrapper-Funktion ersetzt werden sollen. Setzt man ein „OFF“ vor die Direktive „REPLACE“, so werden keine Records für diese Funktionsaufrufe generiert.

Das Verhalten der Bibliothek kann durch eine Reihe von Umgebungsvariablen beeinflußt werden. So kann beispielsweise mit der Variable `VAMPIR_BUFFERSIZE` die Größe des anzulegenden Speichers, der als Puffer für die Ereignis-Records genutzt wird, manipuliert werden. Mit der Variable `VAMPIR_SYSCOUNT` kann die Größe der Symboltabelle verändert werden, in der die Bibiliothek alle Nummern und Funktionsnamen aus der Symboldatei „`instrc`“ speichert, da diese Tabelle normalerweise nur 2048 Einträge aufnimmt.

Weitere Umgebungsvariablen dienen zur Steuerung der Ausgabe der Bibliothek und zur Veränderung der Pfade der genutzten und zu erzeugenden Dateien.

4.2 Programmierschnittstelle

Neben den Wrapper-Funktionen stellt die VAMPIR-Tracebibliothek Funktionen zur Kontrolle der Generierung von Ereignisspuren und zur Erzeugung von Ereignis-Records zur Verfügung.

Zum Starten bzw. Beenden der Aufzeichnung von Ereignis-Records dienen die beiden Funktionen `vptsetup` und `vptflush`. Üblicherweise wird die Funktion `vptsetup` zu Beginn eines Programms aufgerufen, während die Funktion `vptflush` vor dem Verlassen des Programms aufgerufen wird, um die Ereignisspur korrekt abzuschließen.

Mit der Funktion `vptformat` kann das Format der Ereignisspur, ASCII oder Binär, gewählt werden. Die Funktion `vptlimit` wird zur Festlegung der maximalen Anzahl der zu schreibenden Ereignis-Records genutzt. Beide Funktionen müssen sofort nach der Funktion `vptsetup` aufgerufen werden.

Die Funktionen `vpton` und `vptoff` können genutzt werden, um die Aufzeichnung der Ereignis-Records zu starten bzw. zu stoppen, um so bestimmte Programmteile, die für die spätere Analyse uninteressant sind, vom Tracing auszuschließen.

Schließlich dienen die Funktionen `vptenter`, `vptleave`, `tracesend`, `tracerecv` `vptleavename` und `vptentername` zur Generierung von Ereignis-Records.

Die Funktionen `vptenter` und `vptentername` legen ein Ereignis-Record an, in dem der Eintritt in eine Funktion protokolliert wird. Deshalb müssen diese Funktionen zu Beginn einer Funktion aufgerufen werden. Der Unterschied zwischen den Funktionen ist, daß `vptenter` die Nummer der Funktion entsprechend der Symboldatei „`instrc`“ als Argument benötigt, während die Funktion `vptentername` als Argument einen Funktionsnamen erwartet. Sollte der Funktionsname nicht in der Symboldatei „`instrc`“ vorhanden sein, so wird ein Eintrag mit diesem Funktionsnamen erzeugt und an die Symboldatei angehängt.

Die Funktionen `vptleave` und `vptleavename` legen entsprechend ein Ereignis-Record an, in dem das Verlassen einer Funktion protokolliert wird und müssen deshalb vor allen möglichen Stellen, an denen die Funktion beendet werden kann, aufgerufen werden. Zu jedem Aufruf der Funktionen `vptenter` und `vptentername` sollten die entsprechenden Aufrufe der Funktionen `vptleave` und `vptleavename` vorhanden sein.

Zur Generierung von Ereignis-Records, die Kommunikationsereignisse protokollieren, zu denen es keine Wrapper-Funktion gibt, werden die Funktionen `tracesend` und `tracerecv` verwendet. Als Argument erwarten diese Funktionen die Nummer des Prozessors bzw. Prozesses, der die Nachricht versendet bzw. empfängt, den Nachrichten-Typ, die Länge der Nachricht und den Kommunikator, falls es sich um eine MPI-basierte Nachricht handelt. Die Bibliothek ist nicht in der Lage zu überwachen, ob es zu jedem Sende- ein entspechendes Empfangsereignis gibt. Deshalb muß darauf geachtet werden, daß zu jedem Sende- auch das entsprechende Empfangsereignis protokolliert wird.

4.3 Versionen und Verfügbarkeit

Die VAMPIR-Tracebibliothek steht in verschiedenen Versionen für die jeweils genutzten Message-Passing Varianten zur Verfügung. In diesen Versionen sind neben den oben beschriebenen Funktionen jeweils die Wrapper-Funktionen für die jeweilige Message-Passing Variante enthalten. So werden z. B. in der Bibliotheksdatei *libmpitrace.a* nur Wrapper-Funktionen für die Funktionen der MPI-Bibliothek bereitgestellt. Des weiteren gibt es spezielle Versionen für die verwendeten Programmiersprachen, so stellt z. B. die Bibliotheksdatei *libmpi_cvers_trace.a* nur die Wrapper-Funktionen für die MPI-Funktionen in einer C-Version zur Verfügung. In der Bibliotheksdatei *libtrace.a* sind jedoch alle Wrapper-Funktionen, sowohl die C-Versionen als auch die Fortran-Versionen, für alle Message-Passing Varianten enthalten, die unterstützt werden. Zu beachten ist, daß dabei nicht für alle Message-Passing-Funktionen der jeweiligen Variante auch Wrapper-Funktionen zur Verfügung stehen, und daß nicht für jede Wrapper-Funktion, die als Fortran-Version verfügbar ist, auch eine C-Version existiert.

Obwohl die VAMPIR-Tracebibliothek im entsprechenden Verzeichnis einer VAMPIR Installation zu finden ist, ist sie lediglich auf den Systemen CRAY T3D, CRAY T3E, und Intel Paragon vollständig implementiert. Auf anderen Systemen sind die Funktionen der Bibliothek zwar definiert, aber teilweise nicht vollständig ausprogrammiert, und somit kann die Bibliothek auf diesen Systemen nicht zur Generierung von Ereignisspuren eingesetzt werden.

Auf den Systemen CRAY T3D und CRAY T3E werden die Kommunikationsbibliotheken SHMEM[1] [8], PVM und MPI unterstützt, während für die Intel Paragon Wrapper-Funktionen für NX [19], PVM und MPI zur Verfügung stehen.

[1]Shared Memory

Kapitel 5

Instrumentierung von Assembler-Quellcodes

Die Instrumentierung des Assembler-Quellcodes eines Programms stellt neben der Instrumentierung des eigentlichen Quellcodes und des Objektcodes eine weitere Möglichkeit dar, die nötigen Änderungen zur Erzeugung von Ereignisspuren an dem Programm vorzunehmen.

Zu diesem Zweck wurde im Rahmen dieser Diplomarbeit das System SALTO so erweitert, daß es auf einfache Weise möglich ist, zielsystemunabhängig Funktionsaufrufe an geeigneten Stellen im Assembler-Quellcode zu integrieren und Funktionsaufrufe innerhalb des Assembler-Quellcodes zu manipulieren. Diese Erweiterung wurde exemplarisch genutzt, um einen Instrumentierer zu implementieren, der den Assembler-Quellcode eines Programms so verändert, daß Routinen der VAMPIR-Tracebibliothek aufgerufen und somit VAMPIR-Ereignisspuren für dieses Programm erzeugt werden.

Da auf SALTO basierende Werkzeuge stets nur einen Assembler-Quellcode bearbeiten können, aber reale Programme in der Regel aus mehreren Quelldateien bestehen, wurde ein System entwickelt, das die Instrumentierung eines vollständigen Programms weitestgehend automatisiert. Dieses System basiert auf einem Perl-Skript [24], das die zur Instrumentierung nötigen Compiler- und Instrumentiereraufrufe koordiniert und die Verwaltung der benötigten Symboldateien übernimmt.

Zur Steuerung der Instrumentierung können sowohl Instrumentierungs-Kontrollskripte sowie eine graphische Benutzeroberfläche eingesetzt werden. Die notwendigen Schritte zur Installation des gesamten Systems sind in Anhang B beschrieben.

In diesem Kapitel werden die einzelnen Komponenten dieses Instrumentierungssystems vorgestellt sowie auf Probleme und Besonderheiten dieses Ansatzes der Instrumentierung von Programmen eingegangen.

5.1 Die Instrumentierungsbibliothek ASINST

Die grundlegende Idee der Instrumentierung von Assembler-Quellcodes ist es, mit den genutzten Compilern statt Objektcodes Assembler-Quellcodes zu erzeugen. Dies ist in der Regel durch die Nutzung einer Option der Compiler leicht möglich. Anschließend werden die Assembler-Quellcodes instrumentiert und danach durch die entsprechenden Assembler die Objektcodes des Programms generiert. Diese instrumentierten Objektcodes müssen dann noch mit der genutzten Tracebibliothek gebunden werden, um ein lauffähiges und instrumentiertes Programm zu erhalten (Abb. 5.1).

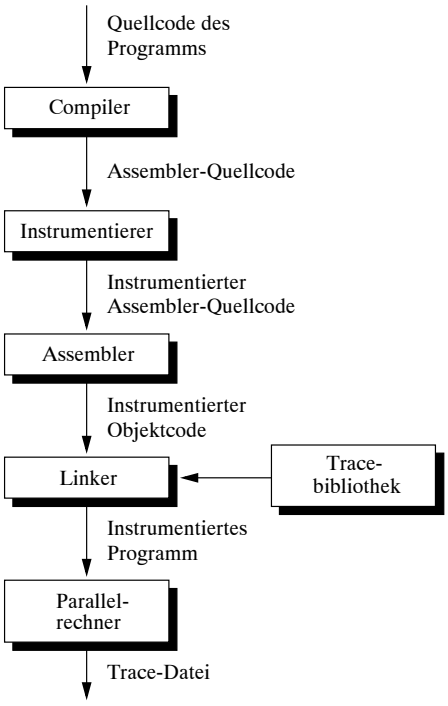


Abbildung 5.1: Instrumentierung des Assembler-Quellcodes

Wie schon in Kapitel 3 beschrieben, ist SALTO ein System zur Entwicklung von Werkzeugen, die auf der Manipulation von Assembler-Quellcodes basieren. Dabei stellt SALTO keine Optimierungs- oder Instrumentierungsalgorithmen bereit, sondern bietet über eine objekt-orientierte Programmierschnittstelle eine große Anzahl von Funktionen, um solche Algorithmen zu erstellen.

Um möglichst einfach Assembler-Quellcode-Instrumentierungswerkzeuge zu implementieren, wurde SALTO um einige Funktionen erweitert, die es zum einen erlauben, an bestimmten Stellen im Assembler-Quellcode Aufrufe von Funktionen einzufügen und zum anderen Funktionsaufrufe innerhalb des Assembler-Quellcodes zu verändern, indem die Namen der aufzurufenden Funktionen durch andere Funktionsnamen ersetzt werden können. Diese Funktionen sind in einer Bibliothek zusammengefaßt, um sie dem Benutzer leichter zugänglich zu machen.

5.1.1 Architektur und Implementierung

Wie in Abb. 5.2 zu sehen ist, baut die Bibliothek ASINST auf SALTO auf und kann mit den Funktionen von SALTO zur Implementierung von Benutzer-Werkzeugen, insbesondere Instrumentierungswerkzeugen, genutzt werden. Wie SALTO besteht diese Bibliothek aus einem zielsystemunabhängigen und einem zielsystemabhängigen Teil, der jedoch möglichst klein gehalten wurde, um die Portierung der Bibliothek zu erleichtern.

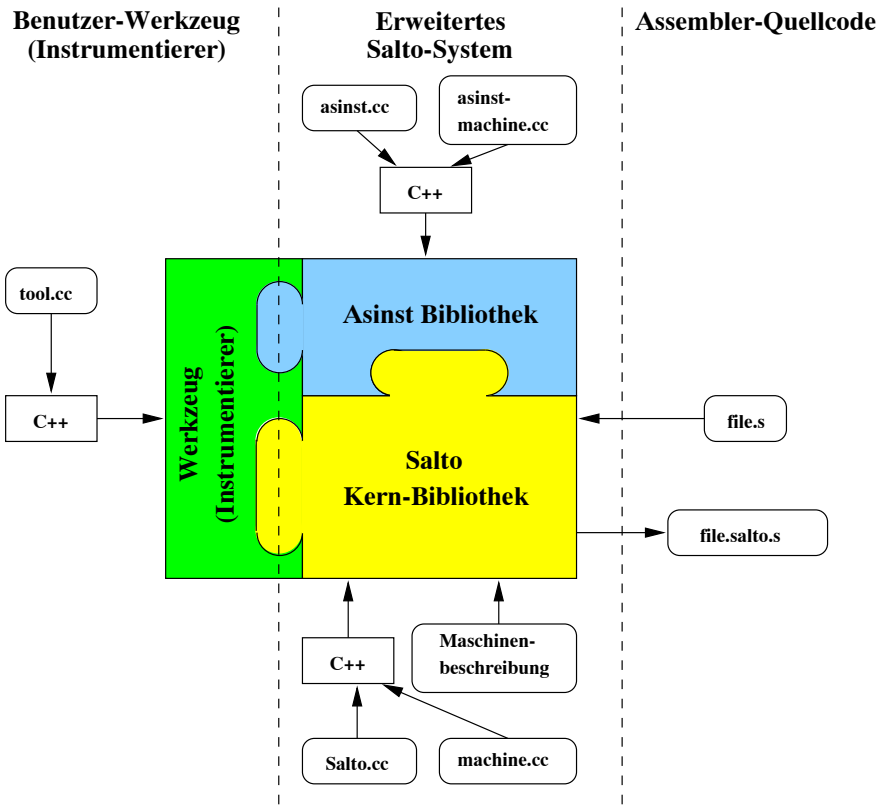


Abbildung 5.2: Implementierung der Bibliothek ASINST

Im zielsystemunabhängigen Teil sind die Funktionen zur Änderung von Funktionsaufrufen und die Funktionen, um geeignete Stellen zur Instrumentierung im Assembler-Quellcode zu finden, implementiert. Diese Stellen sind der Anfang und das Ende des Programms, einer Prozedur oder eines Basisblocks. Um die genaue Position zu lokalisieren, an denen neue Instruktionen in den Assembler-Quellcode eingefügt werden können, sind zielsystemspezifische Funktionen notwendig, die im zielsystemabhängigen Teil der Bibliothek implementiert sind. Dort ist auch die Funktion untergebracht, die die zum Aufruf einer Funktion benötigten Instruktionen im Syntax der Zielsystem-Assemblersprache liefert.

Zur Integration eines Funktionsaufrufs in einen Assembler-Quellcode, müssen alle globalen Register zuvor gerettet und nach Rückkehr aus der aufgerufenen Funktion wiederhergestellt werden. Da moderne Prozessoren eine Vielzahl von Registern besit-

zen, ist es notwendig, diese Routinen als separate Objektdatei für jedes unterstützte Zielsystem bereitzustellen, um den Assembler-Quellcode des zu instrumentierenden Programms durch die Instrumentierung nicht zu sehr aufzublähen. Diese Assemblerroutinen müssen, neben den neu im instrumentierten Assembler-Quellcode aufgerufenen Funktionen, zu den Assembler-Quellcodes hinzu gebunden werden, um ein lauffähiges Programm zu erhalten.

Die Bibliothek steht zur Zeit für das System CRAY T3E und für auf Sparc Prozessoren basierenden Systemen zur Verfügung und wurde auf einer CRAY T3E unter Unicos/mk 2.0.3.24 sowie auf einer SparcStation20 unter SunOS 5.6 getestet.

5.1.2 Programmierschnittstelle

Insgesamt erweitert die Bibliothek ASINST das SALTO-System um 12 Funktionen. Ähnlich wie mit dem Werkzeug ATOM [13], das allerdings auf dem Objektcode eines Programms arbeitet, ist es nun möglich, sich durch den Assembler-Quellcode zu bewegen und neue Funktionsaufrufe an bestimmten Stellen einzufügen. Dabei lassen sich die Funktionen der Bibliothek ASINST in drei Gruppen einteilen.

Die erste Gruppe besteht aus zwei Funktionen, die Kontrollaufgaben übernehmen. Mit der Funktion `CheckTargetMachine` läßt sich überprüfen, ob die geladene Maschinenbeschreibung mit der im Benutzer-Werkzeug verwendeten, systemabhängigen SALTO-Bibliothek übereinstimmt. Die Funktion `AddProtoType` dient zur Anmeldung einer Funktion, zu der Funktionsaufrufe in den Assembler-Quellcode eingefügt werden sollen. Neben dem Namen der Funktion muß die Anzahl ihrer Parameter angegeben werden. Als Einschränkung dürfen nur Funktionen verwendet werden, die höchstens sechs Integer-Parameter besitzen, doch sollten solche Funktionen ausreichen, um geeignete Trace-Funktionen zu implementieren.

Die zweite Gruppe von Funktionen dient zur Änderung von Funktionsaufrufen innerhalb des Assembler-Quellcodes. Mit der Funktion `ChangeCallInProg` kann im gesamten Assembler-Quellcode der Funktionsname in einem Funktionsaufruf durch einen anderen ersetzt werden. Die Ersetzung läßt sich mit der Funktion `ChangeCallInFunc` auf eine Prozedur des Assembler-Quellcodes beschränken, wobei ein Zeiger auf das Kontrollflußgraphen-Objekt dieser Prozedur als Argument übergeben werden muß. Beide Funktionen dienen zur Nutzung von Wrapper-Funktionen, so daß z. B. `ChangeCallInProg("MPI_Send", "MPI_Send_rep")` alle Aufrufe der Funktion `MPI_Send` durch Aufrufe der Funktion `MPI_Send_rep` im gesamten Assembler-Quellcode ersetzt, wobei die Funktionsnamen als Zeichenketten zu übergeben sind. Falls mindestens eine Ersetzung erfolgt ist, liefern diese Funktionen eine Eins, anderenfalls eine Null zurück.

In der dritten Gruppe sind Funktionen enthalten, um neue Funktionsaufrufe in einen Assembler-Quellcode zu integrieren. Es stehen Funktionen, die einen neuen Funktionsaufruf am Anfang bzw. am Ende eines Programms, einer Prozedur sowie eines Basisblocks einfügen, zur Verfügung. Des weiteren existiert eine Funktion,

Name	Beschreibung
`AddCallAtBeginOfProg`	Fügt einen Funktionsaufruf am Anfang eines Programms ein
`AddCallAtEndOfProg`	Fügt einen Funktionsaufruf am Ende eines Programms ein
`AddCallAtExit`	Fügt einen Funktionsaufruf vor Aufrufen der Systemfunktion `exit` ein
`AddCallAtAbort`	Fügt einen Funktionsaufruf vor Aufrufen der Systemfunktion `abort` ein
`AddCallAtBeginOfFunc`	Fügt einen Funktionsaufruf am Anfang einer Prozedur ein
`AddCallAtEndOfFunc`	Fügt einen Funktionsaufruf am Ende einer Prozedur ein
`AddCallAtBeginOfBasicBlock`	Fügt einen Funktionsaufruf am Anfang eines Basisblocks ein
`AddCallAtEndOfBasicBlock`	Fügt einen Funktionsaufruf am Ende eines Basisblocks ein

Tabelle 5.1: Weitere Funktionen der Bibliothek ASINST

die Funktionsaufrufe vor dem Verlassen des Programms durch die Systemfunktion `exit` einfügt, sowie eine Funktion, die Funktionsaufrufe vor dem Abbruch des Programms durch die Systemfunktion `abort` integriert. Da die in der Bibliothek enthaltenen Funktionen systemunabhängig sein sollen, werden keine systemspezifischen Funktionen zum Abbruch eines Programms berücksichtigt. In Tabelle 5.1 sind die Funktionen dieser Gruppe mit einer kurzen Beschreibung aufgeführt. Alle Funktionen benötigen als Argument den Namen der aufzurufenden Funktion, sowie deren Argumente und liefern eine Null zurück, falls der Funktionsaufruf nicht in den Assembler-Quellcode aufgenommmen werden konnte.

Zu beachten ist, daß ein Programm bzw. Assembler-Quellcode von innen nach außen instrumentiert werden muß, da SALTO Änderungen an dem Assembler-Quellcode sofort in die interne Darstellung übernimmt. Der Beginn einer Prozedur stellt gleichzeitig den Beginn des ersten Basisblocks dieser Prozedur dar. Wird ein Funktionsaufruf zu Beginn der Prozedur eingefügt, so ist dieser Aufruf folglich Bestandteil des ersten Basisblocks. Ein Funktionsaufruf, der zu einem späteren Zeitpunkt am Beginn dieses Basisblocks integriert wird, steht dann vor dem Aufruf der Funktion, die eigentlich am Beginn der betreffenden Prozedur aufgerufen werden sollte. Also müssen die Funktionsaufrufe zuerst in Basisblöcken, dann in Prozeduren und zuletzt an dem Anfang bzw. an dem Ende des Programms aufgenommen werden, um die korrekte Reihenfolge der Funktionsaufrufe zu gewährleisten.

5.1.3 Beispiel für die Nutzung der Bibliothek ASINST

In diesem Abschnitt soll nun als Beispiel für die Nutzung der Bibliothek ASINST mit dem SALTO-System ein kleines Instrumentierungswerkzeug mit dem Namen `ptrace` vorgestellt werden. Dieses Werkzeug soll einen Assembler-Quellcode so instrumentieren, daß die Reihenfolge der Aufrufe der in dem Assembler-Quellcode enthaltenen Prozeduren ausgegeben wird.

Dazu muß in dem Assembler-Quellcode zu Beginn jeder Prozedur ein Funktionsaufruf eingefügt werden. Diese aufzurufende Funktion sorgt für die Ausgabe, daß die entsprechende Prozedur aufgerufen wurde.

```
01 /* Program : ptrace */
02 #include <stdio.h>
03 #include "asinst.h"
04 #include "salto.h"
05
06 void Salto_hook() {
07       int num_of_proc, proc_id;
08       FILE *output;
09       CFG *procedure;
10
11       CheckTargetMachine();
12       AddProtoType("proc_trace", 1);
13       num_of_proc = numberOfCFG();
14       for (proc_id = 0; proc_id < num_of_proc; proc_id++) {
15            procedure = getCFG(proc_id);
16            printf ("Name of procedure %d is %s\n", proc_id, procedure->getName());
17            AddCallAtBeginOfFunc(procedure, "proc_trace", proc_id);
18       }
19       output = fopen("test_it.i.s", "w");
20       if (output){ produceCode(output); }
21 }
22 void Salto_init_hook(int c, char *v[]) {}
23 void Salto_end_hook(){}
24 int updateDelay(int delay, INST *, INST *, dependence) {return delay;}
```

Abbildung 5.3: Beispiel für die Nutzung von ASINST

In Abb. 5.3 ist der Quellcode für das Werkzeug `ptrace` dargestellt. Dieser Quellcode muß übersetzt, die entsprechenden SALTO-Bibliotheken und die Bibliothek ASINST hinzugebunden werden, um ein ausführbares Programm zu erhalten.

Wird `ptrace` gestartet, wertet SALTO zunächst die Argumente der Kommandozeile aus und generiert danach die interne Darstellung des Assembler-Quellcodes, dessen Dateiname als Argument - neben anderen Informationen - angegeben werden muß. Konnte die interne Darstellung erfolgreich aufgebaut werden, übergibt SALTO die Kontrolle des Programms an die Funktion `Salto_hook`.

Die Anweisung in Zeile 12 ruft die Funktion `AddProtoType` auf, um den Namen (`proc_trace`) und die Anzahl der Parameter (`1`) der Funktion anzumelden, zu der Funktionsaufrufe in den Assembler-Quellcode eingefügt werden sollen.

In Zeile 13 wird die Anzahl der von SALTO identifizierten Prozeduren ermittelt und in Zeile 15 ein Zeiger auf die `proc_id`-te Prozedur des Assembler-Quellcodes in der Variablen `procedure` gespeichert.

Durch Aufruf der Funktion `AddCallAtBeginOfFunc` in Zeile 17 wird nun ein Aufruf der Funktion `proc_trace` zu Beginn der Prozedur eingefügt, auf die der Zeiger in der Variablen `procedure` verweist. Als Argument wird der Funktion `proc_trace` der Inhalt der Variablen `proc_id` übergeben.

Am Ende der Funktion `Salto_hook` wird in Zeile 20 der modifizierte Assembler-Quellcode mit Hilfe der Funktion `produceCode` unter dem Dateinamen „`test_it.i.s`“ abgespeichert.

```
01 /* File : proc_trace.c */
02 #include <stdio.h>
03
04 void proc_trace(int i) { printf ("Procedure %d called.\n", i); }
```

Abbildung 5.4: Die Funktion `proc_trace`

Damit ein Programm mit dem Werkzeug `ptrace` instrumentiert werden kann, muß zunächst dessen Quellcode in Assembler-Quellcode transformiert werden. Nachdem der Assembler-Quellcode mit `ptrace` instrumentiert wurde, steht die instrumentierte Version unter dem Dateinamen „`test_it.i.s`“ zur Verfügung. Dieser Assembler-Quellcode muß nun assembliert und mit den Routinen zur Rettung und Wiederherstellung der Registerinhalte der benutzten Zielmaschine und der Funktion `proc_trace` gebunden werden, um ein ausführbares und instrumentiertes Programm zu erhalten.

```
01 /* File : test_it.c */
02 void bar() {}
03
04 void foo() { bar(); }
05
06 int main (){ foo(); bar(); return 0; }
```

Abbildung 5.5: Das Programm `test_it`

Eine sehr einfache Implementierung der Funktion `proc_trace` ist in Abb. 5.4 angegeben. Diese Funktion gibt lediglich aus, daß die `i`-te Funktion des instrumentierten Programms aufgerufen wurde.

In Abb. 5.5 ist ein kleines C-Programm angegeben, das nur aus Funktionsaufrufen besteht.

Um dieses Programm mit Hilfe des Werkzeugs `ptrace` zu instrumentieren, muß zunächst der Quellcode in Assembler-Quellcode transformiert werden:

```
> gcc -S -O0 test_it.c
```

Danach kann der Assembler-Quellcode mit dem Programm `ptrace` modifiziert werden. Dazu müssen neben dem Namen der Assembler-Quellcode-Datei die verwendete Zielmaschine (hier: Sparc) und einige weitere SALTO-Optionen angegeben werden:

```
> ptrace -m sparc -c -f filled_delay_slot -i test_it.s
```

Das Programm `ptrace` erzeugt die Ausgabe:

```
Salto rev. 1.3 (built with CC on sparc)
Copyright (C) 1997 Inria, France

Machine description file ok.
Name of procedure 0 is bar
Name of procedure 1 is foo
Name of procedure 2 is main
```

Nun kann der modifizierte Assembler-Quellcode assembliert und die nötigen Objektdateien hinzugebunden werden:

```
> gcc test_it.i.s stack-sparc.o proc_trace.o -o test_it.i
```

Nach dieser Prozedur kann das instrumentierte Programm gestartet werden. Während seiner Laufzeit gibt es die Reihenfolge der Funktionsaufrufe aus:

```
> test_it.i
Procedure 2 called.
Procedure 1 called.
Procedure 0 called.
Procedure 0 called.
```

5.1.4 Vergleich der Instrumentierungsansätze

In Kapitel 2 wurde bereits auf die Vor- und Nachteile der konventionellen Instrumentierungsansätze eingegangen. Insgesamt können diese als Eigenschaften aufge-

faßt werden, die die jeweiligen Ansätze besitzen. Tabelle 5.2 gibt Auskunft, ob der jeweilige Instrumentierungs-Ansatz die entsprechende Eigenschaft beinhaltet.

Eigenschaft	Quellcode	Assemblercode	Objektcode
Informationen lassen sich den Quellcodes leicht zuordnen	✔	teilweise	✘
Instrumentierer sind leicht zu portieren	✔	✔	✘
Instrumentierer sind für alle Programmiersprachen nutzbar	✘	✔	✔
Programme, die in mehreren Programmiersprachen verfaßt sind, können instrumentiert werden	✘	✔	✔
Programmteile, für die keine Quellcodes zur Verfügung stehen, können instrumentiert werden	✘	✘	✔
Nach Änderung der Instrumentierung, muß das Programm nicht neu übersetzt werden	✘	✘	✔

Tabelle 5.2: Vergleich der Instrumentierungsansätze

Es ist von Vorteil, wenn ein Ansatz möglichst viele dieser Eigenschaften in sich vereint, da entsprechende Instrumentierungswerkzeuge eine große Funktionalität erhalten und somit in vielen Fällen einsetzbar sind.

In der Tabelle ist zu erkennen, daß die Vorteile der Quellcode-Instrumentierung die Nachteile der Objektcode-Instrumentierung sind und umgekehrt, während die Assembler-Quellcode-Instrumentierung sowohl Eigenschaften der Objektcode- als auch der Quellcode-Instrumentierung besitzt. Dabei ähnelt die Assembler-Quellcode-Instrumentierung eher der Quellcode-Instrumentierung, da es sich bei Assembler um eine Programmiersprache handelt. Sie enthält allerdings einige positive Eigenschaften der Objektcode-Instrumentierung.

Das teilweise Fehlen der Eigenschaft, daß sich die gewonnenen Informationen leicht wieder den eigentlichen Quellcodes zuordnen lassen, könnte eventuell durch die Nutzung der häufig in den Kommentaren enthaltenen Originalanweisungen bei von Compilern erzeugten Assembler-Quellcodes hinzufügen lassen. So wäre es möglich, die gesammelten Informationen nicht nur den Funktionen des Programms zuzuordnen, sondern eine feinere Zuordnung vorzunehmen, so daß z. B. auch Schleifen innerhalb von Funktionen berücksichtigt werden könnten. Dies ist allerdings mit dem SALTO-System nicht möglich, da SALTO die in einem Assembler-Quellcode enthal-

tenen Kommentare nicht mit in die interne Darstellung aufnimmt und somit sind diese Kommentare auch nicht von auf SALTO aufbauenden Werkzeugen nutzbar.

5.1.5 Probleme der Assembler-Quellcode-Instrumentierung

Bei der Instrumentierung von Assembler-Quellcodes gibt es einige Probleme, die teilweise nur schwer oder ohne Unterstützung der Compiler-Hersteller nicht zu lösen sind.

Die Probleme lassen sich in zwei Gruppen einteilen; zum einen sind es Probleme, die durch die Nutzung des SALTO-Systems entstehen, zum anderen Probleme, die durch die zur Erzeugung des Assembler-Quellcodes verwendeten Compiler verursacht werden.

Im wesentlichen gibt es drei Probleme, die durch SALTO bedingt sind:

- SALTO ist zum Teil nicht in der Lage, den Assembler-Quellcode korrekt zu verarbeiten und kann deshalb die interne Darstellung nicht aufbauen. In solchen Fällen wird die Kontrolle nicht an das Benutzer-Werkzeug abgegeben und der Assembler-Quellcode kann folglich nicht instrumentiert werden.

- Werkzeuge, die auf SALTO basieren, können nur eine bestimmte Assemblersyntax verarbeiten, die durch die Maschinenbeschreibung und die jeweilig genutzte zielsystemspezifische Bibliothek von SALTO vorgegeben ist. Diese Syntax wird aber nur von einer Gruppe von Compilern genutzt, so daß angepaßte Maschinenbeschreibungen und zielsystemspezifische Bibliotheken genutzt werden müssen, um von anderen Compilern auf derselben Architektur erzeugte Assembler-Quellcodes zu verwenden. Also muß das Instrumentierungswerkzeug in verschiedenen Versionen für die jeweilig genutzten Compiler vorliegen. Dies ist allerdings weder benutzerfreundlich noch effizient, da jede Version Speicherplatz auf dem jeweiligen Hintergrundspeicher benötigt, der bei Systemen, die nicht über die Möglichkeit des dynamischen Bindens verfügen, wie z. B. das System CRAY T3E, beachtlich sein kann.

- Der mit Hilfe von SALTO manipulierte Assembler-Quellcode kann nicht immer von dem Compiler, der diesen erzeugt hat, übersetzt werden.

Die ersten beiden Probleme lassen sich durch eine geeignete Transformation des Assembler-Quellcodes, welche vor der Verwendung von SALTO durchgeführt werden muß, zum Teil ausschließen. Indem die verschiedenen Assembler-Quellcodes auf eine einheitliche Assemblersyntax abgebildet werden, läßt sich das zweite Problem völlig lösen, während sich das erste Problem durch eine solche Transformation minimieren läßt, wobei aufgrund der Komplexität des SALTO-Systems dieser Fall jedoch nicht völlig auszuschließen ist. Auch das dritte Problem läßt sich auf diese Weise lösen. Da SALTO die Assembler-Quellcodes nur geringfügig verändert, können diese wieder

in eine Form gebracht werden, die der jeweilige Compiler übersetzen kann. Diese Transformationen sind relativ leicht durchzuführen, da sich die auf einer Architektur verwendeten Assemblersprachen im Syntax nur unwesentlich unterscheiden.

Compiler	Sparc (SunOS)	Cray T3E
KCC	eingeschränkt	eingeschränkt
CC	eingeschränkt	eingeschränkt
g++	ok	nicht vorhanden
cc	ok	ok
gcc	ok	nicht vorhanden
f90	ok	eingeschränkt
f77	ok	ok

Tabelle 5.3: Nutzbarkeit der Compiler

Die Probleme, die durch die verwendeten Compiler verursacht werden, sind vor allem, daß einige Compiler nicht in der Lage sind, den von ihnen erzeugten Assembler-Quellcode zu übersetzen. Dies läßt sich insbesondere bei Compilern für komplexe moderne Programmiersprachen beobachten. In Tabelle 5.3 ist eine Auswahl von Compilern zusammengestellt, und deren Möglichkeit, auf der jeweiligen Architektur uneingeschränkt zur Erzeugung und somit Instrumentierung von Assembler-Quellcodes eingesetzt werden zu können.

Bis auf den GNU-Compiler g++ haben alle C++-Compiler Probleme, ihre erzeugten Assembler-Quellcodes zu übersetzen, sobald Templates [5] im Quellcode des Programms verwendet werden. Zumeist wird das Programm nicht korrekt gebunden, so daß ein ausführbares Programm nicht erzeugt werden kann.

Weitere Probleme während des Bindens treten mit dem C++-Compiler der Firma Cray Research Inc. auf, falls die `main` Funktion in der Programmiersprache C++ implementiert ist. In diesem Fall erkennt der Compiler nicht, daß es sich bei der `main` Funktion im entsprechenden Assembler-Quellcode um eine C++-Funktion handelt und meldet deshalb einen Fehler während des Bindens des Programms. Ebenfalls ist der Fortran90-Compiler der Firma Cray Research Inc. nicht in der Lage, seine selbst erzeugten Assembler-Quellcodes zu verarbeiten, falls Module im Quellcode verwendet werden.

Diese Probleme schränken die Nutzbarkeit dieses Instrumentierungsansatzes für einige Programmiersprachen soweit ein, daß er für die von den entsprechenden Compilern erzeugten Assembler-Quellcodes kaum noch sinnvoll einsetzbar ist.

5.2 Der Instrumentierer `vampir_asinst`

Exemplarisch wurde das SALTO-System und die Erweiterungsbibliothek ASINST genutzt, um ein Instrumentierungswerkzeug mit dem Namen `vampir_asinst` zu imple-

mentieren. Dabei werden Assembler-Quellcodes so instrumentiert, daß Aufrufe von Funktionen der VAMPIR-Tracebibliothek eingefügt und die Wrapper-Funktionen dieser Tracebibliothek genutzt werden können, um VAMPIR-Ereignisspuren zu erzeugen.

Durch die Verwendung des SALTO-Systems und der Bibliothek ASINST ist die Implementierung systemunabhängig, so daß der Quellcode des Instrumentierungswerkzeuges lediglich auf dem jeweiligen Rechnersystem übersetzt werden muß. Auf diesen Systemen muß SALTO installiert sein und die Bibliothek ASINST zur Verfügung stehen. Damit die instrumentierten Assembler-Quellcodes zu lauffähigen Programmen gebunden werden können, muß zusätzlich die VAMPIR-Tracebibliothek vollständig auf dem verwendeten Rechnersystem vorhanden sein.

Derzeit ist somit das Instrumentierungswerkzeug nur auf dem System CRAY T3E sinnvoll einsetzbar.

5.2.1 Benötigte Arbeitsumgebung

Um das Instrumentierungswerkzeug `vampir_asinst` zu nutzen, muß neben dem zu instrumentierenden Assembler-Quellcode zumindest eine Symboldatei mit Namen der Assembler-Quellcode-Datei mit dem Suffix „`.instrc`“ bereitgestellt werden. Diese Symboldatei dient zur Kontrolle der Instrumentierung, indem in ihr vermerkt wird, ob eine Benutzerfunktion instrumentiert werden soll.

Sollen auch die Wrapper-Funktionen der VAMPIR-Tracebibliothek genutzt werden, so muß zusätzlich eine weitere Symboldatei mit Namen der Assembler-Quellcode-Datei mit dem Suffix „`.system.instrc`“ generiert werden, in der die zu ersetzenden Funktionsnamen mit den entsprechenden Wrapper-Funktionsnamen anzugeben sind.

Beide Dateien müssen im gleichen Verzeichnis wie der zu instrumentierende Assembler-Quellcode abgelegt werden.

Spezifikation der „`.instrc`“ Symboldateien

Die Symboldateien mit der Endung „`.instrc`“ bestehen aus einer Zeile pro einer im Assembler-Quellcode vorhandenen Benutzerfunktion. Die Benutzerfunktionen müssen in der Reihenfolge, in der sie im Assembler-Quellcode definiert sind, in die Symboldatei aufgenommen werden. Außerdem muß für jede Benutzerfunktion ein Eintrag in der entsprechenden Symboldatei vorhanden sein. In den einzelnen Zeilen wird eine Beziehung zwischen dem Funktionsnamen, einer eindeutigen Funktionsnummer und dem Gruppennamen, zu der diese Funktion gehören soll, hergestellt. Eine Zeile dieses Symboldatei-Typs hat den Aufbau:

<Nummer> <Linkername> <Funktionsname> <Gruppenname> <Direktive> <Dateiname>

In der ersten Spalte steht eine Nummer, die der entsprechenden Benutzerfunktion eindeutig zugeordnet werden muß. Diese Nummer wird für die Funktionen

`vptenter` und `vptleave` der VAMPIR-Tracebibliothek als Argument benötigt und darf deshalb nur ein einziges Mal an eine Benutzerfunktion vergeben werden. Die zweite Spalte enthält den *Linkernamen* der Funktion, also den Namen, mit der die Funktion vor dem Binden des Programms bezeichnet ist. Zur Speicherung des eigentlichen Funktionsnamens dient die dritte Spalte. Die Gruppe, zu der die Funktion gehören soll, wird in der vierten Spalte vermerkt, während die fünfte zur Kontrolle der Instrumentierung genutzt wird. Soll die Benutzerfunktion durch das Instrumentierungswerkzeug `vampir_asinst` instrumentiert werden, so muß die Direktive „ON“, anderenfalls die Direktive „OFF“ angegeben werden. In der letzten Spalte ist der Name der Quellcode-Datei, aus der die Benutzerfunktion stammt, mit komplettem Pfad einzutragen. Die einzelnen Spalten müssen durch mindestens ein Leerzeichen getrennt sein.

Da der Aufbau einer solchen Symboldatei recht komplex ist, wurde ein Hilfsprogramm `getnames` entwickelt, welches ebenfalls auf SALTO basiert und für einen Assembler-Quellcode diese Symboldatei erzeugt. Diesem Programm muß neben einigen SALTO-spezifischen Optionen lediglich der Name der Assembler-Quellcode-Datei sowie der Name der eigentlichen Quellcode-Datei übergeben werden. So erzeugte Symboldateien enthalten sowohl in der zweiten als auch in der dritten Spalte den Linkernamen der Benutzerfunktion, da in Assembler-Quellcodes lediglich die Linkernamen der Funktionen zu finden sind. Alle Funktionen werden der Gruppe „Calculation“ zugeordnet. Bis auf eine eventuell vorhandene `main` Funktion wird für alle Funktionen die Direktive „OFF“ gesetzt.

Spezifikation der „. `system.instrc`“ Symboldateien

Diese Symboldatei dient zur Steuerung der Verwendung von Wrapper-Funktionen. Mit ihr ist es möglich, Funktionsaufrufe durch Wrapper-Funktionsaufrufe im gesamten Assembler-Quellcode durch das Instrumentierungswerkzeug `vampir_asinst` ersetzen zu lassen oder diese Ersetzung auf bestimmte Benutzerfunktionen innerhalb des Assembler-Quellcodes zu beschränken. Auch diese Symboldatei ist zeilenweise organisiert, wobei eine Zeile den folgenden Aufbau besitzt:

<Nummer> <Linkername> <Wrapper-Name> <Gruppenname> <Direktive> <Funktionsname>

In der ersten Spalte kann eine Nummer oder das Zeichen „*“ stehen. Eine Nummer gibt an, daß die Ersetzung der Funktionsaufrufe nur in der Benutzerfunktion erfolgen soll, die dieser Nummer in der entsprechenden „. `instrc`“ Symboldatei zugeordnet ist. Das Eintragen des Zeichens „*“ sorgt dafür, daß die Ersetzung auf den gesamten Assembler-Quellcode angewandt wird. Der Linkername der Wrapper-Funktion steht in der zweiten Spalte, während der eigentliche Name der Wrapper-Funktion in der dritten Spalte vermerkt werden muß. Die Gruppe, zu der die Wrapperfunktion gehören soll, wird in der vierten Spalte angegeben. Die fünfte Spalte dient der Steuerung der Instrumentierung. Wird hier die Direktive „REPLACE“ eingetragen, so wird die Ersetzung durchgeführt, anderenfalls muß

die Direktive „NOREPLACE“ verwendet werden. In der letzten Spalte steht der Funktionsname der Funktion, deren Aufrufe durch die Aufrufe der angegebenen Wrapper-Funktion ersetzt werden sollen. Die einzelnen Spalten müssen ebenfalls durch mindestens ein Leerzeichen voneinander getrennt werden.

5.2.2 Arbeitsweise des Instrumentierers

Anhand des in Abb. 5.6 angegebenen Message-Passing-Programms, das mit Hilfe von MPI implementiert wurde, soll im folgenden die Arbeitsweise des Instrumentierers erläutert werden.

Das Message-Passing-Programm besteht aus zwei Funktionen, wobei die Funktion `print` lediglich zur Ausgabe einer Zeichenkette dient. In der Funktion `main` wird zuerst, falls es sich um den Prozeß mit dem Rang 0 handelt, eine Nachricht, die den Namen des Prozesses beinhaltet, an alle anderen beteiligten Prozesse gesendet. Anderenfalls wird auf den Empfang der entsprechenden Nachricht gewartet. Danach wird, falls es sich nicht um den Prozeß mit dem Rang 0 handelt, eine Nachricht, die eine Zeichenkette enthält, an den Prozeß mit dem Rang 0 versendet, ansonsten auf diese Nachrichten gewartet und schließlich ausgegeben.

Um nun das Progamm mit Hilfe des Instrumentierungswerkzeuges `vampir_asinst` zu instrumentieren, muß neben der Assembler-Quellcode-Datei „`names.s`“ des Programms die Symboldatei „`names.instrc`“ erzeugt werden, die wie folgt aussehen könnte, falls sie mit dem Programm `getnames` generiert und als Direktive für die Funktionen `main` und `print` „ON“ angegeben wird:

```
1 print print Calculation ON .../names.c
2 main main Calculation ON .../names.c
```

Dabei soll „.../names.c“ andeuten, daß der Dateiname mit komplettem Pfad angegeben werden muß. Durch diese Symboldatei wird das Instrumentierungswerkzeug angewiesen, sowohl die Funktion `main` als auch die Funktion `print` zu instrumentieren.

Um die im Programm `names` vorhandenen Aufrufe der MPI-Routinen `MPI_Send` und `MPI_Recv` durch Aufrufe der entsprechenden Wrapper-Funktionen der VAMPIR-Tracebibliothek zu ersetzen, muß eine Symboldatei „`names.system.instrc`“ erzeugt werden, die folgende Gestalt besitzen könnte:

```
* MPI_Send_rep MPI_Send_rep MPI REPLACE MPI_Send
* MPI_Recv_rep MPI_Recv_rep MPI REPLACE MPI_Recv
```

Wird nun das Instrumentierungswerkzeug mit dem Assembler-Quellcode des Programms `names` gestartet, werden die Symboldateien analysiert und der Assembler-Quellcode entsprechend verändert. In diesem konkreten Fall wird am Anfang der Funktionen `main` und `print` ein Aufruf der Funktion `vptenter` und

```
01 #include <stdio.h>
02 #include <string.h>
03 #include "mpi.h"
04
05 int print (char *char_buffer) { printf ("%s\n", char_buffer);}
06
07 int main(int argc, char *argv[ ]) {
08   int pool_size, my_rank, destination, source, node_name_length;
09   char char_buffer[BUFSIZ], my_node_name[BUFSIZ], host_name[BUFSIZ];
10
11   MPI_Status status;
12   MPI_Init ( &argc, &argv );
13   MPI_Comm_size ( MPI_COMM_WORLD, &pool_size );
14   MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );
15   MPI_Get_processor_name ( my_node_name, &node_name_length );
16
17   if ( my_rank == 0 ) {
18     for (destination = 1; destination < pool_size; destination++)
19       MPI_Send(my_node_name, strlen(my_node_name) + 1, MPI_CHAR,
20                destination, 1001, MPI_COMM_WORLD);
21   }
22   else {
23     MPI_Recv(host_name, BUFSIZ, MPI_CHAR, 0, 1001,
24              MPI_COMM_WORLD, &status);
25   }
26   if ( my_rank ≠ 0 ) {
27   sprintf (char_buffer, "Greetings to %s from (%s, %d)",
28           host_name, my_node_name, my_rank);
29   MPI_Send (char_buffer, strlen(char_buffer) + 1, MPI_CHAR, 0,
30             2002, MPI_COMM_WORLD);
31   }
32   else {
33     for (source = 1; source < pool_size; source++) {
34       MPI_Recv(char_buffer, BUFSIZ, MPI_CHAR, source, 2002,
35                MPI_COMM_WORLD, &status);
36       print (char_buffer);
37     }
38   }
39   MPI_Finalize ();
40 }
```

Abbildung 5.6: Das Programm `names`

am Ende dieser Funktionen ein Aufruf der Funktion `vptleave` der VAMPIR-Tracebibliothek eingefügt. Außerdem werden Aufrufe zu den Funktionen `vptsetup` und `vptflush` am Anfang bzw. am Ende der `main` Funktion integriert, um die Aufzeichnung der Ereignisspur zu starten und wieder zu beenden. Zusätzlich werden alle Aufrufe der MPI-Routinen `MPI_Send` und `MPI_Recv` durch Aufrufe der

entsprechenden Wrapper-Funktionen substituiert.

In Abb. 5.7 ist ein Ausschnitt des Assembler-Quellcodes des Programms `names` sowohl vor als auch nach der Instrumentierung durch das Programm `vampir_asinst` zu sehen, wobei alle Änderungen und eingefügten Instruktionen hervorgehoben sind. Der Assembler-Quellcode wurde auf dem System CRAY T3E mit Hilfe des C-Compilers der Firma Cray Research Inc. erzeugt.

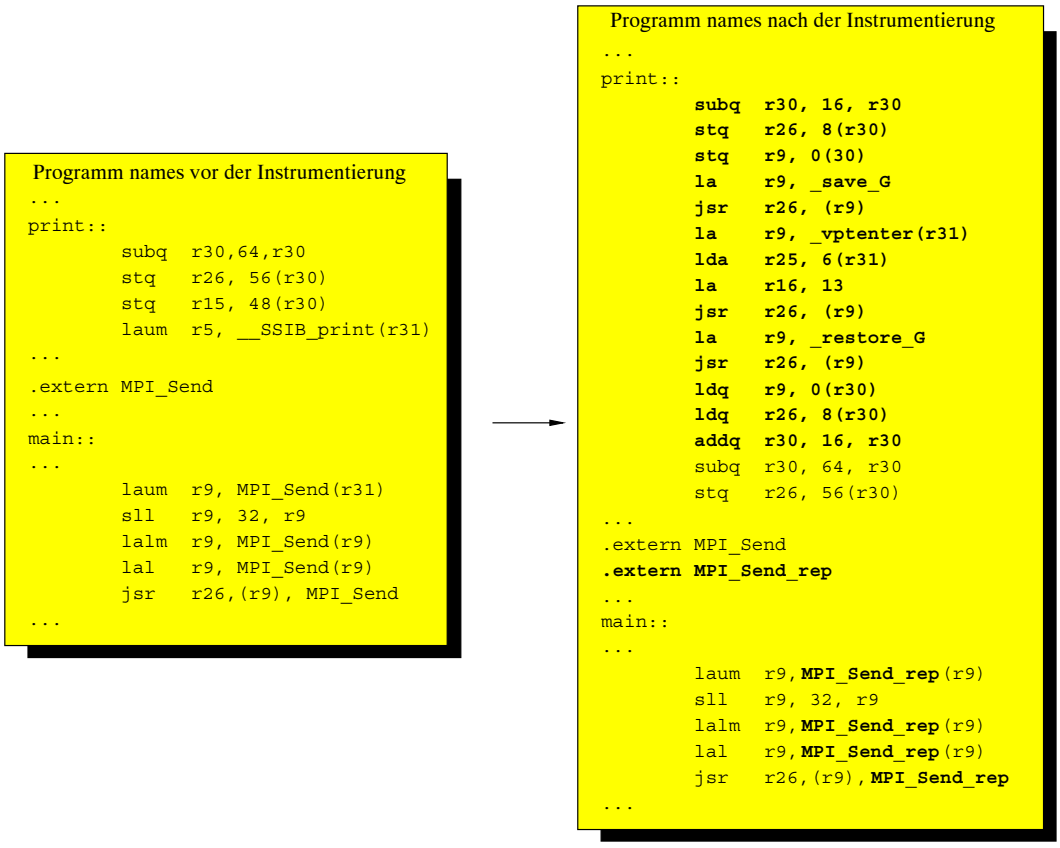


Abbildung 5.7: Manipulation des Assembler-Quellcodes

An der Prozedur `print` ist zu sehen, daß das Einfügen von Funktionsaufrufen sehr aufwendig ist. Neben der eigentlich aufzurufenden Funktion werden zwei weitere Funktionsaufrufe eingefügt. Die Funktion `_save_G` dient zur Rettung der Inhalte der globalen Register, während die Funktion `_restore_G` diese wiederherstellt. Diese Funktionen müssen vor bzw. nach dem eigentlich einzufügenden Funktionsaufruf aufgerufen werden, um ein korrektes Laufzeitverhalten des instrumentierten Programms zu gewährleisten.

Die Manipulation von bereits im Assembler-Quellcode vorhandenen Funktionsaufrufen stellt im Gegensatz zur Integration neuer Funktionsaufrufe einen geringen Aufwand dar. Um einen Funktionsaufruf durch einen anderen zu ersetzen, müssen lediglich die Funktionsnamen innerhalb der Assembler-Instruktionen substituiert werden. Da alle extern genutzten Funktionen in der Assemblerspache der Firma

Cray Research Inc. [9] durch die Pseudo-Instruktion „.extern“ bekanntgegeben werden müssen, werden diese Instruktionen für neue Funktionsnamen generiert und an geeigneter Stelle in den Assembler-Quellcode aufgenommen.

5.2.3 Performance Vergleich

Die durch die Instrumentierung hervorgerufenen Änderungen an dem Assembler-Quellcode wirken sich natürlich auf die Ausführungszeit des Programms aus. Um diese Auswirkungen zu ermitteln, wurde die Zeit gemessen, die verbraucht wird, um eine Funktion, die keine Anweisungen enthält, 10000 mal aufzurufen. Diese Funktion wurde dazu einmal nicht instrumentiert und einmal mit dem Programm `vampir_asinst` instrumentiert, jedoch ohne die Trace-Funktionen der VAMPIR-Tracebibliothek aufzurufen, so daß nur die Funktionen zur Rettung und Wiederherstellung der globalen Register Einfluß auf die Messung hatten.

Des weiteren wurde diese Funktion per Hand instrumentiert, also die Funktionsaufrufe der Trace-Funktionen in den Quellcode der Funktion eingefügt, sowie einmal mit dem Programm `vampir_asinst` instrumentiert.

Art der Instrumentierung	durchschnittliche Ausführungszeit	Overhead
nicht instrumentiert	0,16 ms	——
per `vampir_asinst`, ohne Trace-Funktions-aufrufe, instrumentiert	1,19 ms	+1,02 ms
per Hand instrumentiert	38,16 ms	——
per `vampir_asinst` instrumentiert	39,49 ms	+1,32 ms

Tabelle 5.4: Benötigte Zeit für 10000 Funktionsausführungen

Die Messung wurde jeweils 100 mal durchgeführt, und die durchschnittliche Ausführungszeit für die vier Versionen berechnet (Tabelle 5.4). Zusätzlich wurden die Differenzen der Ausführungszeiten zwischen der nicht instrumentierten Version und der mit `vampir_asinst` manipulierten Version, in der die Trace-Funktionen nicht aufgerufen werden, sowie zwischen der per Hand und der mit `vampir_asinst` instrumentierten Version ermittelt. Diese Differenzen geben Auskunft über den durch `vampir_asinst` erzeugten Overhead.

Zu beachten ist, daß sich diese Meßergebnisse auf die Instrumentierung einer Funktion beziehen, die keine Anweisungen enthält. Dies stellt den ungünstigsten Fall dar, womit sich die hohen Differenzen in den Ausführungszeiten der instrumentierten Versionen im Bezug auf die nicht instrumentierte Version erklären lassen.

Daß ein Programm, das mit Hilfe der VAMPIR-Tracebibliothek instrumentiert wurde, langsamer ist als die nicht instrumentierte Version, ist aber keine Seltenheit. Deshalb ist bei der Instrumentierung stets darauf zu achten nur die Programmteile zu instrumentieren, die untersucht werden sollen, um die Auswirkungen auf die Ausführungszeit des Programms möglichst gering zu halten.

Die größere Differenz in den Ausführungszeiten zwischen der mit `vampir_asinst` und der per Hand instrumentierten Versionen kommt dadurch zu stande, daß bei der Instrumentierung mit Hilfe der Bibliothek ASINST alle globalen Register gerettet und wiederhergestellt werden, während ein Compiler vor einem Funktionsaufruf nur die Register rettet bzw. nach einem Funktionsaufruf wiederherstellt, die bis zu diesem Zeitpunkt im Programm genutzt werden, wobei diese dem Compiler, da er das Programm in Maschinensprache übersetzt, bekannt sind.

Ferner werden Funktionsaufrufe durch Compiler für Systeme, die auf dem DEC Alpha Prozessor [6, 7] aufgebaut sind in der Regel optimiert, indem die dazu notwendigen Instruktionen durch eine Abfolge äquivalenter Instruktionen ersetzt werden, die dann vom gleichem Typ sind und somit von dem Prozessor häufig schneller abgearbeitet werden.

Daß die Rettung bzw. Wiederherstellung aller globalen Register, insbesondere im Bezug auf den durch die Verwendung der Funktionen der VAMPIR-Tracebibliothek verursachten Overhead, nicht besonders ins Gewicht fallen, läßt sich auf die verwendeten Prozessoren zurückführen.

Obwohl der DEC Alpha Prozessor, der im System CRAY T3E genutzt wird, über 32 Integer- und 32 Fließkomma-Register verfügt, die alle auf dem Stack gerettet und wiederhergestellt werden müssen, sind die dazu nötigen Instruktionen vom gleichen Typ, so daß die beiden Instruktionen-Pipelines des Alpha Prozessor gut ausgelastet sind und somit die Instruktionen sehr schnell ausgeführt werden können.

Bei der Familie der Sparc Prozessoren wird eine Technik namens *Registerfenster* [34] verwendet, wobei acht Fenster mit jeweils 32 Registern sowie acht globale Register zur Verfügung stehen. Wird nun eine Funktion aufgerufen, müssen lediglich die globalen Register gerettet und ein neues Registerfenster verwendet werden. Nach Beendigung der Funktion werden die globalen Register wiederhergestellt und das Registerfenster wieder freigegeben. Erst wenn keine freien Registerfenster mehr zur Verfügung stehen, wird ein Fenster gerettet, um dem Programm das benötigte Fenster bereitstellen zu können. Deshalb werden nur wenige Assembler-Instruktionen benötigt, um einen Funktionsaufruf in Assembler-Quellcodes für auf Sparc Prozessoren basierende Rechnersysteme einzufügen, womit ebenfalls die Auswirkungen auf die Ausführungszeiten der entsprechenden Programme gering ist.

5.3 System zur Instrumentierung von Programmen

Die Instrumentierung eines Programms mittels `vampir_asinst` ist ein sehr aufwendiger Prozeß. Zunächst müssen die Assembler-Quellcodes des Programms erzeugt, zu jedem Assembler-Quellcode die entsprechenden Symboldateien generiert und eventuell die Assembler-Quellcodes manipuliert werden, um diese dann jeweils mit `vampir_asinst` zu instrumentieren.

Nach der Instrumentierung der Assembler-Quellcodes müssen diese eventuell nochmals transformiert werden, um sie mit den entsprechenden Compilern übersetzen zu können. Zum Abschluß dieses Prozesses müssen die erzeugten Objektcodes noch mit der VAMPIR-Tracebibliothek und den Funktionen zur Rettung und Wiederherstellung der globalen Register gebunden werden. Erst dann steht ein ausführbares und instrumentiertes Programm zur Verfügung.

Damit nun allerdings VAMPIR-Ereignisspuren erzeugt werden können, muß noch die von den Funktionen der VAMPIR-Tracebibliothek benötigte Symboldatei „`instrc`“ aus den von `vampir_asinst` genutzten Symboldateien gewonnen werden.

Um diesen Prozeß nicht dem Benutzer aufzubürden, sondern weitgehend automatisiert durchzuführen, wurde ein Perl-Skript entwickelt, daß alle notwendigen Schritte übernimmt und somit die Generierung von instrumentierten Programmen benutzerfreundlich realisiert.

Im folgenden wird dieses System zur automatisierten Instrumentierung von Programmen vorgestellt, dabei seine Implementierung und Arbeitsweise beschrieben sowie seine Benutzung erläutert.

5.3.1 Systemarchitektur und Implementierung

Die zugrundeliegende Idee des Instrumentierungssystems ist es, diesem System den kompletten Compiler-Aufruf zu übergeben, so daß es einerseits leicht von der Kommandozeile zu starten ist und andererseits leicht in Makefiles [32] verwendet werden kann. Das System ist mittels eines Perl-Skriptes namens `inst` implementiert.

Die Skript-Sprache Perl wurde zur Implementierung gewählt, da sie auf allen modernen Rechnersystemen einsetzbar ist und sich gut für Kontrollaufgaben eignet. Außerdem existiert mit dem Modul Tk [22] eine Möglichkeit, leicht eine graphische Benutzerschnittstelle zu realisieren, und durch die gute Unterstützung von regluären Ausdrücken lassen sich Vergleiche und Modifikationen von Textdateien einfach durchführen.

Neben dem eigentlichen Perl-Skript `inst` wurden zwei weitere Perl-Skripte `pre-ascon` und `post-ascon` implementiert, die Assembler-Quellcodes so konvertieren, daß sie vom SALTO-System erkannt bzw. nach der Manipulation mit Hilfe von

SALTO von dem jeweilig verwendeten Compiler übersetzt werden können. Diese beiden Skripte müssen, soweit sie notwendig sind, für jedes verwendete Rechnersystem zur Verfügung stehen.

Wird das System gestartet, müssen alle Compiler-Aufrufe, die zur Erzeugung eines ausführbaren Programms erforderlich sind, nacheinander an das Perl-Skript `inst` übergeben werden.

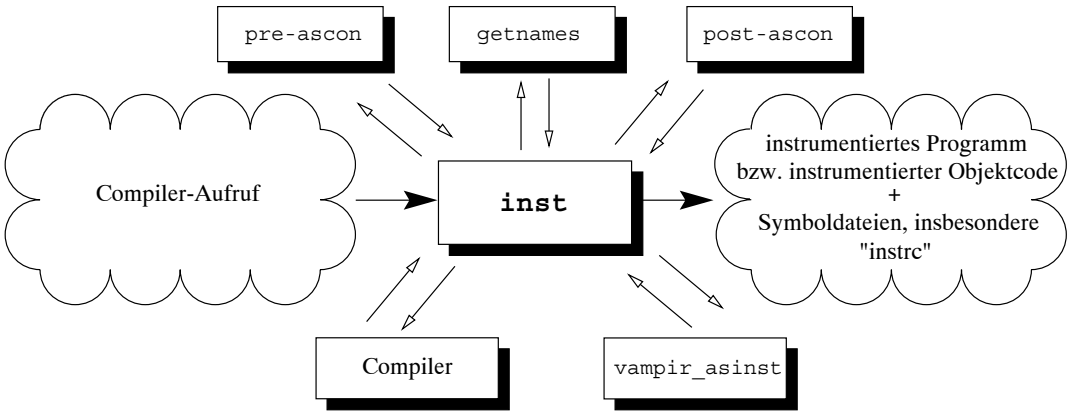


Abbildung 5.8: System zur Instrumentierung von Programmen

Das Perl-Skript `inst` übernimmt dann für die einzelnen Übersetzungsprozesse alle notwendigen Aufrufe des Compilers, der Konvertierungsskripte `pre-ascon` und `post-ascon`, des Hilfsprogramms `getnames` und des Instrumentierers `vampir_asinst`, um ein instrumentiertes Programm bzw. einen instrumentierten Objektcode zu erzeugen (Abb. 5.8). Des weiteren werden alle benötigten Symboldateien, insbesondere die Symboldatei „`instrc`“, die zur Nutzung der Funktionen der VAMPIR-Tracebibliothek vorausgesetzt wird, erzeugt bzw. aktualisiert. Zur Generierung von VAMPIR-Ereignisspuren können nun die instrumentierten Programme sofort gestartet werden.

Das System sorgt zusätzlich dafür, daß nur die wirklich notwendigen Schritte durchgeführt werden, um die Bearbeitungszeiten möglichst gering zu halten.

In Abb. 5.9 ist ein stark vereinfachtes Ablaufdiagramm des Perl-Skriptes `inst` dargestellt. Nach Aufruf des Systems werden zunächst die Argumente ausgewertet, die über die Kommandozeile an das Skript `inst` übergeben werden. Als letztes Argument muß dem Skript der komplette Compiler-Aufruf zur Verfügung gestellt werden. Dieser Aufruf wird analysiert, dabei der verwendete Compiler gespeichert, die Argumente des Compiler-Aufrufs in Compiler-Optionen, zu übersetzende Quellcode-Dateien und sonstige Dateien unterteilt.

Im nächsten Schritt wird überprüft, ob die einzelnen Quellcodes in Assembler-Quellcodes überführt werden müssen. Dies ist nur notwendig, falls zu den entsprechenden Quellcode-Dateien noch keine Assembler-Quellcode-Dateien existieren oder Änderungen an den entsprechenden Quellcodes vorgenommen wurden. Mußte

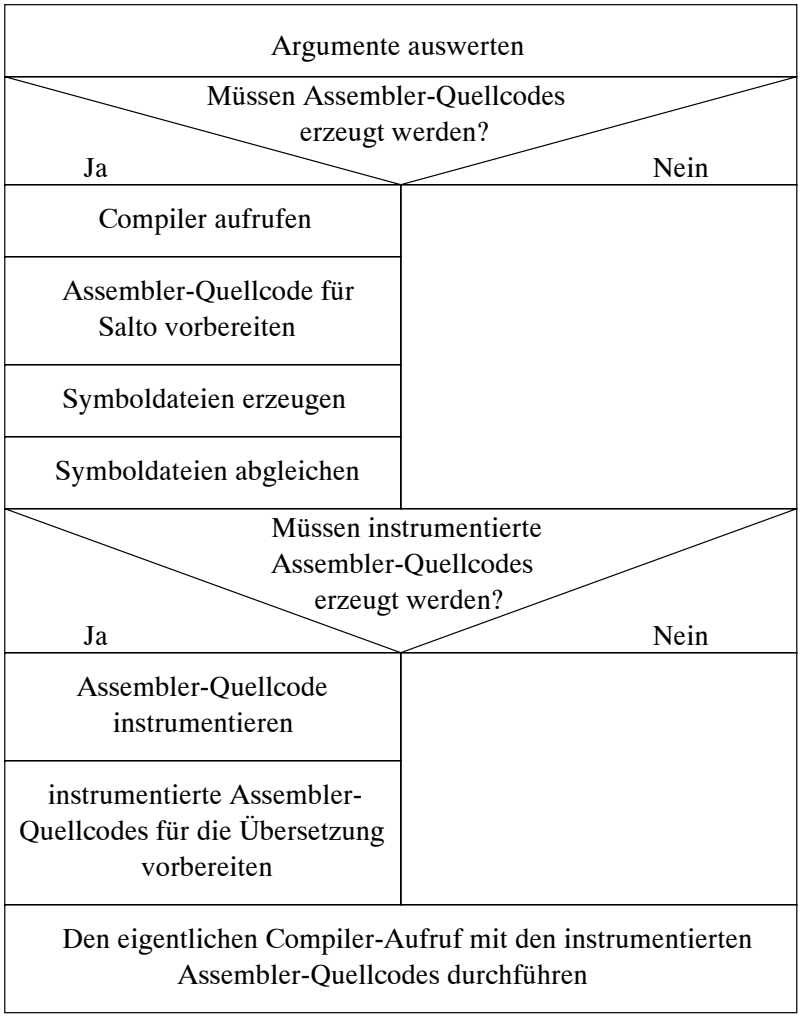


Abbildung 5.9: Ablaufdiagramm des Skriptes `inst`

ein Assembler-Quellcode generiert werden, so wird er mittels `pre-ascon` konvertiert und unter anderem werden mit Hilfe des Programms `getnames` die benötigten Symboldateien erzeugt, die dann noch mit einer globalen Symboldatei namens „`global.instrc`“ abgeglichen werden, um den Funktionen innerhalb des zu instrumentierenden Programms eine eindeutige Nummer zuordnen zu können.

Falls C++- oder Fortran90-Quellcodes zum Einsatz kommen, werden die Funktionsnamen durch die Compiler zum Teil in sehr schwer zu interpretierende Linkernamen transformiert. Deshalb werden diese Linkernamen, die in Assembler-Quellcodes genutzt werden, durch entsprechende Programme wieder in eine für den Benutzer lesbare Form transformiert und in die jeweiligen Symboldateien in der dafür vorgesehenen Spalte eingetragen.

Als Standardeinstellung werden alle Funktionen des Programms der Gruppe „Calculation“ zugeordnet und die Direktive „OFF“ verwendet mit Ausnahme der Funktion `main`, deren Direktive auf „ON“ gesetzt und diese Funktion somit instrumentiert

wird. Für die Verwendung der Wrapper-Funktionen gilt als Standardeinstellung, daß die Aufrufe der acht Funktionen `MPI_Init`, `MPI_Finalize`, `MPI_Send`, `MPI_Recv`, `MPI_Bcast`, `MPI_Gather`, `MPI_Scatter` und `MPI_Barrier` im gesamten Programm ersetzt werden.

Danach wird überprüft, ob Assembler-Quellcodes mittels `vampir_asinst` instrumentiert werden müssen. Dies ist der Fall, falls die entsprechenden instrumentierten Assembler-Quellcode-Dateien nicht existieren oder Änderungen an der Instrumentierung vorgenommen werden müssen, was durch Veränderungen an den entsprechenden Symboldateien erkannt wird. Nach der Instrumentierung werden die Assembler-Quellcodes mittels `post-ascon` für den Compiler aufbereitet.

Als letztes wird der eigentliche Compiler-Aufruf durchgeführt, in dem jetzt nicht mehr die Quellcode-Dateien, sondern die instrumentierten Assembler-Quellcode-Dateien verwendet werden.

5.3.2 Arbeitsweise des Systems

In Abb. 5.10 ist eine Makefile-Datei zu sehen, die so modifiziert ist, daß diese zur Erzeugung einer instrumentierten Version des Programms verwendet werden kann. Dazu ist den Namen der genutzten Compiler, die oft zu Beginn eines Makefiles in Umgebungsvariablen gespeichert werden, das Perl-Skript `inst` vorangestellt, um zu gewährleisten, daß das Perl-Skript mit den jeweiligen Compiler-Aufrufen an Stelle des Compiler gestartet wird.

```
#CC = cc
CC = inst cc
#CXX = CC
CXX = inst CC
#F90 = f90
F90 = inst f90

prog: main.c bcast.o param.o
    $(CC) -o prog main.c bcast.o param.o

bcast.o: bcast.cc
    $(CXX) -c param.cc

param.o: param.f90
    $(f90) -c param.f90
```

Abbildung 5.10: Manipulierte Makefile-Datei

Durch Verwendung des Befehls `make` wird das instrumentierte Programm erzeugt, ohne daß der Benutzer in den Prozeß eingreifen muß.

Zur Darstellung der Arbeitsweise sind in Abb. 5.11 die von dem Perl-Skript aufgerufenen Programme mit den erforderlichen Argumenten für den Compiler-Aufruf „`$(CC) -o prog main.c bcast.o param.o`“ der Makefile-Datei zu sehen, sowie die von diesen Programmen erzeugten Dateien. Zusätzlich sind alle vom Perl-Skript `inst` generierten Dateien aufgeführt.

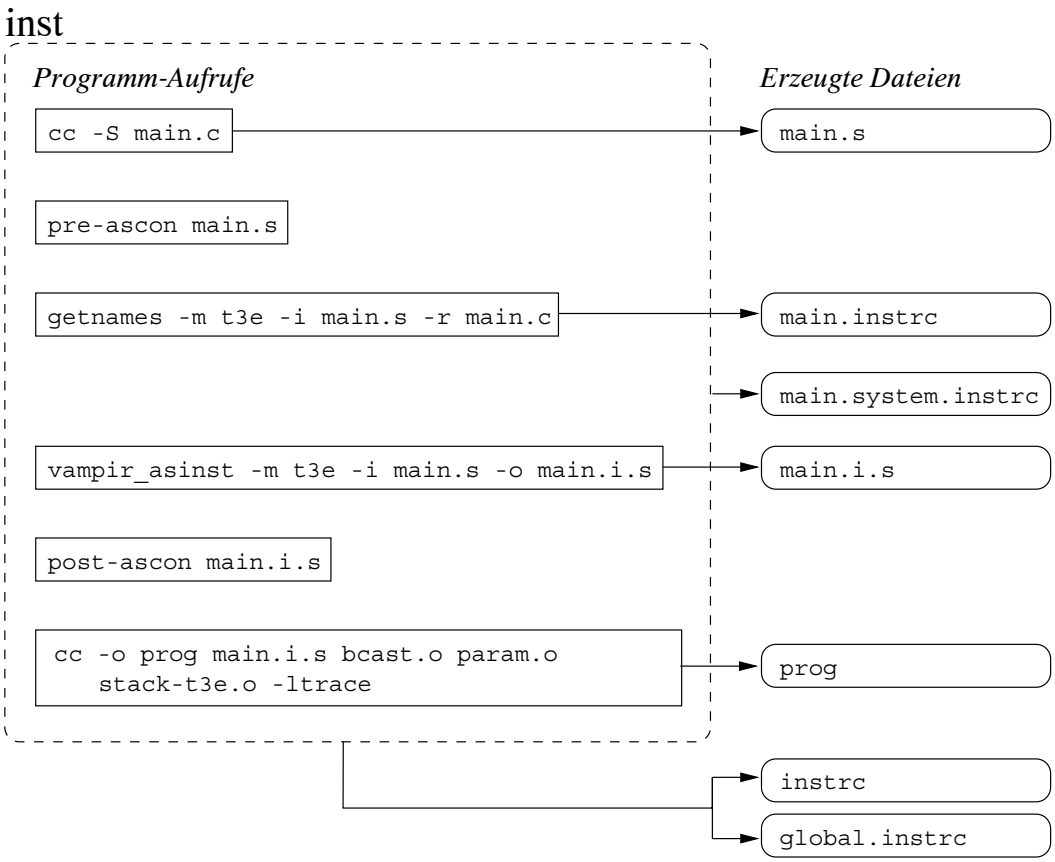


Abbildung 5.11: Arbeitsweise des Skriptes `inst`

Das Perl-Skript ruft also alle benötigten Programme in der richtigen Reihenfolge auf und sorgt dafür, daß die jeweilig erzeugten Dateien durch die entsprechenden Programme weiterverarbeitet werden. Wird durch den abschließenden Compiler-Aufruf des Perl-Skriptes `inst` ein ausführbares Programm generiert, so wird dieser Aufruf von `inst` so verändert, daß die VAMPIR-Tracebibliothek sowie die beiden Funktionen zur Rettung und Wiederherstellung der globalen Register hinzugebunden werden.

Falls sich in den verwendeten Quellcodes Syntaxfehler befinden sollten, so wird das Perl-Skript `inst` mit einer Fehlermeldung abgebrochen. Nachdem die Syntaxfehler behoben sind, kann das System erneut gestartet werden, ohne das dazu bereits erstellte Dateien verändert werden müssen.

Wurde das Skript `inst` erfolgreich beendet, so kann das instrumentierte Programm gestartet werden. Nach Ausführung des Programms befinden sich pro Prozessor bzw. Prozeß, die für den Programmlauf genutzt wurden, eine Datei „`.trace.`*n*“ im

aktuellen Verzeichnis. Dabei steht n für die Nummer des jeweiligen Prozessors bzw. Prozesses. Diese Dateien müssen mittels des Hilfsprogramm `vtpmerge` [26] zu einer Ergeinisspur zusammengefaßt werden, um diese beispielsweise mit dem Werkzeug VAMPIR anlaysieren zu können.

5.3.3 Benutzung des Systems

Neben dem Compiler-Aufruf können weitere Argumente an das Perl-Skript `inst` übergeben werden, die zum einen die Arbeitsweise des Systems und zum anderen die Instrumentierung der Assembler-Quellcodes beeinflussen.

Mit der Option `-f` *`Name`* kann eine globale Symboldatei angegeben werden, die dann vom Instrumentierungssystem zum Abgleichen der Symboldateien verwendet wird. Einerseits kann diese Option genutzt werden, um der globalen Symboldatei einen anderen Namen als „`global.instrc`“ zu geben, falls eine Quellcode-Datei mit Namen „`global.*`“ existiert, da für eine solche Quellcode-Datei durch das Instrumentierungssystem eine Symboldatei namens „`global.instrc`“ angelegt werden und somit das System nicht mehr korrekt arbeiten würde. Andererseits ist es mit dieser Option möglich, Programme zu instrumentieren, deren Quellcode-Dateien auf mehrere Verzeichnisse verteilt sind und durch mehr als einen Makefile erzeugt werden. In solchen Fällen muß stets dieselbe globale Symboldatei verwendet werden, um zu gewährleisten, daß den Funktionen in den verschiedenen Programmteilen, die durch die jeweiligen Makefiles generiert werden, eindeutige Funktionsnummern zugeordnet werden können.

Ein Instrumentierungs-Kontrollskript kann mit Hilfe der Option `-s` *`Name`* an das Perl-Skript `inst` übergeben werden, durch das der Benutzer die Instrumentierung steuern kann. Der Aufbau und die Wirkungsweise der Kontrollskripte werden in Abschnitt 5.4 vorgestellt.

Die Option `-p` *`Anweisung`* dient zu Kontrolle der graphischen Benutzeroberfläche, deren Funktionalität im Abschnitt 5.5 beschrieben wird. Die Standardeinstellung `no` bewirkt, daß die Oberfläche nicht in Erscheinung tritt, während mit der Anweisung `always` die Oberfläche vor jedem Aufruf des Instrumentierers `vampir_asinst` eingeblendet wird, um dem Benutzer die Möglichkeit zu bieten die Instrumentierung der an dem Compiler-Aufruf beteiligten Assembler-Quellcodes zu beinflussen. Mit der Anweisung `yes` wird die Oberfläche lediglich angezeigt, falls sich Änderungen an den entsprechenden Symboldateien ergeben haben. Dieses kann eintreten, falls der entsprechende Quellcode manipuliert wurde oder aber ein Instrumentierungs-Kontrollskript die Symboldateien verändert hat. Wird die Anweisung `yes` oder `always` genutzt, so wird zu Beginn des Instrumentierungsprozesses ein Fenster angezeigt, in dem die Wrapper-Funktionen ausgewählt werden können, die im gesamten Programm verwendet werden sollen.

Durch die Option `-L` *`Tracelimit`* wird die Anzahl der von der VAMPIR-Tracebibliothek maximal zu erzeugenden Ereignis-Records festgelegt, so daß die

Größe der zu erstellenden Ereignisspuren begrenzt werden kann. Die Standardeinstellung ist `10000`. Mit der Option `-F` *`Format`* kann das Format `ascii` oder `binary` gewählt werden, in dem die Ereignisspur gespeichert wird. Als Standardeinstellung wird das Format `binary` gewählt.

Bei der Verwendung von Makefiles zur Erzeugung von Programmen ist zu beachten, daß im allgemeinen die notwendigen Compiler-Aufrufe nur durchgeführt werden, falls die zu erzeugenden Objektcode-Dateien bzw. das ausführbare Programm noch nicht existieren oder die entsprechenden Quellcode-Dateien verändert wurden.

Wird das Instrumentierungswerkzeug eingesetzt, ist dafür zu sorgen, daß durch die Makefile-Datei die erforderlichen Aufrufe ausgeführt werden. Also müssen das ausführbare Programm und die zum Programm gehörenden Objektcode-Dateien gelöscht werden, um eine instrumentierte Version des Programms zu generieren.

Dies stellt insbesondere ein Problem dar, falls Teile des Programms neu instrumentiert werden sollen. Auch in diesem Fall muß das instrumentierte, ausführbare Programm, sowie die Objektcode-Dateien gelöscht werden, an deren Assembler-Quellcodes durch die neue Instrumentierung Änderungen vorgenommen werden müssen. Die Folge ist, daß der Benutzer wissen muß, welche Objektcode-Dateien für die neue Instrumentierung zu löschen sind, und daß das gesamte Programm neu übersetzt werden muß, falls die Änderung der Instrumentierung globalen Charakter besitzt, wie z. B. das Ersetzen weiterer Funktionsaufrufe im gesamten Programm durch Aufrufe der entsprechenden Wrapper-Funktionen.

5.4 Instrumentierungs-Kontrollskripte

Zum Eingriff in die Instrumentierung durch das Instrumentierungssystem, kann dem Perl-Skript `inst` ein Instrumentierungs-Kontrollskript übergeben werden. Anhand der im Kontrollskript enthaltenen Anweisungen kann bestimmt werden, welche Funktionen instrumentiert und welche Funktionsaufrufe durch Aufrufe von Wrapper-Funktionen substituiert werden sollen.

In diesem Abschnitt werden die Anweisungen der Instrumentierungs-Kontrollskriptsprache und ihre Syntax vorgestellt. Des weiteren wird die Abarbeitung der Kontrollskripte und die damit verbundenen Auswirkungen auf die Symboldateien beschrieben. Abschließend wird ein Beispiel für ein Instrumentierungs-Kontrollskript behandelt und auf die aus diesem Kontrollskript resultierende Instrumentierung eingegangen.

5.4.1 Syntax der Anweisungen

Die Instrumentierungs-Kontrollskriptsprache besteht aus einigen reservierten Schlüsselworten, Anweisungen, Zeichenketten und regulären Ausdrücken. Dabei wer-

den im folgenden Schlüsselworte und Anweisungen in Schreibmaschinenschrift gehalten, Auswahlmöglichkeiten durch einen Schrägstrich voneinander getrennt und von eckigen Klammern umgeben, und variable Argumente kursiv gesetzt. Bei diesen Argumenten handelt es sich ausschließlich um reguläre Ausdrücke und um Zeichenketten. Damit diese leicht voneinander zu unterscheiden sind werden reguläre Ausdrücke zusätzlich in Schreibmaschinenschrift geschrieben. In Anhang D sind die Syntaxdiagramme der Instrumentierungs-Kontrollskriptsprache abgebildet.

`FILE` *filename*

Durch diese Anweisung wird ein Block festgelegt, in dem sowohl `TRACE`- als auch `REPLACE`-Anweisungen in beliebiger Reihenfolge und Häufigkeit enthalten sein dürfen. Diese `TRACE`- und `REPLACE`-Anweisungen wirken sich nur auf die Symboldateien der Quellcode-Dateien aus, deren Dateinamen zu dem Muster passen, das durch den regulären Ausdruck *filename* beschrieben ist. Der Block muß mittels des reservierten Schlüsselworts `END` abgeschlossen werden. Ein Instrumentierungs-Kontrollskript darf aus beliebig vielen `FILE`-Blöcken bestehen.

`TRACE [ON/OFF]`

Diese Anweisung muß innerhalb eines `FILE`-Blocks stehen und legt ebenfalls einen Block an. In diesem Block können nun Funktionen spezifiziert werden, die instrumentiert (`ON`) bzw. nicht instrumentiert (`OFF`) werden sollen. Die Funktionen sind dabei wie folgt anzugeben:

functionname group

Der reguläre Ausdruck *functionname* legt ein Muster fest, so daß alle Funktionen, deren Funktionsnamen zu diesem Muster passen, von der `TRACE`-Anweisung betroffen sind. Mit der Zeichenkette *group* wird festgelegt zu welcher Gruppe die Funktionen gehören sollen. Auf diese Weise dürfen beliebig viele Funktionen innerhalb eines `TRACE`-Blocks beschrieben werden. Den Abschluß eines `TRACE`-Blocks bildet ebenfalls das Schlüsselwort `END`.

`REPLACE [ON/OFF]`

Mit dieser Anweisung kann die Substitution von Funktionsaufrufen durch Aufrufe von Wrapper-Funktionen innerhalb von Benutzerfunktionen beeinflußt werden. Durch `REPLACE ON` werden die Ersetzungen durchgeführt, während `REPLACE OFF` diese unterdrückt. Durch die `REPLACE`-Anweisung wird ebenfalls ein Block festgelegt, in dem Funktionen aufgeführt werden können, in denen Funktionsaufrufe ersetzt bzw. nicht ersetzt werden sollen. Dabei werden diese Funktionen, die

Funktionsaufrufe und die Wrapper-Funktionen wie folgt angegeben:

functionname functioncall wrapperfunction group

Mit dem regulären Ausdruck *functionname* wird, wie bei der `TRACE`-Anweisung, ein Muster beschrieben, so daß sich die `REPLACE`-Anweisung auf alle Funktionen auswirkt, deren Funktionsnamen zu diesem Muster passen. Mit der Zeichenkette *functioncall* wird der Name der Funktion festgelegt, deren Aufrufe duch Aufrufe der Wrapper-Funktion, die mit der Zeichenkette *wrapperfunction* angegeben wird, substituiert werden sollen. Mit der Zeichenkette *group* wird die Gruppe bestimmt, in der die Wrapper-Funktion eingeordnet wird. Auf diese Weise können beliebig viele Funktionen innerhalb des Blocks selektiert werden. Auch die `REPLACE`-Blöcke werden mittels des Schlüsselworts `END` abgeschlossen.

Kommentare können durch das Zeichen „`#`“ in die Instrumentierungs-Kontrollskripte aufgenommen werden. Dabei wird die Zeile des Kontrollskriptes ab diesem Zeichen vom Instrumentierungssystem ignoriert und hat somit keine Auswirkungen auf die Instrumentierung.

5.4.2 Abarbeitung der Kontrollskripte

Die Instrumentierungs-Kontrollskripte werden streng von oben nach unten zeilenweise abgearbeitet, wobei stets die zuletzt gelesene Anweisung verbindlich ist, so daß Instrumentierungsanweisungen, die im Kontrollskript weiter oben stehen, durch nachfolgende Anweisungen verändert oder gar rückgängig gemacht werden können.

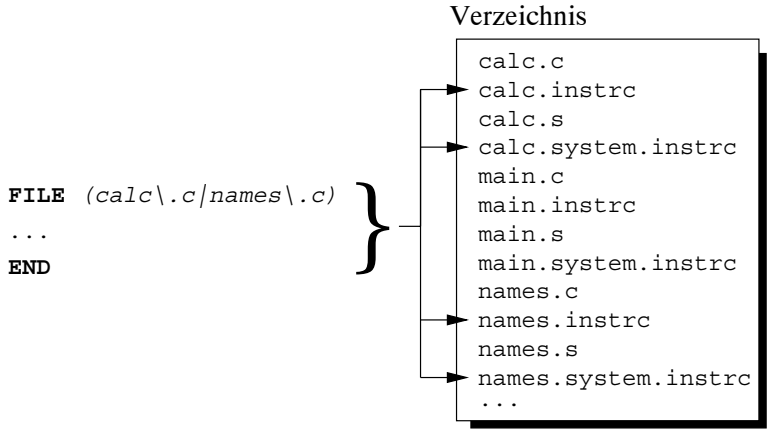


Abbildung 5.12: Abarbeitung einer `FILE`-Anweisung

Ein Kontrollskript muß mit einer `FILE`-Anweisung beginnen, um einen `FILE`-Block zu kennzeichnen und mit Hilfe des anzugebenden regulären Ausdrucks die Symboldateien der Quellcode-Dateien zu selektieren, die dann durch die Anweisungen innerhalb des Blocks manipuliert werden sollen (Abb. 5.12).

Sind alle Aweisungen des `FILE`-Blocks ausgewertet, so wird, falls weitere `FILE`-Anweisungen existieren, dieser Vorgang für die folgenden `FILE`-Blöcke wiederholt, bis das komplette Skript abgearbeitet ist.

Die `TRACE`-Anweisung dient zur Kontrolle der Instrumentierung von Benutzerfunktionen. Deshalb manipuliert diese Anweisung die „`.instrc`“ Symboldateien der durch die `FILE`-Anweisung ausgewählten Quellcode-Dateien, da die Information, ob eine Benutzerfunktion durch das Programm `vampir_asinst` intrumentiert werden soll, in diesen Symboldateien gespeichert ist.

Innerhalb des `TRACE`-Blocks können mit Hilfe von regulären Ausdrücken die Benutzerfunktionen bestimmt werden, die instrumentiert bzw. nicht instrumentiert werden sollen. Dies geschieht, indem überprüft wird, ob die Funktionsnamen der Benutzerfunktionen, die in den Symboldateien gespeichert sind, zu dem Muster passen, das durch den regulären Ausdruck festgelegt wird. Paßt der Funktionsname zu dem Muster, so wird die entsprechende Zeile der Symboldatei verändert, wobei die Direktive auf dem in der `TRACE`-Anweisung genutzten Wert gesetzt und die Funktion der Gruppe zugeordnet wird, deren Name durch die Zeichenkette bezeichnet ist, die nach dem regulären Ausdruck angegeben werden muß (Abb. 5.13).

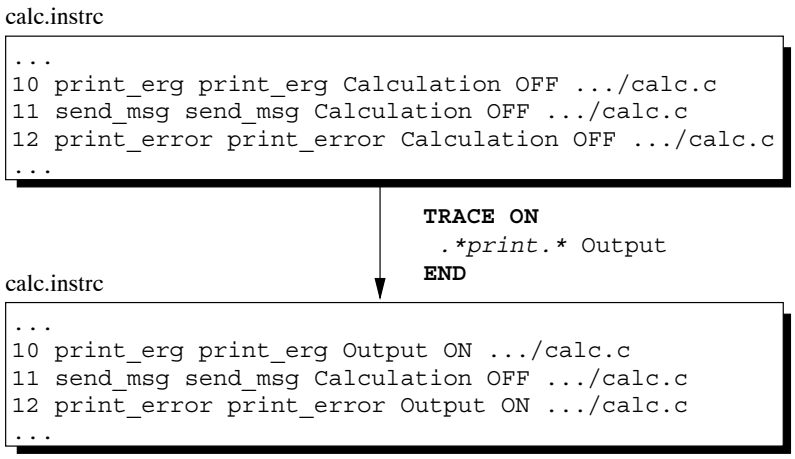


Abbildung 5.13: Abarbeitung einer `TRACE`-Anweisung

`REPLACE`-Anweisungen kontrollieren die Verwendung von Wrapper-Funktionen. Aus diesem Grund manipulieren sie die „`.system.instrc`“ Symboldateien der durch die `FILE`-Anweisung ausgewählten Quellcode-Dateien. Die Benutzerfunktionen, in denen die Verwendung von Wrapper-Funktionen verändert werden soll, können in `REPLACE`-Block mit regulären Ausdrücken bestimmt werden. Dazu wird überprüft, ob das Muster zu einem Funktionsnamen einer Benutzerfunktion innerhalb der entsprechenden „`.instrc`“ Symboldatei paßt. Ist dies der Fall, so wird die Funktionsnummer dieser Benutzerfunktion extrahiert, mit deren Hilfe die zu manipulierende Zeile in der „`.system.instrc`“ Symboldatei aufgefunden oder eine neue Zeile erzeugt werden kann.

Nach dem regulären Ausdruck müssen in `REPLACE`-Blöcken drei Zeichenketten vorhanden sein. Mit der ersten wird der Name der Funktion angegeben, deren Aufrufe

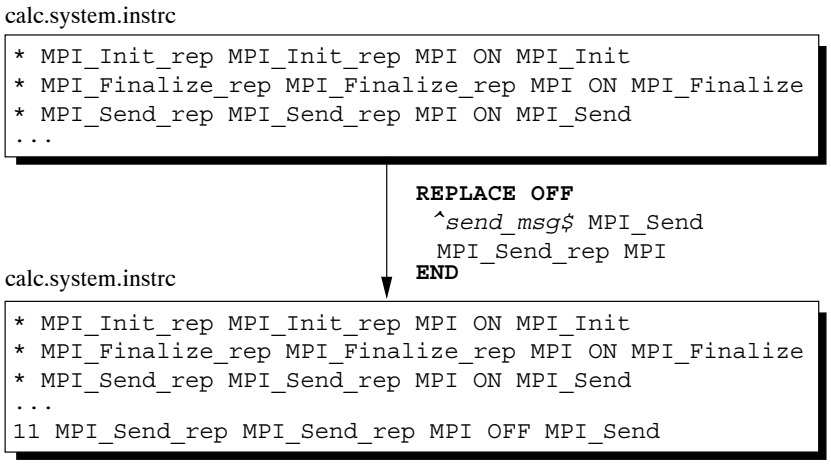


Abbildung 5.14: Abarbeitung einer REPLACE-Anweisung

in der jeweiligen Benutzerfunktion ersetzt werden sollen, mit der zweiten wird der Name der Wrapper-Funktion bestimmt und mit der dritten die Gruppe festgelegt, zu der die Wrapper-Funktion gehören soll.

Existiert bereits ein Eintrag für die Funktionsnummer mit dem Namen der Funktion, deren Aufrufe substituiert werden sollen, und der dafür verwendeten Wrapper-Funktion, so wird dieser Eintrag anhand des in der REPLACE-Anweisung genutzten Werts und dem angegebenen Gruppennamen manipuliert. Falls noch kein Eintrag existiert, wird dieser mit der Funktionsnummer und den Informationen aus dem Kontrollskript generiert und an das Ende der Symboldatei angehängt (Abb. 5.14).

5.4.3 Beispiel

In Abb. 5.15 ist ein Beispiel für ein Instrumentierungs-Kontrollskript angegeben. Dieses Kontrollskript besteht aus drei FILE-Blöcken, die in diesem Beispiel jeweils für genau eine Quellcode-Datei des zu instrumentierenden Programms angelegt sind.

Mit dem ersten FILE-Block wird die Instrumentierung der Funktionen der Quellcode-Datei „main.c“ beschrieben. Durch die REPLACE-Anweisung und dem zugehörigen Block wird festgelegt, daß in der Funktion main die Aufrufe der MPI-Funktionen MPI_Init und MPI_Finalize nicht durch Aufrufe der entsprechenden Wrapper-Funktionen der VAMPIR-Tracebibliothek substituiert werden. Dies macht die Standardeinstellung des Instrumentierungssystems rückgängig, die vorsieht, daß Aufrufe dieser MPI-Funktionen im gesamten Programm ersetzt werden sollen. Die folgende TRACE-Anweisung bestimmt, daß die Funktion main nicht instrumentiert wird, was ebenfalls die Standardeinstellung zurücksetzt.

Die zweite FILE-Anweisung und der damit verbundene Block beziehen sich auf die Funktionen der Quellcode-Datei „bcast.cc“. Mit Hilfe der TRACE-Anweisung und des angegebenen, regulären Ausdrucks wird festgelegt, daß alle Funktionen innerhalb

```
# Instrumentierung für die Datei "main.c"
FILE main\.c
   REPLACE OFF # In folgenden Funktionen Aufrufe nicht ersetzen
      main MPI_Init MPI_Init_rep MPI
      main MPI_Finalize MPI_Finalize_rep MPI
   END
   TRACE OFF # Folgende Funktionen nicht instrumentieren
      main Calculation
   END
END
# Instrumentierung für die Datei "bcast.cc"
FILE bcast\.cc
   TRACE ON # Folgende Funktionen instrumentieren
      .*bcast Broadcast
   END
   REPLACE ON # In folgenden Funktionen Aufrufe ersetzen
      .*bcast MPI_Bcast MPI_Bcast_rep MPI
   END
END
# Instrumentierung für die Datei "param.f90"
FILE param\.f90
   TRACE ON # Folgende Funktionen instrumentieren
      get_param Calculation
   END
END
```

Abbildung 5.15: Instrumentierungs-Kontrollskript Beispiel

der Quellcode-Datei instrumentiert werden, deren Funktionsnamen dem Muster „beliebig viele Zeichen, gefolgt von der Zeichenkette `bcast`“ entsprechen. Dieses Muster wird ebenfalls in dem `REPLACE`-Block verwendet, um in diesen Benutzerfunktionen die Aufrufe der Kommunikationsfunktion `MPI_Bcast` durch Aufrufe der Wrapper-Funktion `MPI_Bcast_rep` der VAMPIR-Tracebibliothek zu ersetzen.

Im letzten `FILE`-Block werden die Funktionen der Quellcode-Datei „`param.f90`“ behandelt. Durch die `TRACE`-Anweisung wird lediglich der Eintrag in der Symboldatei für die Funktion `get_param` so modifiziert, daß sie durch das Programm `vampir_asinst` instrumentiert wird.

5.5 Graphische Benutzeroberfläche

Neben den Instrumentierungs-Kontrollskripten existiert die Möglichkeit, mittels einer graphischen Oberfläche die Instrumentierung eines Programms durch das Instrumentierungssystem zu steuern.

Die Oberfläche steht dafür in zwei Varianten zur Verfügung. Einmal kann eine *online* Version während der Instrumentierung duch das Perl-Skript `inst` genutzt werden, des weiteren kann die Oberfläche in einer *offline* Version nach der Instrumentierung eingesetzt werden. Beide Varianten bieten die gleichen Möglichkeiten wie die Instrumentierungs-Kontrollskripte zur Steuerung der Instrumentierung von Programmen durch das Instrumentierungssystem.

An dieser Stelle wird die graphische Benutzeroberfläche präsentiert. Zunächst wird die offline Version beschrieben und auf die einzelnen Einstellungsmöglichkeiten eingegangen. Danach wird die online Version vorgestellt, wobei vor allem auf die Unterschiede zur offline Version eingegangen wird.

5.5.1 Offline Version

Die offline Version der graphischen Benutzeroberfläche wird über die Kommandozeile durch die Eingabe `vaip` (Vampir Assembler Instrumentation Popup) gestartet. Falls die Oberfläche aus einem Verzeichnis aufgerufen wird, in dem eine globale Symboldatei namens „`global.instrc`“ vorhanden ist, so wird diese automatisch geladen und die in der Symboldatei enthaltenen Informationen im Hauptfenster der Benutzeroberfläche angezeigt (Abb. 5.16). Anderenfalls ist die linke Hälfte des Fensters leer, die zur Darstellung der Funktionen dient, die zu den Quellcodes des zu instrumentierenden Programms gehören. Mittels des Menüpunkts `open ...` des File-Menüs können allerdings globale Symboldateien geöffnet werden.

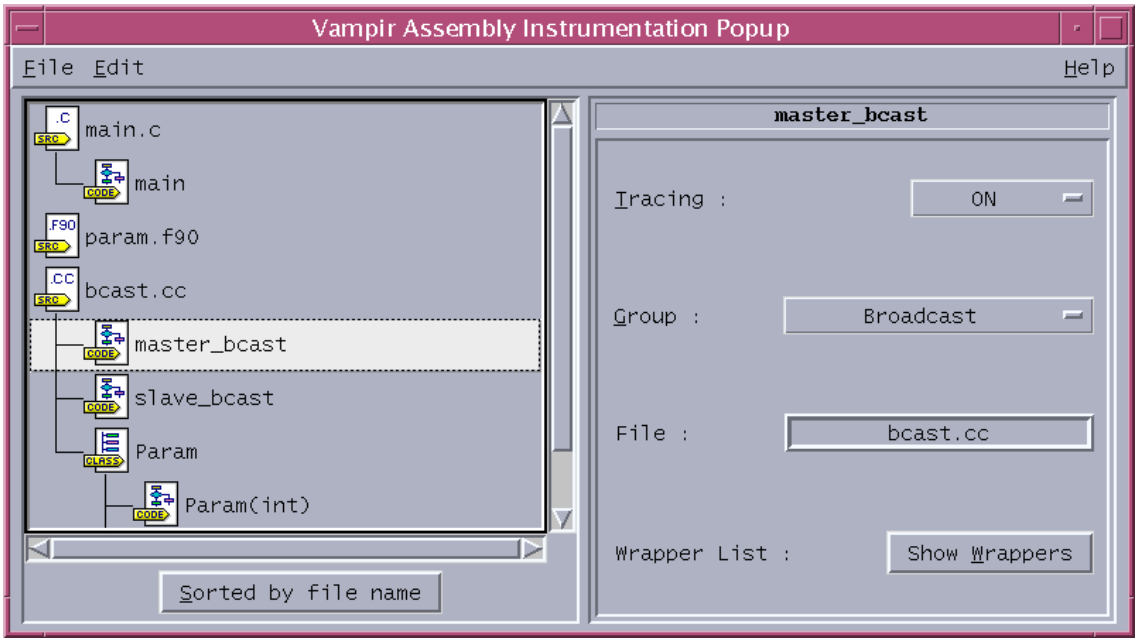


Abbildung 5.16: Hauptfenster der offline Version

Bei der Darstellung der Funktionen kann zwischen zwei Varianten gewählt werden. Die Funktionen können den Quellcodes und eventuell vorhandenen Modulen und

Klassen zugeordnet werden; es ist aber auch möglich, sie anhand der Gruppen zu sortieren. Zwischen diesen Ansichten kann durch betätigen des Funktionsknopfs gewechselt werden, der sich unterhalb der Liste der Funktionen befindet.

In der dargestellten Liste können Funktionen, aber auch Einheiten von Funktionen, also Gruppen, bzw. Dateien, Klassen oder Module ausgewählt werden. Die jeweiligen Informationen werden dann auf der rechten Hälfte des Hauptfensters angezeigt. Dort ist angegeben, ob die ausgwählte Funktion bzw. die Funktionen der ausgewählten Einheit instrumentiert bzw. nicht instrumentiert werden sollen, oder ob diese Einstellung für die Funktionen der ausgewählten Einheit variieren. Des weiteren wird, falls möglich, die Gruppe und der Name der Quellcode-Datei der ausgewählten Funktion bzw. Funktionen angezeigt.

Mit Hilfe der beiden Optionsmenüs lassen sich die Einstellungen vornehmen, ob die selektierte Funktion bzw. Einheit von Funktionen instrumentiert werden und zu welcher Gruppe diese Funktion bzw. Einheit von Funktionen gehören soll.

Über den Menüpunkt `new group ...` des Edit-Menüs können neue Gruppen angelegt werden, während es mit dem Menüpunkt `delete group ...` möglich ist Gruppen zu löschen. Sollten dadurch Funktionen zu keiner Gruppe mehr gehören, werden sie automatisch der Gruppe „Calculation“ zugeordnet.

Wird der Funktionsknopf `Show Wrappers` angeklickt, so wird ein neues Fenster geöffnet, in dem die innerhalb der ausgewählten Funktion bzw. Einheit von Funktionen genutzten Wrapper-Funktionen angezeigt werden (Abb. 5.17).

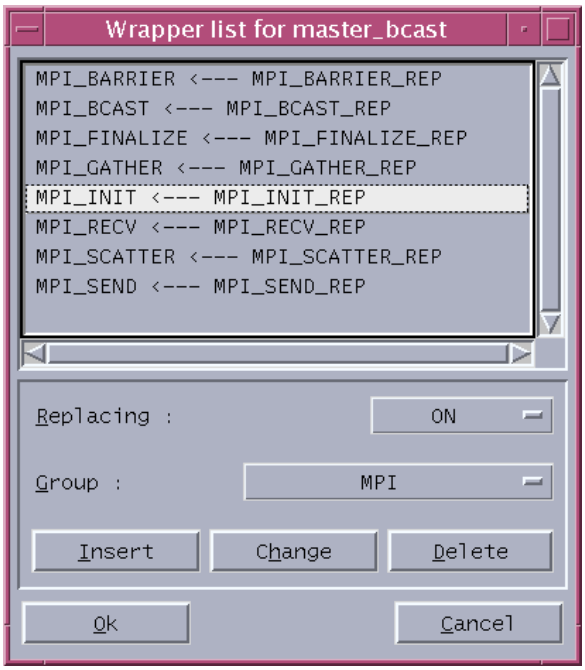


Abbildung 5.17: Einstellung der lokalen Wrapper-Funktionen

Im oberen Teil ist eine Liste mit den Namen der Funktionen zu sehen, deren Aufrufe durch Aufrufe der dort angegebenen Wrapper-Funktionen ersetzt werden, während

im unteren Teil Informationen über den jeweilig ausgewählten Eintrag dieser Liste angezeigt werden. Mit Hilfe der Optionsmenüs kann gewählt werden, ob die Substitution bei der Instrumentierung durchgeführt werden und zu welcher Gruppe die angewählte Wrapper-Funktion gehören soll.

Durch die Funktionsknöpfe `Insert`, `Change` und `Delete` können neue Funktionsaufrufersetzungen eingefügt, der ausgewählte Eintrag verändert oder gelöscht werden.

All diese Einstellungen beziehen sich auf die Funktion bzw. Einheit von Funktionen, die im Hauptfenster ausgewählt wurde.

Über den Menüpunkt `default wrappers ...` des Edit-Menüs im Hauptfenster können die Wrapper-Funktionen der VAMPIR-Tracebibliothek ausgewählt werden, die im gesamten Programm die entsprechenden Funktionen ersetzen sollen. Dazu wird ein Fenster geöffnet, in dessen oberen Teil eine Auswahlliste mit allen Wrapper-Funktionen zu sehen ist. Dabei sind die Funktionen nach den jeweiligen Kommunikationsbibliotheken sortiert und noch in die Funktionseinheiten „Managment“, „Point-to-Point“ und „Collective“ unterteilt (Abb. 5.18).

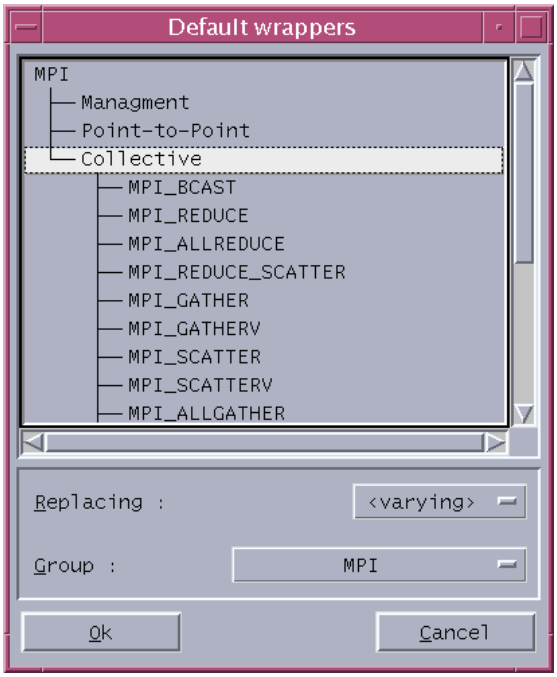


Abbildung 5.18: Einstellung der globalen Wrapper-Funktionen

Im unteren Teil werden wiederum die Informationen über den ausgewählten Eintrag angezeigt; zum einen, ob die ausgewählte Wrapper-Funktion bzw. Einheit von Wrapper-Funktionen zur Substitution genutzt wird und, zum anderen, der Name der Gruppe, zu der die Wrapper-Funktion bzw. die Einheit von Wrapper-Funktionen gehört. Mit den Optionsmenüs können diese Einstellungen verändert werden.

Das File-Menü des Hauptfensters beinhaltet den Menüpunkt `save`, mit dem alle Änderungen gespeichet werden. Dabei werden alle notwendigen Änderungen sowohl

an der globalen Symboldatei als auch an den Symboldateien der Quellcodes vorgenommen, die durch die in der globalen Symboldatei gespeicherten Informationen leicht aufgefunden werden können.

Um die offline Version der Benutzeroberfläche nutzen zu können, muß das zu instrumentierende Programm zumindest einmal mittels des Perl-Skriptes `inst` verarbeitet werden, um eine globale Symboldatei für das Programm zu erzeugen. Dann können alle gewünschten Änderungen an der Instrumentierung vorgenommen, die entsprechenden Objektcode-Dateien und das ausführbare Programm gelöscht und schließlich das Programm erneut mit dem Skript `inst` instrumentiert werden. Danach steht das neu instrumentierte, ausführbare Programm zur Verfügung.

Der wesentliche Vorteil der offline Version ist, daß alle Informationen über das zu instrumentierende Programm zu diesem Zeitpunkt bekannt sind, d. h. alle Funktionen sowie Quellcode-Dateien des Programms können angezeigt und bearbeitet werden.

5.5.2 Online Version

Die online Version ist eine leicht modifizierte Variante der offline Version, um mit der graphischen Oberfläche auch während des Instrumentierungsprozesses die zu instrumentierenden Elemente des Programms auswählen zu können.

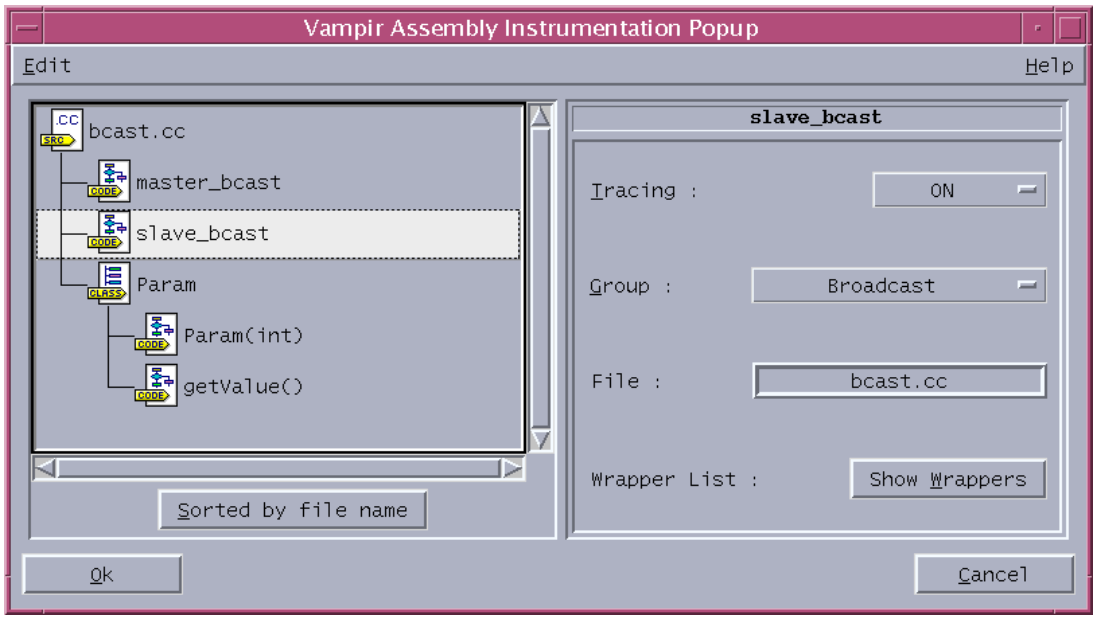


Abbildung 5.19: Hauptfenster der online Version

Wird die Oberfläche eingesetzt, so wird zu Beginn der Instrumentierung das Fenster zur globalen Einstellung der zu verwendenden Wrapper-Funktionen (Abb. 5.18) angezeigt, um die Wrapper-Funktionen auszuwählen, die im gesamten Programm genutzt werden sollen. Dieses Fenster ist nicht mehr über den Menüpunkt `default`

`wrappers` ... des Hauptfensters zu erreichen. Die Auswahl der zu nutzenden Wrapper-Funktionen kann mit dem Funktionsknopf `Ok` bestätigt werden, wodurch eine Datei mit dem Namen „`default_wrappers`“ angelegt wird, in der die gewählten Wrapper-Funktionen aufgelistet sind. Diese Datei wird vom Perl-Skript `inst` an Stelle der Standardeinstellung zur Substitution von Funktionsaufrufen genutzt. Solange diese Datei im Verzeichnis existiert, tritt das Fenster zur Auswahl der globalen Wrapper-Funktionen nicht mehr in Erscheinung.

Bevor vom Perl-Skript `inst` das Instrumentierungswerkzeug `vampir_asinst` aufgerufen wird, öffnet sich gegebenenfalls das Hauptfenster der Benutzeroberfläche. In diesem werden allerdings nur die Funktionen der Quellcode-Dateien angezeigt, die an dem an das Perl-Skript übergebenen Compiler-Aufruf beteiligt sind. Diese Funktionen können - genau wie in der offline Version - ausgewählt und somit die Instrumentierung gesteuert werden.

Das Hauptfenster besitzt in der online Version kein File-Menü. Deshalb existieren zwei weitere Funktionsknöpfe `Ok` und `Cancel` am unteren Rand des Fensters, um die Einstellungen zu übernehmen bzw. zu verwerfen (Abb. 5.19).

An dem Einstellungsfenster zur Kontrolle der Wrapper-Funktionen innerhalb von Benutzerfunktionen (Abb. 5.17) wurden keine Änderungen vorgenommen, so daß es genau wie in der offline Version genutzt werden kann.

Der Vorteil der online Version ist, daß die Instrumentierung während des Instrumentierungsprozesses beeinflußt werden kann und somit eine Kontrolle der Instrumentierung bereits während der ersten Anwendung des Instrumentierungssystems möglich ist.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Das Laufzeitverhalten paralleler Programme ist im voraus nicht zuverlässig zu bestimmen, so daß der Entwicklungsprozeß vom ständigen Wechsel zwischen Beobachtung und anschließender Modifikation der Programme gekennzeichnet ist, bis die gewünschten Anforderungen erfüllt werden.

Insbesondere hat sich die Methode des Tracings bei der Analyse von Massage-Passing-Programmen als nützlich erwiesen, bei der die benötigten Laufzeitdaten in eine Ereignisspur aufgezeichnet werden, die mittels geeigneter Werkzeuge nach Beendigung der Programme verarbeitet werden kann. Dazu ist eine Instrumentierung der Programme notwendig, so daß während der Laufzeit der Programme alle relevanten Ereignisse protokolliert werden.

Zur automatisierten Integration von Anweisungen in die Programme, existieren zwei konventionelle Ansätze; zum einen die Quellcode-Instrumentierung, bei der die Anweisungen in den Quellcode des Programms eingefügt werden, und zum anderen die Objektcode-Instrumentierung, die die notwendigen Modifikationen direkt an den ausführbaren Programmen vornimmt.

Beide Ansätze besitzen sowohl Vor- als auch Nachteile, so daß Interesse an der Untersuchung neuer Ansätze besteht, die möglichst die Vorteile der gängigen Ansätze kombinieren, dabei aber deren Nachteile weitgehend vermeiden.

Der im Rahmen dieser Diplomarbeit untersuchte Ansatz, die Instrumentierung von Assembler-Quellcodes, bietet im wesentlichen die Vorteile der Quellcode-Instrumentierung, kann jedoch für alle Programmiersprachen verwendet werden und beinhaltet somit positive Eigenschaften der Objektcode-Instrumentierung.

Implementiert wurde der Ansatz durch die Erweiterung des Assemblercode-Manipulationssystems SALTO um eine Bibliothek namens ASINST, die Funktionen enthält, mit denen es möglich ist, systemunabhängig Funktionsaufrufe in Assemblercodes zu integrieren und Aufrufe von Funktionen zu substituieren.

Exemplarisch wurde auf der Basis dieser Erweiterung der Instrumentierer `vampir_asinst` entwickelt, der Assembler-Quellcodes so verändert, daß Ereignisspuren für das Analysewerkzeug VAMPIR generiert werden können.

Des weiteren wurde mit der Entwicklung eines Instrumentierungssystems gezeigt, daß diese Art der Instrumentierung benutzerfreundlich durchführbar ist, obwohl der Umweg über die Assembler-Quellcodes während der Übersetzung der Programme zusätzliche Arbeitschritte erfordert.

Dieses Instrumentierungssystem wurde mittels eines Perl-Skriptes mit dem Namen `inst` implementiert, das dem Benutzer alle notwendigen Schritte zur Erzeugung eines instrumentierten Programms abnimmt. Es kann sowohl von der Kommandozeile gestartet als auch in Makefiles verwendet werden, so daß sich auch komplexe Programme leicht instrumentieren lassen.

Um in den Instrumentierungsprozeß einzugreifen, stehen dem Benutzer zwei Möglichkeiten zur Verfügung, die zu diesem Zweck erstellt wurden:

- Mittels einer kleinen Instrumentierungs-Kontrollskriptsprache, die aus einigen wenigen Anweisungen besteht, können Kontrollskripte geschrieben werden, die die Instrumentierung während der Bearbeitung des Programms durch das Instrumentierungssystem steuern.
- Eine interaktive Auswahl der zu instrumentierenden Elemente des Programms kann mit Hilfe einer graphischen Oberfläche getroffen werden, die in einer online Version, die während der Ausführung des Perl-Skriptes `inst` verwendet wird und in einer offline Version, die nach der Durchführung der Instrumentierung eingesetzt werden kann, implementiert wurde.

Allerdings existieren einige schwerwiegende Probleme bei der Instrumentierung von Assembler-Quellcodes, die sich in zwei Gruppen einteilen lassen. Einerseits resultieren die Probleme aus der Verwendung des SALTO-Systems, andererseits treten Probleme mit den zur Erzeugung der Assembler-Quellcodes verwendeten Compilern auf.

Auch wenn die Probleme, die durch die unvollständige Funktionalität von SALTO hervorgerufen werden, mit geeigneten Transformationen der Assembler-Quellcodes minimiert werden können, stellen diese jedoch die zuverlässige Nutzung des Instrumentierungssystems in Frage.

Noch gravierender sind allerdings die Probleme mit den verwendeten Compilern, die zum Teil nicht in der Lage sind, die von ihnen erzeugten Assemblercodes zu übersetzen. Dieses Problem tritt vor allem bei komplexen Compilern für moderne Programmiersprachen wie C++ und Fortran90 auf. Diese offensichtlichen Fehler in den Compilern lassen sich nicht ohne Unterstützung der jeweiligen Compiler-Hersteller beheben, womit in naher Zunkunft nicht zu rechnen ist, da die Option, Assemblercodes erzeugen zu können, in den letzten Jahren an Bedeutung verloren hat.

Aufgrund dieser Probleme ist die Assembler-Instrumentierung zur Zeit nur beschränkt einsetzbar, wobei eine weitgehend zuverlässige Verwendung nur für Programme möglich ist, die in den Programmiersprachen C und Fortran77 implementiert sind, in denen allerdings immer noch ein Großteil der heutigen Quellcodes verfaßt werden.

6.2 Ausblick

Zur flexibleren Nutzung der Bibliothek ASINST wäre eine Erweiterung mit zusätzlichen Funktionen, die beispielsweise die Instrumentierung von Schleifen erlauben, und die Portierung der Bibliothek auf weitere Systeme, denkbar.

Außerdem wäre es sinnvoll, die Bibliothek ASINST für die Implementierung weiterer Instrumentierer zu verwenden, um andere Analysewekzeuge bzw. Ereignisspur-Formate zu unterstützen.

Ein großes Arbeitsfeld wäre die Ersetzung des Systems SALTO durch ein besser geeignetes Werkzeug, so daß die SALTO-spezifischen Probleme wegfallen würden.

Geplant ist bereits die Wiederverwertung der graphischen Benutzeroberfläche für das Leistungsanalyse-Werkzeug PAT der Firma Cray Research Inc., um die Auswahl der mit PAT zu instrumentierenden Funktionen ebenfalls benutzerfreundlich realisieren zu können.

Anhang A

Installation von SALTO auf CRAY T3E

Um das System SALTO auf das System CRAY T3E unter dem Betriebssystem Unicos/mk zu installieren, müssen einige Änderungen an der Makefile-Datei sowie an zwei Quellcode-Dateien vorgenommen werden.

Bei der Installation des Basissystems muß die verwendete Architektur und der zu nutzende C++-Compiler in der Datei „`config.makefile`“ ausgewählt werden, indem die entsprechenden Zeilen editiert werden. Für das System CRAY T3E sind jedoch keine Einträge vorhanden. Deshalb müssen die folgenden Zeilen der Datei „`config.makefile`“ hinzugefügt werden:

```
# --- KCC under CRAY T3E unicos/mk 2.0.3.24
CPP = KCC
CPPFLAGS = -c -Dirix -Dsalto_cray -D__EXTENSIONS__ \
           $(COMMONFLAGS) $(DEBUG)
CC = cc -fpic
CFLAGS = -c -Kpic $(COMMONFLAGS)
LD = $(CPP)
LDFLAGS =
```

Diese Zeilen bereiten die Installation auf das System CRAY T3E mittels des C++-Compilers KCC vor.

Des weiteren müssen Änderungen an der Makefile-Datei vorgenommen werden, die sich im Verzeichnis „`src`“ befindet. Dort müssen diese Modifikationen vorgenommen werden, wobei die Original-Anweisungen in Kommentare umgewandelt sind:

```
...
# PLATFORM="`uname -p`"
PLATFORM="`uname -m`"
```

```
...
# all: libsalto.so
all: libsalto.a

# libsalto.so: $(OFILES)
#   $(LD) $(LDFLAGS) -o libsalto.so $(OFILES)
libsalto.a: $(OFILES)
   $(LD) $(LDFLAGS) -o libsalto.a $(OFILES)
...
#install: libsalto.so
#   /bin/cp -f libsalto.so ../lib
#   chmod a+rx ../lib/libsalto.so

install: libsalto.a
   /bin/cp -f libsalto.a ../lib
   chmod a+rx ../lib/libsalto.a
...
veryclean: clean
#   /bin/rm -f libsalto.so
   /bin/rm -f libsalto.a
```

Abschließend muß noch dafür gesorgt werden, daß in den Quellcode-Dateien „`as.cc`“ und „`rtl.cc`“ die Dateien „`unistd.h`“ bzw. „`malloc.h`“ eingefügt werden.

Dazu sollten in der Datei „`as.cc`“ ab Zeile 194 drei Zeilen integriert werden:

```
#ifdef salto_cray
#include <unistd.h>
#endif
```

In der Quellcode-Datei „`rtl.cc`“ sollten die Zeilen 67 bis 71 durch die folgenden Zeilen ersetzt werden:

```
#if defined _WIN32
#include <malloc.h>
#elif defined salto_cray
#include <malloc.h>
#else
#include <alloca.h>
#endif
```

Nach diesen Änderungen kann das System SALTO, wie in der Dokumentation beschrieben, mit Hilfe des C++-Compilers KCC auf das System CRAY T3E installiert werden.

Zu beachten ist, daß für die Installation des Zielsystems CRAY T3E die für diese Diplomarbeit modifizierte Maschinenbeschreibung mit dem zugehörigen systemspezifischen Programmteil genutzt werden sollte, um eine stabilere Arbeitsweise des SALTO-Systems zu gewährleisten.

Anhang B

Installation des Instrumentierungssystems

Um das Instrumentierungssystem zu installieren, müssen die folgenden Schritte durchgeführt werden:

1. SALTO muß auf das System mit der entsprechenden Maschinenbeschreibung und dem zugehörigen systemspezifischen Programmteil installiert werden.

2. Ein Verzeichnis, in dem das Instrumentierungssystem abgelegt werden soll, muß gewählt bzw. angelegt werden, z. B. „`/home/testbed/`“. Es sind keine *root* Privilegien erforderlich, um das System zu installieren.

3. Nun kann das Archiv „`asinst.tar`“ in das gewählte Verzeichnis ausgepackt werden, wobei automatisch das Verzeichnis „`Asinst`“ erzeugt wird.

4. Mittels des Befehls `cd` muß das aktuelle Verzeichnis auf das Verzeichnis „`Asinst`“ gewechselt werden.

5. Die Umgebungsvariable `ASINST_HONE` muß auf dieses Verzeichnis gesetzt werden, z. B. :

   ```
   setenv ASINST_HOME /home/testbed/Asinst #csh/tcsh shell

   ASINST_HOME=/home/testbed/Asinst #sh shell
   export ASINST_HOME
   ```

   Diese Umgebungsvariable sollte, falls das System häufig genutzt wird, in der Datei „`.profile`“ gesetzt werden.

6. Die Datei „`config.makefile`“ muß editiert werden, um das Zielsystem auszuwählen, indem die Kommentarzeichen vor den entsprechenden Einträgen gelöscht werden.

7. Nun kann in das Verzeichnis „`src`“ gewechselt und mittels `make T3E` die Installation für das System CRAY T3E bzw. mittels `make SPARC` für auf Sparc Prozessoren basierende Systeme gestartet werden.

Nach diesen Schritten sollte das Instrumentierungssystem vollständig installiert sein. Im Verzeichnis „`bin`“ sind dann alle ausführbaren Programme und Skripte des Systems zu finden, so daß der Pfad zu diesem Verzeichnis zu den Pfaden der ausführbaren Programme hinzugefügt werden sollte.

Anhang C

Portierung der Bibliothek ASINST

In diesem Teil des Anhangs werden die Funktionen beschrieben, die für eine Portierung der Bibliothek ASINST neu implementiert werden müssen. Zusätzlich wird auf die beiden Assembler-Funktionen eingegangen, die zur Rettung und Wiederherstellung der Inhalte der globalen Register dienen. Auch diese Funktionen müssen für jedes zu unterstützende System zur Verfügung gestellt werden.

C.1 Systemspezifische Funktionen

Wie schon im Kapitel 5 beschrieben, besteht die Bibliothek ASISNT aus einem systemspezifischen und einem systemunabhängigen Programmteil. Die systemspezifischen Funktionen sind in der Quellcode-Datei mit dem Namen „`asinst-`*`machine`*`.cc`“ untergebracht. Zur Zeit existiert diese Quellcode-Datei für das System CRAY T3E und für auf Sparc Prozessoren basierende Systeme, also „`asinst-t3e.cc`“ und „`asinst-sparc.cc`“.

Insgesamt beinhalten diese Quellcode-Dateien acht Funktionen sowie zwei Zeichenketten, die die notwendigen Instruktionen in der jeweiligen Assemblersprache bereitstellen, um die Assembler-Funktionen zur Rettung und Wiederherstellung der globalen Registerinhalte aufzurufen.

Die Funktionen bedienen sich zum Teil einer Klasse namens `String`, die für diese Arbeit entwickelt wurde, um den Umgang mit Zeichenketten zu erleichtern.

An dieser Stelle werden die Instruktionen-Zeichenketten und die einzelnen Funktionen sowie ihre Aufgaben vorgestellt.

`reg_svg` und `reg_rst`

Um den Aufruf der Assembler-Funktionen `_save_G` bzw. `_restore_G` in einen Assembler-Quellcode integrieren zu können, müssen die dazu verwendeten Register gerettet werden. Exemplarisch ist die Zeichenkette zur Integration des Aufrufs der

Funktion `_save_G` mit Instruktionen in der Assemblersprache für das System CRAY T3E [9, 12] zu sehen:

```
static char *reg_svg =
    "\tsubq r30, 16, r30\n"
    "\tstq r26, 8(r30)\n"
    "\tstq r9, 0(r30)\n"
    "\tla r9, _save_G\n"
    "\tjsr r26, (r9)\n";
```

Zu Beginn wird mit der Instruktion `sub r30,r16,r30` Platz auf dem Stack reserviert, dann die beiden betroffenen Register `r9` und `r26` auf dem Stack gesichert. Nun kann die Adresse der Funktion `_save_G` in das Register `r9` geladen und mit der Instruktion `jsr r26, (r9)` die Funktion aufgerufen werden, wobei die Rücksprungadresse im Register `r26` abgelegt wird.

Die Zeichenkette `reg_rst` beinhaltet die Instruktionen zum Aufruf der Funktion `_restore_G`:

```
static char *reg_rst =
    "\tla r9, _restore_G\n"
    "\tjsr r26, (r9)\n"
    "\tldq r9, 0(r30)\n"
    "\tldq r26, 8(r30)\n"
    "\taddq r30, 16,r30\n";
```

Hier wird zuerst die Funktion `_restore_G` aufgerufen, dann die beiden Register `r9` und `r26` mit den vor dem Aufruf der Funktionen `_save_G` geretteten Inhalten wiederhergestellt. Abschließend wird der Speicherplatz auf dem Stack wieder freigegeben.

Die in den beiden Zeichenketten enthaltenen Instruktionen müssen also einen in den Assembler-Quellcode zu integrierenden Funktionsaufruf umgeben.

```
void CodeForFunctionCall(char *newcode, String func_name, int num_of_args,
                         int args[ ])
```

Diese Funktion liefert eine Zeichenkette (`newcode`) zurück, in der alle nötigen Instruktionen zum Aufruf einer Funktion enthalten sind. Dazu muß der Funktion der Name der aufzurufenden Funktion, die Anzahl der Argumente und ein Feld, das die Argumente enthält, übergeben werden. Hier ist ein Auszug aus dieser Funktion für die Assemblersprache für Sparc Prozessoren [33] abgebildet:

```
...
strcpy(newcode, reg_svg);
for (int i=0; i<num_of_args; i++) {
```

```
            sprintf (buffer,
            "\tsethi %%hi(%u),%%o%u\n"
            "\tor %%o%u, %%lo(%u),%%o%u\n",
            args[i], i, i, args[i], i);

            strcat (newcode, buffer);}
    sprintf (buffer, "\tcall %s\n\tnop\n", func_name.string());
    strcat (newcode, buffer);
    strcat (newcode, reg_rst);
    ...
```

Zuerst werden die Instruktionen der Zeichenkette `reg_svg` in die Zeichenkette `newcode` kopiert, dann in der `for`-Schleife die Instruktionen zum Laden der Register mit den entsprechenden Argumenten der aufzurufenden Funktion erzeugt und an die Zeichenkette `newcode` angehängt. Zuletzt werden die Instruktionen für den Funktionsaufruf und die Zeichenkette `reg_rst` an die Zeichenkette angefügt, in der nun alle notwendigen Instruktionen enthalten sind, so daß diese Zeichenkette in den Assembler-Quellcode an einer geeigneten Stelle eingefügt werden kann.

void GetBeginInsertPosition(CFG *procedure, **int** &block_number,
int &insert_position)

Diese Funktion liefert über die Variable `insert_position` die Position und über die Variable `block_number` die Nummer des Basisblocks innerhalb der anzugebenen Prozedur zurück, an der Assembler-Instruktionen am Beginn dieser Prozedur eingefügt werden können. In der Regel ist dies die Position Null im ersten Basisblock. In Assembler-Quellcodes für Sparc Prozessoren kann aber die Instruktion `unimp` am Anfang einer Prozedur auftreten. Diese Instruktion gehört eigentlich an das Ende der zuvor definierten Prozedur und muß deshalb an dieser Stelle verbleiben, um das korrekte Verhalten des Programms zu gewährleisten.

```
...
if ( (block→numberOfAsm()) &&
   (!strncmp(block→getAsm(0)→unparse(), "\tunimp",6)) )
{ insert_position = 1;}
...
```

In solchen Fällen ist als Position, an der Instruktionen in die Prozedur aufgenommen werden können, der Wert eins zurückzuliefern.

int CheckIfEndOfProcedure (CFG *procedure, BB *block, INST *instruction,
int &block_number, **int** asm_number)

Mit Hilfe dieser Funktion wird überprüft, ob die angegebene Instruktion in der übergebenen Prozedur und dem zugehörigen Basisblock das Ende einer Prozedur anzeigt und liefert die Position zurück, an der in diesem Basisblock vor der Beendigung der Prozedur Instruktionen eingefügt werden können. Sollte keine geeignete Stelle gefunden werden, so gibt die Funktion den Wert `-1` zurück. Exemplarisch für das System CRAY T3E:

```
...
int insert_position = -1;
if ( (!(strcmp (instruction→getName(), "ret")))
   || (!( strncmp (instruction→unparse(),"\tjsr r26,(r9),$END",18))) )
{
     insert_position = asm_number;
}
return insert_position;
...
```

Dabei wird die Instruktion `ret` in von C/C++-Compilern erzeugten Assembler-Quellcodes verwendet, während der Aufruf der Funktion `$END` in von Fortran-Compilern generierten Assemblercodes genutzt wird, um eine Prozedur zu beenden.

Bei Assembler-Quellcodes für Sparc Prozessoren wird das Ende einer Prozedur durch die Kombination zweier Instruktionen angezeigt. Dies können die Kombinationen `ret/restore`, `jupl/restore` und `nop/call _END` sein. All diese Kombinationen müssen abgeprüft werden. Erschwerend kommt hinzu, daß diese Instruktionen über zwei Basisblöcke verteilt sein können. Dann muß neben der Position zur Integration von Assembler-Instruktionen auch noch die Nummer des entsprechenden Basisblocks zurückgeliefert werden.

int CheckIfExitCall(BB *block)

Zur Überprüfung, ob am Ende eines Basisblocks ein Aufruf der Systemfunktion `exit` vorhanden ist, existiert die Funktion `CheckIfExitCall`. Exemplarisch für das System CRAY T3E:

```
...
if ((current_asm_inst → isCall())
 && (!strncmp(current_asm_inst → unparse(), "\tjsr r26,(r9),exit",18)))
{ insert_position = num_of_asm_inst-1;}
...
```

Die Funktion liefert gegebenenfalls die Position zurück, an der Instruktionen vor dem Aufruf dieser Systemfunktion innerhalb des Basisblocks integriert werden können. Anderenfalls liefert die Funktion den Wert `-1`.

int CheckIfAbortCall(BB *block)

Im wesentlichen entspricht diese Funktion der Funktion `CheckIfExitCall`, nur daß in diesem Fall geprüft wird, ob ein Aufruf der Systemfunktion `abort` vorhanden ist.

void GetBlockBeginInsertPosition(BB *block, **int** &insert_position)

Diese Funktion wird genutzt, um die Position zu ermitteln, an der am Beginn eines Basisblocks Instruktionen eingefügt werden können. In der Regel ist dies die Position Null, doch bei Assemblercodes für Sparc Prozessoren muß wie bei der Funktion `GetBeginInsertPosition` darauf geachtet werden, daß eventuell vorhandene `unimp` Instruktionen an der Position Null verbleiben.

void GetBlockEndInsertPosition(BB *block, **int** &insert_position)

Diese Funktion liefert die Position zurück, an der am Ende des anzugebenden Basisblocks Instruktionen eingefügt werden können. Exemplarisch ein Auszug aus dem Quellcode für das System CRAY T3E:

```
...
num_of_asm = block → numberOfAsm();
if (num_of_asm > 0) {
    insert_position = num_of_asm;
    if (num_of_asm > 1) {
        instruction = block→getAsm(num_of_asm - 1);
        if (instruction→isCTI()) {
            insert_position = num_of_asm - 1;
            if (instruction→isBranch()) {
                if (num_of_asm > 2) {
                    insert_position = num_of_asm - 2;
                }
            }
        }
    }
}
...
```

Dabei ist zu beachten, daß der Basisblock mit einer Verzweigung oder sogar mit einer bedingten Verzweigung enden kann. Deshalb müssen Instruktionen vor diesen Verzweigungen eingefügt werden, und bei bedingten Verzweigungen müssen die neuen Instruktionen bereits vor derjenigen Instruktion integriert werden, die den in der Bedingung verwendeten Wert liefert.

void CheckMachine(String name)

Mit der Funktion `CheckMachine` kann geprüft werden, ob die geladene Maschinenbeschreibung zur verwendeten Bibliothek paßt. Ist dies nicht der Fall, so wird das Programm abgebrochen.

C.2 Assembler-Funktionen

Um die Inhalte der globalen Register zu retten und wiederherzustellen, müssen zwei Assembler-Funktionen implementiert werden, die diese Aufgabe übernehmen. Diese Funktionen besitzen die Namen `_save_G` und `_restore_G` und sind in der Datei „`stack-`*machine*`.s`“ gespeichert. Zur Zeit existieren diese Funktionen für das System CRAY T3E und für auf Sparc Prozessoren basierende Systeme, die in den Dateien „`stack-t3e.s`“ und „`stack-sparc.s`“ zu finden sind. Hier sind die beiden Funktionen für Sparc Prozessoren aufgeführt:

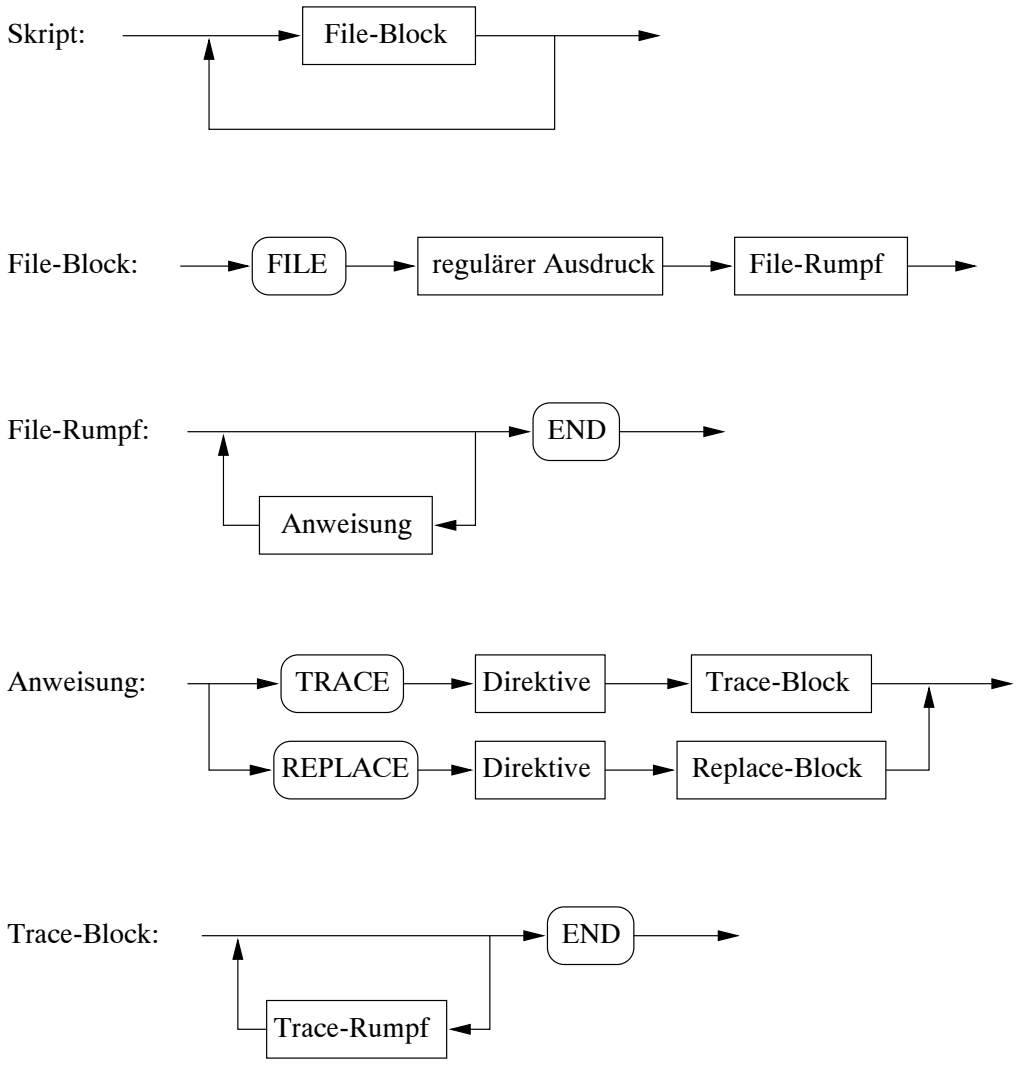
```
.section ".data"
.reserve _les_regs_g,28,".bss"
.section ".text"
.global _save_G
_save_G:
set _les_regs_g,%o1
st %g1,[%o1]
st %g2,[%o1+4]
st %g3,[%o1+8]
st %g4,[%o1+12]
st %g5,[%o1+16]
st %g6,[%o1+20]
retl
st %g7,[%o1+24]

.global _restore_G
_restore_G:
set _les_regs_g,%o1
ld [%o1],%g1
ld [%o1+4],%g2
ld [%o1+8],%g3
ld [%o1+12],%g4
ld [%o1+16],%g5
ld [%o1+20],%g6
retl
ld [%o1+24],%g7
```

Zu Beginn wird mittels der Pseudo-Instruktion `.reserve` Speicherplatz für die sieben zu rettenden globalen Register angefordert. In der Funktion `_save_G` wer-

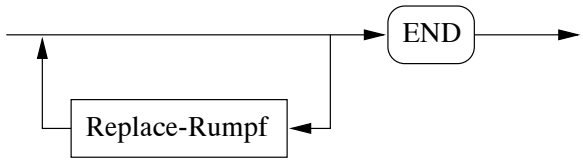
den die Registerinhalte in diesem Speicherplatz gesichert, während in der Funktion `_restore_G` die Register mit den Werten aus dem Speicherplatz geladen werden.

Anhang D

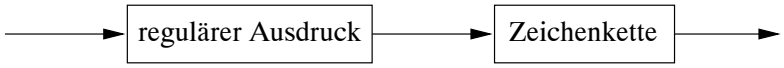
Syntaxdiagramme der Kontroll-skriptsprache



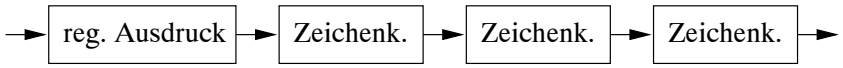
Replace-Block:



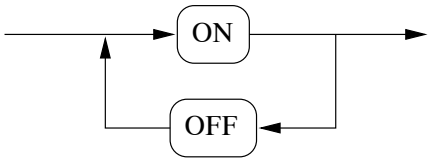
Trace-Rumpf:



Replace-Rumpf:



Direktive:



Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] A. Arnold: *Erzeugung von VAMPIR-Tracefiles mit dem Portland-HPF-Compiler*, Technische Kurzinformation, FZJ-ZAM-TKI-311, Forschungszentrum Jülich, Juni 1997

[2] A. Arnold, U. Detert, W. E. Nagel: *Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding*, erschienen in: R. Winget (Hrsg.), K. Winget (Hrsg.): *Proc. of Cray User Group Meeting*, Denver, CO, März 1995, S. 252-258

[3] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, V. Sunderam: *A user's Guide to PVM: Parallel virtual machine*, Technical Report ORNL/TM-11826, Oak Rigde National Laboratory, 1991

[4] B. P. Miller, J. M. Cargille, R. B. Irvin, K. Kunchithapadam, M. D. Callaghan, J. K. Hollingsworth, K. L. Karavanic, T. Newhall: *The Paradyn Parallel Performance Measurement Tools*, erschienen in: *IEEE Computer* 28, IEEE Computer Society Press, November 1995

[5] B. Stroustrup: *The C++ Programming Language*, 3rd edition, Addison-Wesley, 1997

[6] Compaq Computer Corporation: *Alpha Architecture Handbook*, Reference Manual, Version 4, Oktober 1998

[7] Compaq Computer Corporation: *Alpha 21164 Microprocessor*, Hardware Reference Manual, Dezember 1998

[8] Cray Research Inc. : *Application Programmer's Library*, Cray Reference Manual SR-2165 3.1, Volume 2, August 1998

[9] Cray Research Inc. : *Cray Assembler for MPP (CAM)*, Cray Reference Manual SR-2510 2.2, September 1996

[10] Cray Research Inc. : *Introducing the MPP Apprentice Tool*, Cray Manual IN-2511 2.0, November 1995

[11] D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyasta, T. Birkett, D. W. Jensen, A. A. Nazief, B. K. Totty: *Scalable Performance Environments for Parallel*

Systems, erschienen in: *Proc. 6th Distributed Memory Computing Conference*, IEEE Computer Society Press, 1991, S. 562-569

[12] Digital Equipment Corporation: *Assembler Language Programmer's Guide* DEC Manual, August 1994

[13] Digital Equipment Corporation: *ATOM User Manual*, DEC Manual, März 1994

[14] E. Rohou, F. Bodin, A. Seznec, G. L. Fol, F. Charot, F. Raimbault: *SALTO: System for Assembly-Language Transformation and Optimization*, interner Bericht Nr. 1032, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Campus Universitaire de Beaulieu, Frankreich, Juni 1996

[15] F. Bodin, Z. Chamski, E. Rohou, A. Senzec: *Functional Specification of SALTO: A Retargetable System for Assembly Language Transformation and Optimization rev. 1.2pl2*, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Campus Universitaire de Beaulieu, Frankreich, Oktober 1997

[16] F. Wolf: *EARL - Eine programmierbare Umgebung zur Bewertung paralleler Prozesse auf Message-Passing-Systemen*, Technischer Bericht, FZJ-ZAM-JÜL-3551, Forschungszentrum Jülich, Juni 1998

[17] F. Wolf, B. Mohr: *EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs*, Technischer Bericht, FZJ-ZAM-IB-9803, Forschungszentrum Jülich, April 1998

[18] I. Forster: *Designing and Building Parallel Programs*, Addison-Wesley, 1995

[19] Intel Corporation: *Paragon C System Calls*, Intel Reference Manual, Oktober 1993

[20] J. Galarowicz, B. Mohr: *Analysing Message Passing Programs on the Cray T3E with PAT and VAMPIR*, erschienen in: *Proc. of Fourth European CRAY-SGI MPP Workshop*, Garching/Munich, Germany, September 1998

[21] J. K. Hollingsworth, B. P. Miller, J. Cargille: *Dynamic Program Instrumentation for Scalable Performance Tools*, erschienen in: *Proc. Scalable High Performance Computing Conference*, Knoxville TN, May 1995

[22] J. K. Osterhout: *Tcl and the Tk Toolkit*, Addison-Wesley, 1993

[23] L. A. De Rose, Y. Zhang, R. A. Aydt: *SvPablo Guide*, Pablo Research Group, Department of Computer Science, University of Illinois, September 1997

[24] L. Wall, T. Christiansen, R. L. Schwartz: *Programming in Perl*, 2nd edition, O'Reilly & Associates Inc., 1996

[25] M. D. Smith: *Tracing with pixie*, Technical Report, Havard University, Cambridge, 1991

[26] M. Röth, A. Arnold: *Vampir.inst: Tool for Instrumentation of FORTRAN Programs to generate Trace Files*, Technical Information, FZJ-ZAM-TKI-302, Forschungszentrum Jülich, Juni 1997

[27] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra: *MPI: The Complete Reference*, MIT Press, 1996

[28] M. Voss: *Untersuchung der Performance-Analyse-Umgebung PABLO Release 4.0*, Technischer Bericht, FZJ-ZAM-IB-9529, Forschungszentrum Jülich, Juni 1995

[29] Pallas GmbH: *Vampir 2.0, User's Manual*, Germany, 1998

[30] R. M. Stallman: *The C Preprocessor*, Free Software Foundation, März 1997

[31] R. M. Stallman: *Using and porting GNU CC*, Free Software Foundation, Februar 1998

[32] R. M. Stallman, R. McGrath: *GNU Make*, Free Software Foundation, April 1995

[33] Sun Microsystems Inc. : *SPARC Assembly Language*, Sun Refenence Manual, August 1995

[34] Sun Microsystems Inc. : *The Ultra SPARC-IIi Processor*, Sun Technology White Paper, 1997

[35] T. Ball, J. R. Larus: *Optimally profiling and tracing programs*, erschienen in: *ACM Transactions on Programming Languages and Systems* 16, July 1994, S. 1319–1360

Danksagung

Die vorliegende Arbeit wurde als Diplomarbeit in Informatik am Lehrstuhl für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule (RWTH) Aachen erstellt.

Dem Lehrstuhlinhaber, Herrn Prof. Dr. Friedel Hoßfeld, danke ich für die Möglichkeit, diese Arbeit am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrum Jülich GmbH (FZJ) anfertigen zu können.

Herrn Prof. Dr. Thomas Bemmerl, dem Inhaber des Lehrstuhls für Betriebssysteme an der RWTH Aachen, danke ich für die Übernahme des Zweitgutachtens.

Besonderer Dank gilt Dr. Ing. Bernd Mohr für die ausgezeichnete Betreuung, für seine stete Ansprechbarkeit und für die vielen konstruktiven Diskussionen.

Für die Beantwortung meiner Fragen bezüglich SALTO bedanke ich mich bei Erven Rohou, Ph. D., vom Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA).

Den Mitarbeitern des ZAM, insbesondere Astrid Göke, Tamara Knödler und Alexander Lucas, danke ich für die freundliche und gute Arbeitsatmosphäre sowie für ihre Hilfsbereitschaft, die meine Arbeit positiv beinflußt hat.

Besonders danke ich meiner Freundin Eva-Maria Kürten, die mich mit viel Verständnis und Geduld durch die Zeit meiner Diplomarbeit begleitet hat und meinen Eltern, die mich während meiner gesamten Studienzeit tatkräftig unterstützt und mir diese Ausbildung ermöglicht haben.