# PARASITIC COMPUTING
# Projektbericht

Version 1.0

Luzian Scherrer
Jürg Reusser

6. Januar 2003

**Zusammenfassung**

Das Prinzip „Parasitic Computing" wurde erstmals im August 2001 öffentlich erwähnt [BFJB01], als es Wissenschaftlern der Notre Dame Universität im US-Bundesstaat Indiana gelang, fremde Rechenkapazität ohne Wissen und Einwilligung derer Besitzer zur Lösung mathematischer Probleme zu nutzen.

Der vorliegende Projektbericht zieht eine Schlussbilanz über die gleichnamige Diplomarbeit der Hochschule für Technik und Architekt Bern, in welcher die vorgestellte Methode zu einem vollständig programmierbaren, parasitären Rechenwerk ausgebaut wurde.

# Inhaltsverzeichnis

# 1 Parasitic Computing

„Parasitic Computing" ist die Bezeichnung für ein Vorgehen, mittels welchem durch Ausnutzung externer Ressourcen Berechnungen durchgeführt werden können. Obwohl diese Ausnutzung ohne Wissen und Einwilligung der Betroffenen geschieht, ist „Parasitic Computing" nicht mit einem Computervirus oder ähnlichem vergleichbar. Das Prinzip beruht ausschliesslich auf der Nutzung von willentlich angebotenen Kommunikationsprotokollen, dies allerdings zu anderem als dem ursprünglich vorgesehenen Zwecke.

## 1.1 Ausgangslage

Ende des Jahres 2001 haben Wissenschaftler der Universität Notre Dame im US-Bundesstaat Indiana eine Methode demonstriert, mit der man ohne Wissen und Einwilligung der Anwender deren Rechnerkapazität im Internet nutzen kann [BFJB01]. Sie lösten ein mathematisches Problem mit Hilfe von Web-Servern in Nordamerika, Europa und Asien, ohne dass die Betreiber dieser Sites etwas davon merkten. Dazu nutzten sie das Kommunikationsprotokoll TCP aus: Um die Integrität von Daten sicherzustellen, platziert der Sender eines Datenpakets eine Prüfsumme (Checksumme) über die im Datenpaket enthaltenen Datenbits in dessen Header. Auf der Empfängerseite wird diese Prüfsumme nachgerechnet und je nach Ergebnis wird das Datenpaket als korrekt akzeptiert, oder es wird verworfen. Durch geeignete Modifikation der Pakete und deren Header gelang es, einfache Berechnungen durchzuführen.

Diese Forschungsarbeit wurde in diversen Artikeln öffentlich publiziert und besprochen [Fre02], sie ist weitgehend auch aus technischer Sicht detailliert dokumentiert [oND02].

## 1.2 Inhalt der Arbeit

Im Unterschied zu der vorgestellten Methode, welche mit TCP Datenpaketen arbeitet, verwenden wir in dieser Arbeit ICMP Pakete. Dies, weil das ICMP-Protokoll erstens einen wesentlich geringeren Overhead mit sich bringt, ausserdem nicht von Protokollen höherer OSI-Layer abhängt und schliesslich gemäss [GF89] von jedem Host im Internet zur Verfügung gestellt werden muss.

Die mittels Modifikation von ICMP Datenpaketen verteilt durchführbaren Bit-Operation AND und XOR reichen aus, um jedes auf einem klassischen Computer lösbare Problem parasitär, sprich mit fremder Rechenkapazität, zu berechnen. Mit diesen beiden Operationen können zwei ganze Zahlen addiert werden, indem die zwei Summanden bitweise, beginnend mit dem niederwertisten Bit, zueinander addiert werden (XOR) und das Übertragungsbit (AND) jeweils in die nächste Bit-Addition übernommen wird.

Aufbauend auf dieser Addition von ganzen Zahlen mit Übertragungsbit wurde eine virtuelle Maschine nach dem Modell der Registermaschine entwickelt, welche eine Additions-Instruktion einer Assemblersprache verteilt parasitär berechnen kann. Diese Assemblersprache trägt den Namen 4-Instruction-Assembler (4IA) und besteht, wie ihr Name besagt, lediglich aus der Additions- und drei weiteren Instruktionen. Diese weiteren Instruktionen dienen dem Speicherzugriff auf die virtuellen Register (Wert setzen, Wert kopieren) beziehungsweise dem Stoppen der Maschine (Halt-Instruktion)

und benötigen somit keine Leistung des rechnenden (parasitären) Prozessors. Wie im Realisierungskonzept [RS02b] dieser Arbeit gezeigt wurde, reicht der Instruktionssatz der 4IA-Sprache aus, jedes auf einem klassischen Computer berechenbare Problem zu programmieren und lösen.

Zur Steuerung der virtuellen Maschine durch den Benutzer wurde das Software-Paket `pshell` entwickelt. Es bietet, von der Bedienung einer klassischen UNIX-Shell ähnlich, umfangreiche Parameterisierungs-, Auswertungs- und Kontrollmöglichkeiten, um in der 4IA-Sprache geschriebenen Programmcode parasitär auszuführen.

Eine höhere Assemblersprache mit wesentlich erweitertem Instruktionsumfang ist die Sprache XIA[1], welche mit dem im Rahmen dieser Arbeit entwickelten Cross-Compiler `xto4` in 4IA-Code übersetzt werden kann. Das Instruktionsset der XIA-Sprache ist einerseits darauf ausgelegt, die Entwicklung von nicht trivialen Programmen zu vereinfachen, andererseits um als mögliches Backend für Compiler von Sprachen der dritten Generation zu dienen.

Abbildung 1 zeigt den Aufbau und die Abhängigkeiten der verschiedenen Schichten. Die Schicht „3GL" mit der entsprechenden Abhängigkeit ist kursiv dargestellt, weil sie im Rahmen dieser Arbeit nicht implementiert wurde.

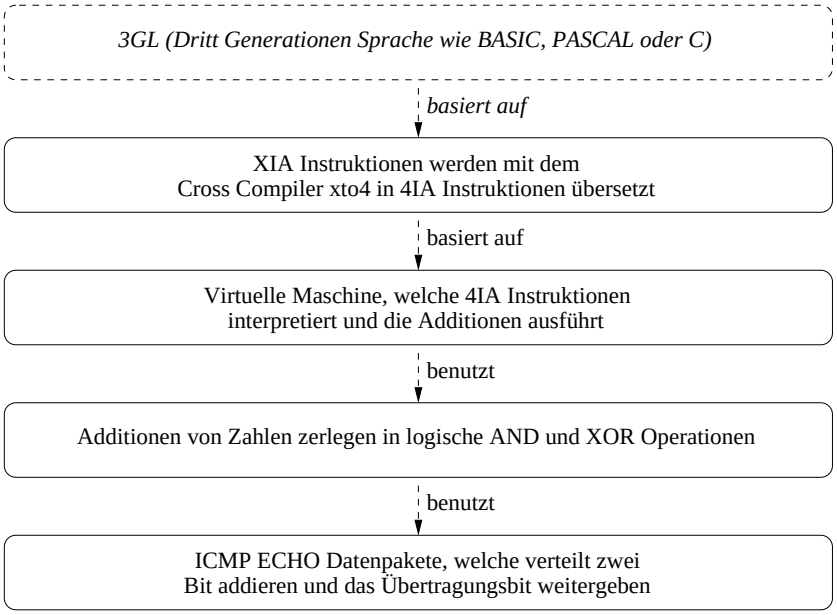


Abbildung 1: Prinzipieller Aufbau der zusammenwirkenden Schichten

[1] Extended-Instruction-Assembler: Instruktionsumfang bestehend aus den mathematischen Grundoperationen `ADD`, `SUB`, `MUL`, `DIV` und `MOD`, Vergleichsoperatoren wie `JG`, `JGE`, `JEQ`, `JNE`, etc., sowie den wichtigsten bitweisen Operationen `AND`, `OR`, `XOR`, `NOT`, `SHL`, `SHR`.

# 2 Projektverlauf

Das Thema dieser Dimplomarbeit wurde uns am 1. März 2002 zugeteilt. Am 26. April 2002 fand ein Kick-off Meeting statt, welches den Start des Projektes darstellte.

Die Diplomarbeit wurde unterteilt in fünf zentrale Phasen: Projekt-Setup, Realisierungskonzept, Implementation, Auswertung und Dokumentation. In den nachfolgenden Abschnitten werden die Arbeiten dieser Phasen kurz aufgezeigt.

## 2.1 Projekt-Setup

Die erste Phase „Projekt-Setup" diente dazu, die Infrastruktur wie beispielsweise ein Versionsverwaltungs-Repository, Mailinglisten oder die LaTeX Umgebung zur Dokumentation einzurichten. Ferner bildete das zusammentragen und studieren von themembezogenen Texten einen wesentlichen Teil dieses Projektabschnittes.

Während der Einarbeitung in die Unterlagen entwickelten wir einen ersten Prototypen, mit welchem wir die aus den vorhandenen Dokumentationen gewonnenen Ideen und Lösungsansätze nachvollziehen und ausbauen konnten. Die daraus gewonnenen Erkenntnisse waren ausschlaggebend für die weiteren Iterationen des Prototyps und schliesslich die endgültige Implementation.

## 2.2 Realisierungskonzept

Die Realisierungskonzept-Phase dauerte länger als angenommen, da aus der ursprünglichen Idee, lediglich Grundzüge der Implementation zu definieren, nach und nach ein Dokument entstand, welches einen konkreten Realisierungsweg mit hohem Detaillierungsgrad aufzeigte. Dies geschah einerseits Aufgrund der Erkenntnisse der fortschreitenden Prototypen, andererseits aber auch grossenteils auf rein theoretischer Grundlage.

Wärend sich das Realisierungskonzept praktisch ausschliesslich auf die virtuelle Maschine und die 4IA-Sprache bezog, wurde über den Cross-Compiler `xto4` zu diesem Zeitpunkt noch kaum diskutiert. Falls ein Compiler entwickelt werden sollte, welcher auf der 4IA-Sprache aufbaute, dann mit dem Ziel, durch einen mächtigeren Instruktionsumfang – vergleichbar etwa mit der in [Sch92] vorgestellten Sprache REGS – die Entwicklung nicht trivialer Programme zu erleichtern.

## 2.3 Implementation

Bereits zu Projektbeginn haben wir uns entschieden, die Software der virtuellen Maschine ausschliesslich für UNIX und Linux Umgebungen zu realisieren. Da die Anforderung bestand, direkt auf den Kommunikations-Stack des Betriebssystems zuzugreiffen, und da dies eine sehr systemspezifische Aufgabe ist, mussten wir uns auf *eine* Umgebung einschränken. Mit dieser Anforderung war auch die zur Realisierung benutzte Programmiersprache (C) definiert.

Als Programmiersprache zur Entwicklung des Cross-Compilers `xto4` entschieden wir uns für Java, dies einerseits um diesen Teil der Arbeit Plattformunabhängig zu halten und andererseits, um das technische Spektrum aus Eigeninteresse etwas zu erweitern.

Dabei haben wir auf weitere Abhängigkeiten zu Drittprodukten, wie etwa JavaCC, bewusst verzichtet, um die Software nicht zuletzt schlank und effizient zu halten.

In den realisierten Paketen legten wir grossen Wert darauf, die einzelnen Teile immer absolut modular und gekapselt zu gestalten. Dies bewahrte uns davor, zeitraubendes „Code Refactoring" zu betreiben oder Source-Code mehrfach neu zu schreiben. Der Code der Prototypen, welcher zu Beginn der Diplomarbeit geschrieben wurde, konnte grösstenteils wiederverwendet werden.

Die Implementationsphase war wesentlich weniger zeitintensiv als wir zu Beginn des Projektes angenommen hatten. Dies weil wir bereits zum Zeitpunkt der Realisierungskonzept-Erstellung die zu implementierenden Algorithmen und Module bestimmten und während der Implementierung keine nennenswerten Überraschungen auftauchten. Einzig die Entwicklung der Funktionalität, mehrstellige Bit-Additionen in einem Schritt berechnen zu können und somit eine Parallelisierung zu erreichen, wurde erst wärend der Implementationsphase diskutiert, entworfen und ausgeführt.

## 2.4 Auswertung

Auswertungen über die Effizient und Fehleranfälligkeit der Berechnungsmethode zu erstellen gestaltete sich aufgrund der implementierten Monitoring-Funktionalitäten als einfach und wenig zeitintensiv.

## 2.5 Dokumentation

Die Dokumentation wurde weitgehend parallel zur Entwicklung der Software laufend nachgeführt. Viel Zeit in Anspruch nahm das nach Abschluss der ersten Beta-Versionen erstellte Dokument „Handbuch", worin sich ausführliche Referenzen und Programmieranleitungen zu den beiden definierten Sprachen finden.

Alle Dokumente wurden mit dem Textsatzsystem LaTeX erstellt. Der Gedankte hinter diesem Entscheid war primär, uns mit dieser bis ahnhin nicht gut bekannten Dokumentations-Umgebung vertraut zu machen.

# 3 Ergebnisse

In den nachfolgenden Abschnitten wird aufgezeigt, wie der Terminplan eingehalten werden konnte, welche Ziele erreicht respektive nicht erreicht wurden und was wir zusätzlich implementierten.

## 3.1 Terminplan

Der Terminplan, welcher sich definierte durch die Meilensteine[2], erwies sich bereits kurz nach den Sommerferien als ungünstig. Durch die viel detailliertere Ausarbeitung des Realisierungskonzeptes als ursprünglich vorgesehen verschob sich die Abgabe dieses Dokumentes nach hinten, kurz vor Beginn der Herbstferien. Dafür dauerte die Implementationsphase, welche planungsgemäss zwei Wochen vor Projektabnahme abgeschlossen sein sollte, weniger lange als angenommen, so dass wir bedeutend mehr Zeit zur Verfügung hatten, die Dokumentationen Handbuch, Systemdesign, Auswertungen und Projektbericht zu erstellen und auszuarbeiten.

## 3.2 Ziele

Als Ziele wurden einzelne Module definiert, welche es zu erarbeiten galt. Tabelle 1 gibt Aufschluss darüber, welche Module mit welcher Priorität zur Realisierung definiert wurden.

| *Modul Name* | *Priorität* | *Realisierungsschritt* |
|---|---|---|
| Virtuelle Maschine, Speicherverwaltung, Codeausführung, Eingabe, Ausgabe | hoch | erste Phase |
| Paketbildung, ICMP-Simulation | hoch | erste Phase |
| 4IA-Microassembler | hoch | erste Phase |
| parasitäre ICMP-Berechnung, Fehlerbehandlung | mittel | zweite Phase |
| Hosts Priority-Queue, Statusanzeige | mittel | zweite Phase |
| Höhere Assemblersprache, Visualisierung, Hochsprache | gering | optionale dritte Phase |

Tabelle 1: Definierte Ziele des Pflichtenheftes

### 3.2.1 Erreichte Ziele

Sämtliche Ziele mit Priorität hoch und mittel sowie ausserdem zwei Ziele mit Priorität gering wurden erreicht.

### 3.2.2 Nicht erreichte Ziele

Das Modul „Hochsprache” mit Priorität gering wurde zu Gunsten eines zusätzlichen Moduls (siehe nachfolgendes Unterkapitel 3.2.3) nicht implementiert, da dieses für den eigentlichen Kern der Arbeit, die parasitäre Berechnung, wenig relevant ist.

[2] Vier Meilensteine wurden festgehalten: Abnahme Pflichtenheft, Abnahme Realisierungskonzept, Abschluss Implementation und Abnahme Projektbericht.

### 3.2.3 Zusätzliche Ziele

Zusätzlich zu den in Tabelle 1 genannten Zielen entstand inmitten der Diplomarbeit, aus Überlegungen der Effizienzsteigerung der parasitären Rechennutzung, ein weiteres Modul, welches zum Ziel hatte, mehrere bitweise Additionen pro Instruktion ausführen zu können. Dieses Modul wurde gegen Ende der Implementationsphase erfolgreich umgesetzt.

# 4 Ausblick

Wie bereits aus der Aufgabenstellung deutlich wird, ist das in dieser Arbeit zur Anwendung kommende parasitäre Prinzip nicht rentabel einsetzbar. Die virtuelle Maschine ist ungefähr um einen Faktor von 7'000 ineffizienter als der Prozessor eines durchschnittlichen Homecomputers (vgl. [RS02a]). Es ist aber durchaus denkbar, die Methode im Zusammenhang mit anderen Protokollen (etwa solcher kryptographischer Natur) so einzusetzen, dass die Effizienz der parasitären Berechnungen diejenige des lokalen Prozessors übersteigt.

Sollte eine solche Methode gefunden werden, so kann sie mit geringem Aufwand auch in die virtuelle Maschine dieser Diplomarbeit integriert werden; Es besteht diesbezüglich eine wohldefinierte, leicht integrierbare Schnittstelle.

# 5 Fazit

Durch die Diplomarbeit ,,Parasitic Computing" konnten wir aufzeigen, dass jedes auf einem klassischen Computer lösbare Problem parasitär, sprich mit fremder Kapazität, berechenbar ist. Die Nutzung dieser Rechenressourcen ist jedoch höchst ineffizient, denn ein Vielfaches an lokaler Rechenleistung ist notwendig, um lediglich das Resultat einer einfach Bit-Operation zu erhalten. Die Arbeit dient somit ausschliesslich als ,,Proof of Concept" der Aufgabenstellung.

# Literatur


[BFJB01] Albert-László Barabási, Vincent W. Freeh, Hawoong Jeong, and Jay B. Brockman. Parasitic computing. *Nature*, 412:894–897, August 2001.

[Fre02] Vincent W. Freeh. Anatomy of a parasitic computer. *Dr. Dobb's Journal*, 332:63–67, Januar 2002.

[GF89] Network Working Group and Internet Engineering Task Force. Rfc 1123, requirements for internet hosts. `http://www.faqs.org/rfcs/rfc1123.html`, Oktober 1989.

[oND02] University of Notre Dame. Parasitic computing website. `http://www.nd.edu/~parasite/`, Juni 2002.

[RS02a] Juerg Reusser and Luzian Scherrer. Parasitic computing: Auswertungen. `http://parasit.org/documentation`, Dezember 2002.

[RS02b] Juerg Reusser and Luzian Scherrer. Parasitic computing: Realisierungskonzept. `http://parasit.org/documentation`, Oktober 2002.

[Sch92] Franz Josef Schmitt. *Praxis des Compilerbaus*. Hanser Studienbuecher der Informatik, Muenchen, Wien, 1992.