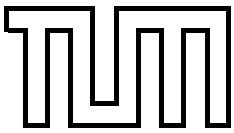
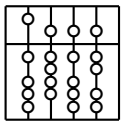


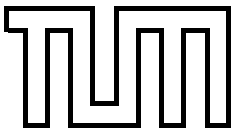
Technische Universität München
Fakultät für Informatik



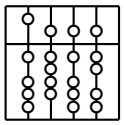
Diplomarbeit

Vervollständigung von Profile-Messungen durch Kombination

Dietrich Christopeit



Technische Universität München
Fakultät für Informatik



Diplomarbeit

Vervollständigung von Profile-Messungen durch Kombination

Dietrich Christopeit

Aufgabensteller:	Prof. Dr. Arndt Bode
Betreuer:	Dr. Josef Weidendorfer
Abgabedatum:	15. August 2004

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. August 2004 Dietrich Christopeit

Zusammenfassung

Bei der Analyse und Erstellung von Programmprofilen von Programmausführungen erlangen Profiling–Tools immer größere Bedeutung. Die Komplexität der modernen Hard– und Software macht die rein manuelle Analyse so gut wie unmöglich. Die modernen Rechensysteme unterstützen den Anwender bei dieser oft schwierigen Aufgabe durch die Bereitstellung von Ereigniszählern, sowohl in den Prozessoren, als auch in Datenstrukturen und Bibliotheken der Betriebssysteme. Die verfügbaren Profiling–Tools lassen sich für die Messung der breiten Vielfalt von Ereignissen bestimmter Ereignistypen konfigurieren und können die gemessenen Werte graphisch darstellen. Viel schneller erlangt der Benutzer so eine Vorstellung, auf welche Programmteile sich die verbrauchten Taktzyklen aufteilen. Zudem ist es möglich, eine reichhaltige Zahl von weiteren Ereignissen messen zu lassen. Solche Ereignisse sind zum Beispiel die Anzahl der fehlerhaft vorhergesagten Verzweigungen. Mit dem Wissen um die Bedeutung dieser gemessenen Werte lassen sich Schwachstellen und Geschwindigkeitsengpässe von Programmen schneller ermitteln, so daß die Lösung für solche Probleme in greifbare Nähe rückt. Die manuelle Analyse des Quellcodes ist bei dem Umfang heutiger Programme kaum erfolgversprechend. Der Erfolg hängt aber auch davon ab, in wie weit der Anwender sich im Klaren über die Zusammenhänge des Auftretens unterschiedlicher Ereignisse unterschiedlicher Ereignistypen ist. Ist diese Basis vorhanden, so kann die Kombination von Ereignistypen einen detaillierteren Einblick in das Verhalten eines Programms in Ausführung liefern, als dies bei der isolierten Betrachtung der aufgetretenen Ereignisse der Fall ist.

Vorwort

Über diese Arbeit Diese Arbeit stellt die Diplomarbeit für den Studiengang Informatik dar und wurde unter der Aufsicht von Dr. Josef Weidendorfer und Prof. Dr. Arndt Bode am Lehrstuhl X für Rechnertechnik und Rechnerorganisation/Parallelrechnerarchitektur der Technischen Universität München verfasst.

Die Arbeit befasst sich mit der Vervollständigung von Profile–Messungen durch die Kombination von Ereignistypen der Hard– und Software. Das Ziel der Arbeit ist die Aufstellung einer allgemeinen Klassifizierung von Ereignistypen, anhand derer die Kombination der Typen verdeutlicht wird. Die vorgeschlagene Klassifizierung wird auf real existente Prozessoren und Betriebssysteme abgebildet, um den angestrebten Überblick am Beispiel zu zeigen. Des weiteren steht die Entwicklung einer abstrakten Spezifikationssprache für die Kombination der Ereignistypen im Blickpunkt dieser Arbeit. Mit Hilfe dieser Sprache soll dem Anwender die Erstellung von Metriken und Heuristiken erleichtert werden. Schließlich wird die Ereignistypkombination in der Praxis angewandt und eine weitere Heuristik für die Berechnung von Inklusiv–/Exklusivkosten von Prozeduraufrufen vorgeschlagen und als Programmmodul für das Visualisierungsprogramm KCachegrind umgesetzt.

Danksagung Vor Allem danke ich meinem Professor Prof. Dr. Arndt Bode und meinem Betreuer Dr. Josef Weidendorfer für die Begleitung während der Erstellung dieser Diplomarbeit. Sie machten den Zugang zu Informationen und deren Aufbereitung möglich. Zu buchstäblich jeder Zeit wurde mir bei Problemen geholfen und geduldig jede, auch wiederkehrende Frage beantwortet. Darüber hinaus danke ich meiner Mutter Gabriele von Witzleben–Christopeit und meinem Vater Prof. Dr. Joachim Christopeit für die stete Sorge und geistige Unterstützung meiner Person während dieser Zeit. Nicht nur wurde darauf geachtet, daß alle anders gelagerten Verpflichtungen von mir ferngehalten wurden, auch kam zu jeder Zeit die nötige Aufmunterung. Meinen Kommilitonen und Freunden Alexander Bornschlegl, Evi Groher, Henrik Schmidt, Martin Groher und Stephanie Biller möchte ich im Speziellen einerseits für die fachlich hilfreichen Diskussionen und andererseits für den treuen seelischen Beistand danken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Der Grad der Komplexität von Mikroprozessoren folgt seit 1965 der Regel von Moore und Noyce, die besagt, dass sich die Anzahl der auf einem integrierten Halbleiterbaustein befindlichen Transistorfunktionen etwa alle 18 bis 24 Monate verdoppelt [11]. Hatte der Intel Prozessor 8080 des Jahres 1972 noch 3500 Transistoren, so waren es 1974 schon 6000 Transistoren und mit der Einführung des Pentium Prozessors im Jahre 1993 sogar schon 3,1 Million. Über diese Zeit hinweg wurde immer wieder prognostiziert, daß die Entwicklung der Mooreschen Vorhersage nicht mehr folgen könne, da technische Grenzen erreicht seien; doch Antworten zum Beispiel auf Probleme bezüglich der Isolation zwischen den Leiterbahnen, Kühlung der Prozessoren, oder der Genauigkeit der Laser, die zum Ätzen der Schaltkreise verwendet werden, wurden immer gefunden. Viele Gründe geben also den Anlass, zu vermuten, daß Moores Gesetz nicht mehr für lange Zeit seine Gültigkeit behalten können wird, Moore selbst zweifelt dies an. Doch der technologische Fortschritt scheint nicht zu bremsen zu sein und so werden wohl am ehesten kühle wirtschaftliche Argumente den Trend ändern [25].

Der steigende Integrationsgrad der Prozessoren wurde auf der Seite der Software von immer größer werdenden Programmen begleitet. Das Betriebssystem Windows 3.1 aus dem Jahr 1990 der Firma Microsoft hatte drei Millionen Source Lines of Code (SLOC), Windows NT des Jahres 1995 benötigte vier Millionen SLOC und Windows 95 bestand mit der Einführung 1997 aus 15 Millionen SLOC, also mehr doppelt so viel wie Windows 3.1 und NT aus den Jahren zuvor zusammen. Debian GNU/Linux 2.2 vom August 2000 bestand sogar aus 55 Millionen SLOC, hätte 14005 Personenjahre für die Entwicklung benötigt und ein Budget 1,9 Milliarden US–Dollar verbraucht [37].

Angesichts der steigenden Komplexität auf den beiden Seiten der elektronischen Informationsverarbeitung ist die Geschwindigkeit eines Programms ein zentrales Thema. Immer größere Programme scheinen immer schnellere Prozessoren zu brauchen, um mit diesen immer komplexere Probleme in immer kürzerer Zeit lösen zu können. Auch wenn Programme für beispielsweise die Wettervorhersage nicht derartig viele Codezeilen haben, so sind die Verursacher für Geschwindigkeitsprobleme schwer im Code zu lokalisieren; und die Performance eines Programms ist mit das wichtigste Verkaufsargument.

Dieser Tatsache sind sich auch die Prozessorhersteller bewusst geworden und haben zum Beispiel mit der Einführung des Pentium dem Programmierer viele Möglichkeiten an die Hand gegeben, um Performance–Probleme im Programm aufspüren zu können. So finden sich dort Zähler, die zum Beispiel die Anzahl der L1–Cache–Misses und das Auftreten ähnlicher Ereignisse anzeigen. Solche Zähler können so programmiert werden, daß mit Erreichen eines bestimmten Wertes ein Interrupt ausgelöst wird, welcher durch Software bearbeitet werden kann. So werden statistische Erhebungen über das Ablaufverhalten von Programmen und die Erstellung eines Ablaufprofils möglich gemacht. Zusätzlich im Source–Code an geeigneten Stellen eingefügte Codefragmente, die Aufschluß über zum Beispiel das Auf-

rufverhalten von Prozeduren liefern, helfen das Geschwindigkeitsproblem bis zur Lösung immer weiter einzugrenzen. Dieses, als Instrumentierung bezeichnete Verfahren setzt aber, im Gegensatz zur Ereignismessung im Prozessor, die Verfügbarkeit des Quellcodes voraus. Sind die Daten aus Hard– und Software gesammelt, so geben Visualisierungsprogramme wie KCachegrind [1] oder VTune [24] in Histogrammen und Kurven ein Bild vom Verhalten des ablaufenden Programms über seine Ausführungszeit hinweg.

1.1 Aufgabenstellung

Wie oben angesprochen, ist es auf Grund der hohen Komplexität heutiger Prozessoren und Programme schwierig, Software zu beschleunigen. Zwar würden schnellere Prozessoren für gewisse Zeit langsamen Programmen auf die Sprünge helfen, doch kann die stetige Anschaffung modernerer Hardware nicht die Lösung sein. Viele Punkte gibt es, an denen der Programmierer ansetzen kann, ein Performance–Problem zu lösen. Von schnelleren Algorithmen, bis hin zu einer optimierten Speicherverwaltung führt der Weg, um dem Anwender Zeit zu ersparen. Dabei wird der Benutzer und der Entwickler durch eine Vielzahl von Werkzeugen unterstützt, die Auskunft über das Laufzeitverhalten geben [33].

Die Aufgabenstellung beinhaltet die Erstellung einer Klassifizierung von Ereignistypen, die für die Performance–Analyse von Belang ist. Die erste Unterscheidung wird zwischen Hardware und Software getroffen.

Im Hinblick auf die Hardware wird untersucht, welche Ereignisse im Prozessor und in der Peripherie auftreten und gemessen werden können. Eine Klassifizierung der Typen dieser Ereignisse gibt dann einen geordneten Gesamtüberblick, der auch die Beziehungen zwischen einzelnen Ereignistypklassen verdeutlicht. Angestrebt ist ein einheitliches Modell, welches unabhängig von der Architektur, die Vielzahl der verschiedenen Ereignistypen zusammenfasst, die in den Referenzhandbüchern der Prozessorhersteller nur schwer und meist ungeordnet zu finden sind. Das vorgeschlagene Modell wird anhand konkreter Architekturen auf Praktikabilität untersucht, nämlich auf die CISC–Architektur IA–32 P6 [3], die 64 Bit VLIW/EPIC Architektur IA–64 Itanium [6] der Firma Intel, sowie auf die RISC–Architektur PowerPC 604e [9] der Firma IBM abgebildet. Die Prozessoren dieser Hersteller werden deshalb als Beispiel herangezogen, da die Referenzhandbücher sich als detaillierte Beschreibung der Architekturen herausstellten.

Den Abstraktionsstufen von der physikalischen Maschine aus nach oben folgend, steht das Betriebssystem im Weiteren im Mittelpunkt. Auch hier soll eine Aufstellung und anschließende Klassifizierung der relevanten Ereignistypen erfolgen, die ein Betriebssystem zur Verfügung stellen sollte. Untersucht wird am allgemeinen Aufbau eines Betriebssystems Prozess– und Thread–Management, die Speicherverwaltung, die Ein–/Ausgabe–Verwaltung, das Dateisystem und das Netzwerksubsystem [20]. Die Implementierung des gefundenen Klassenmodells wird an den Betriebssystemen Windows 2000 der Firma Microsoft und Linux aufgezeigt. Das Augenmerk soll auch hierbei auf die Kombinationsmöglichkeiten der einzelnen Klassen und deren Ereignistypen fallen, derart daß ein neuer Ereignistyp aus der Synthese zweier oder mehrerer verschiedener Ereignistypen entsteht und dem Anwender mehr Information liefern kann.

Damit der Anwender die Kombination von Ereignistypen nicht als eigenständige, fest

programmierte Algorithmen in Softwaremodulen umsetzen muß, wird eine allgemeine Spezifikationssprache vorgeschlagen. Die Definition eines Ereignistyps, sowie die Kombination dieser Typen soll so vereinfacht werden.

Liegen von einem Programm verschiedene Messungen vor, so ist es möglich, durch die Kombination der verschiedenen Daten das Ablaufprofil eines Programms zu vervollständigen. Dieser Aufgabe widmet sich der praktische Teil der Arbeit. Die Aufgabe besteht darin, die Gesamtheit der Kosten, die ein Programm während der Ausführung verursachte, auf die einzelnen Prozeduren im Programm zu verteilen. Kosten bezeichnen hier zum Beispiel die verbrauchten Zyklen, die Anzahl abgeschlossener Instruktionen, aber auch die Anzahl der erfolglosen Zugriffe auf die Cache–Hierarchie. Von einem Beispielprogramm ist nun einerseits das Ergebnis einer Simulation vorhanden, andererseits die Daten einer Sampling–Messung. Mit Simulationsdaten ist es möglich, den Kostenverbrauch exakt auf die Prozeduren zu verteilen. Sowohl die Kosten, die eine Prozedur selbst ohne ihre Aufrufe anderer Prozeduren verursacht[1], als auch die akkumulierten Kosten, die zusätzlich entstehen, wenn eine Prozedur wiederum andere aufruft[2], können in einer Simulation gemessen werden. Mit den Daten einer Sampling–Messung ist dies nicht möglich – allein die Selbstkosten können gemessen werden. Durch die Kombination der Information aus beiden Messungen, kann jedoch auf die Inklusivkosten geschlossen und damit die Sampling–Messung vervollständigt werden. Die von Profiling–Tools, wie `gprof` verwendete Heuristik [2] soll eine Verbesserung derart erfahren, daß die Verteilung der Kosten auf Prozeduraufrufe für bestimmte Situationen exakter wird und der genauen Berechnung, wie es in Simulationen möglich ist, näher kommt. Die gefundene Heuristik wird als Modul für ein Visualisierungsprogramm implementiert. Ein Gütemaß soll dann die Qualität der Heuristik bestätigen.

1.2 Relevante Arbeiten

Die grundlegenden Definitionen, sowie die Implementierung von Ereignissen und deren Erfassung in modernen Prozessoren finden sich in den Referenzhandbüchern der jeweiligen Hersteller [3, 6, 5, 9]. Diese Arbeiten erklären die unterschiedlichen Methoden der Zählung von Ereignissen, wie PEBS und Time–Based–Event–Counting. Es werden auch Verfahren angegeben, wie zum Beispiel das Auftreten von Ereignissen auf Quellcodeadressen abzubilden ist. Die Dokumentation der Bibliothek PAPI [8] erklärte die Kombination verschiedener Ereignistypen bei den betrachteten Prozessoren an den Stellen, an denen diese Kombinationen nicht in den technischen Unterlagen der Hersteller zu finden waren. Die Komponenten moderner Prozessorarchitekturen, wie zum Beispiel der TLB und die ALAT werden in den Arbeiten [13, 27, 38] vorgestellt und erklärt. Die Erkenntnisse aus diesen Arbeiten sind in die Klassifizierung eingeflossen. Bei der Klassifizierung von Ereignistypen der Peripherie gaben die technischen Handbücher und Arbeiten [22, 31, 23, 30, 28, 18] Hinweise ob vorgeschlagene Ereignistypen von Nutzen sind und ob diese bei den verschiedenen Herstellern der Peripheriegeräte umgesetzt wurden. Performance–Verluste durch sinkenden ILP behandelt die Arbeit von Gabbay und Mendelson [21]. Der Artikel von Choi [16] untersucht adaptive Mechanismen zur Lastverteilung auf verteilten Server–Cluster Systemen. In diesem Artikel

[1] Dies sind die Selbst– oder Exklusivkosten.
[2] Dies sind die Inklusivkosten.

werden die dafür nötigen Performance–Zähler angegeben. Die dort aufgeführten Ereignistypen und die Kombination dieser wurde in die Klassifizierung aufgenommen.

Auf der Seite der Software halfen die technischen Unterlagen [14, 17, 29, 7]der Betriebssysteme Windows 2000 und Linux bei der Erstellung der Klassifizierung. Aus [20] wurde die grundlegende Klassenhierarchie für die Ereignistypklassen der Software abgelesen.

Die Entwicklung der Spezifikationssprache wurde inspiriert von den Arbeiten [39, 15]. Die dort verwendeten Verfahren und Vorgehensweisen halfen bei der Spezifikation der Sprache.

1.3 Aufbau der Arbeit

Den vorangegangenen Überlegungen folgend, ergibt sich der Aufbau der Arbeit. Das Kapitel 2 ist den Definitionen der Begriffe Ereignis und Ereignistyp gewidmet. Es werden die verschiedenen Verfahren des Profilings erklärt und in Zusammenhang gebracht. Dieses Kapitel stellt die Basis für das folgende Kapitel 3 dar. In Kapitel 3 wird die Klassifizierung der Ereignistypen eines Rechensystems vorgeschlagen. Dazu beginnen die Abschnitte 3.1.1, 3.1.2 und 3.1.3 mit der Klassifizierung von Hardwareereignistypen für Prozessor, prozessorexternen Speicher und übrige Peripherie. Der Abschnitt 3.2 wendet sich dann der Software und der Klassifizierung von Ereignistypen in einem Betriebssystem zu. In den Abschnitten 3.2.1 bis 3.2.4 werden die Ereignistypen der Komponenten eines Betriebssystems einzeln beleuchtet und klassifiziert. Nachdem die Klassifizierung erfolgt ist, wird in Kapitel 4 auf die tatsächliche Umsetzung in Hard– und Software eingegangen. Das Kapitel spricht die Ereigniszähler der Prozessoren, der Peripherie und der Software an und stellt die Besonderheiten und Unterschiede der verschiedenen Systeme vor. Die Ereignistypklassen und ihre einzelnen Mitglieder werden nochmals aufgegriffen und untersucht, in wie fern sich diese bei den unterschiedlichen Produkten finden lassen.

Das Kapitel 5 beschäftigt sich mit der Entwicklung einer Spezifikationssprache, die es dem Anwender erleichtern soll, Ereignistypen und deren Kombination auf abstrakte Weise spezifizieren zu können. Dieses Kapitel stellt zunächst in Abschnitt 5.1.1 den Begriff Ereignistyp als Menge dar. In Abschnitt 5.2 wird die Grammatik der Spezifikationssprache angegeben und mit Hilfe der Mengendarstellung von Ereignistypen erläutert. Der letzte Abschnitt des Kapitels gibt einige einfache Beispiele in der Sprache an.

Den praktischen Teil dieser Arbeit fasst das Kapitel 6 zusammen. Es wird die Heuristik von `gprof` für die Berechnung von Inklusivkosten und die gefundene Quotientenheuristik erklärt. Die Ergebnisse dieser Messungen, einer Simulation und der realen Daten werden miteinander verglichen.

2 Performance–Messung – Grundbegriffe und Abgrenzung

Dieses Kapitel befasst sich mit der Klassifizierung von Ereignistypen der Hard– und Software. Bevor diese Einteilung vorgeschlagen wird, müssen die notwendigen Grundbegriffe vorgestellt und erklärt werden.

Moderne Hardwarearchitekturen enthalten Mechanismen, die es Profiling–Werkzeugen aktiv und direkt gestatten, geschwindigkeitskritische Komponenten zu beobachten. Darunter fallen unter anderem BPU*s*, Caches und Einheiten für die Übersetzung von virtuellen Speicheradressen. Die diesbezügliche Analyse von Anwendungen und Betriebssystemen ist wichtig für Compileroptimierungsstrategien und die effiziente Nutzung verschiedener architekturabhängiger Merkmale, wie der spekulativen Ausführung (vergleiche [5]).

2.1 Der Ereignistyp und das Ereignis

Wie schon in der Einleitung angesprochen, quantifizieren Ereignisse in einer Rechenanlage Vorgänge, die im Verlauf des Betriebs auftreten. Der Ereignistyp beschreibt die Bedingungen für das Auftreten und die gemessenen Attribute, die mit dem Auftreten eines Ereignisses eines Typs verbunden sind. Das Ereignis ist also quasi die Instanz eines Ereignistyps. Die Attribute eines Ereignisses sind zum Beispiel die Quellcodeadresse beim Auftreten, oder die Prozeß–ID des Prozesses, in dem das Ereignis auftrat. Während der Ausführungszeit eines Programms ändert sich der Zustand der Maschine. Wichtige Zustandsänderungen können klassifiziert und den Funktionseinheiten des Prozessors, respektive den Subsystemen des Betriebssystems zugeschrieben werden. Ein Performance–Zähler summiert das Auftreten von Ereignissen über einen bestimmten Zeitraum hinweg. Diese Zähler können durch Software gelesen und im Anschluß daran bildlich dargestellt werden.

2.1.1 Ereignisarten

Es ist wichtig zu unterscheiden, wie oft ein Ereignis pro Zeiteinheit auftreten kann. Die folgende Aufzählung gibt einen Überblick:

- **Einzelereignisse (Single–Occurrence–Events)**: Der betreffende Ereigniszähler wird um höchstens 1 pro Taktzyklus erhöht. Das heißt, pro Takt kann das Ereignis einmal oder nicht auftreten, nicht aber mehrmals. Beispiele für solche Ereignisse sind bezüglich der Hardware TLB–Misses oder falsch vorhergesagte Verzweigungen.

- **Duration–Counts**: Verwandt mit den Einzelereignissen sind die so genannten Duration–Counts. Hierbei wird ein Zähler pro Taktzyklus solange erhöht, wie eine bestimmte Bedingung erfüllt ist, ein Ereignis andauert. Nach [5] wird diese Zählweise zu den Einzelereignissen gezählt. Die Methode kommt zum Beispiel bei der Messung von Pipeline–Stalls zur Anwendung, wenn bestimmt werden soll, wie lange eine Pipeline angehalten werden mußte.

- **multiple Ereignisse (Multi–Occurence–Events)**: Ereignisse, die aufgrund des Hardware–Parallelismus oder der parallelen Ablaufströmen eines Programms entstehen, werden Multi–Occurrence–Events genannt. Pro Taktzyklus der Hardware kann ein Ereignis mehr als einmal auftreten. Beispiele für solche Ereignisse finden sich bei [6]. Dort wird unter anderem das Ereignis, welches die abgeschlossenen Instruktionen zählt genannt.

- **Thresholding**: Um beispielsweise die Frage, über wie viele Taktzyklen hinweg mehr als eine bestimmte Anzahl von Instruktionen abgeschlossen wurden, zu beantworten, kann ein Schwellwert gesetzt werden. Dieser bestimmt die Erhöhung eines Zählers derart, daß der betreffende Zähler nur dann erhöht wird, wenn der Schwellenwert überschritten ist. Thresholding und Duration–Counts bezeichnen dasselbe – Duration–Counts jedoch für die Einzelereignisse und Thresholding für Ereignisse die mehrfach pro Taktzyklus auftreten können.

2.2 Ereignisraten

Wenn die Performance eines Rechensystems untersucht wird, ist man nicht nur an der Häufigkeit des Auftretens eines Ereignisses zu einem bestimmten Zeitpunkt interessiert, sondern auch an der zeitlichen Veränderung des Vorkommens. Ereignisraten ermöglichen es, die Werte der Ereigniszähler über die Zeit der Ausführung eines Programms darzustellen. Zwei fundamentale Maße sind von Interesse: Ereignisraten (Event–Rate–Monitoring) und die Aufteilung von Programm–Zyklen zu den Ereignissen (Cycle–Accounting).

- **Event–Rate–Monitoring**: Wichtige Ereignisraten während der gesamten Laufzeit eines Programms umfassen beispielsweise die durchschnittliche Anzahl der ausgeführten Instruktionen pro Taktzyklus, die Rate der erfolglosen Zugriffe auf Daten– und Instruktions–Caches oder die Rate der falsch vorhergesagten Verzweigungen. Auch verlangt die Charakterisierung von Betriebssystemen oder großen kommerziellen Anwendungen (z.B. OLTP–Analyse) eine systemweite Sicht auf relevante Ereignisse wie TLB–Miss–Raten, Anzahl der Interrupts pro Sekunde oder die Busauslastung. Der Abschnitt 2.2.1 beschäftigt sich mit diesem Thema.

- **Cycle–Accounting**: Jeder Zyklus, den ein Programm während seiner Laufzeit verbraucht, kann bestimmten Ereignissen zugeschrieben werden. Nebst der dem Programm inhärenten Wartezeiten, werden zusätzlich Zyklen verbraucht die von Pipeline–Stalls oder Ähnlichem kommen. In Abschnitt 2.2.2 wird dieses Thema besprochen.

2.2.1 Event–Rate–Monitoring

Das Event–Rate–Monitoring bestimmt Ereignisraten durch das Lesen von Ereigniszählern vor und nach der Bearbeitung einer Arbeitslast, geteilt durch die verstrichene Zeit. Man stelle sich beispielsweise zwei Hardwarezähler vor. Der eine (`INST_RETIRED`) zähle die Anzahl der ausgeführten Instruktionen und der andere (`CPU_CYCLES`) die Zahl der verstrichenen Zyklen. Der IPC einer gegebenen Arbeitslast kann dann wie folgt berechnet werden:

$$IPC = \frac{(INST\_RETIRED_{t1} - INST\_RETIRED_{t0})}{(CPU\_CYCLES_{t1} - CPU\_CYCLES_{t0})}$$

2.2.2 Cycle–Accounting

Das Beobachten von Ereignisraten beschränkt sich im Wesentlichen auf das Zählen von Ereignissen mit Rücksicht auf die verstrichene Zeit. Ist ein Performance–Problem vorhanden, so ist aber nicht geklärt, inwiefern die beobachteten Ereignisse dazu beitragen. Üblicherweise werden dann die Raten der verschiedenen beobachteten Ereignisse graphisch dargestellt und in Zusammenhang mit dem IPC gesetzt. Stellt sich beispielsweise heraus, daß eine niedrige Rate abgeschlossener Instruktionen mit einem Anstieg erfolgloser Cache–Zugriffe korreliert, kann die Hypothese aufgestellt werden, daß die Cache–Zugriffe ursächlich für das Problem sind. Solche Vermutungen sind nur mit viel Erfahrung aufzustellen und zu verifizieren. Manche Hardwarearchitekturen unterstützen die Programmierer hierbei in der Weise, daß für einige Ereignisse Cycle–Accounting–Monitore bereitgestellt werden. Es besteht so die Möglichkeit jeden Taktzyklus eines Programms einem bestimmten Typ von Ereignis zuzuordnen. Da die meisten Taktzyklen durch Stall– oder Flush–Bedingungen verschiedener Pipelines verbraucht werden und dieser Verbrauch nur schwer den Verursachern zuzuordnen ist, liegt das Augenmerk der Unterstützung auf den Stalls/Flushes der Pipelines, unter Anderem für:

- Branch Prediction
- Memory Pipelines, Daten–TLB– und andere Load–Stalls
- Scoreboard– und FPU–Stalls
- Instruction–Issue–Stops[1]
- Instruction–Fetch–Stalls[2] auf Grund von Instruction–Cache– oder TLB–Misses

Die Implementierung des Cycle–Accounting wird für die untersuchten Architekturen in Kapitel 4 angesprochen. Es ist zu beachten, daß Cycle–Accounting–Mechanismen sich vom Duration–Counting der Stalls und Flushes unterscheiden. Während beim Cycle–Accounting die verbrauchten Taktzyklen bestimmten Verursachern zugerechnet werden können, beschränkt sich das Duration–Counting auf die Gesamtheit der durch Stalls und Flushes verursachten Latenzzeiten.

[1] Situation, bei der die Verteilung von Instruktionen auf die Einheiten der CPU angehalten werden muß. Dies kann zum Beispiel bei Datenabhängigkeiten der Fall sein.

[2] Situation, wenn eine Instruktion nicht sofort verfügbar ist und Wartezyklen entstehen, weil die Instruktion von einem langsameren Speicher geladen werden muß.

2.3 Statistisches Sampling

Beim Sampling wird in bestimmten Abständen eine Messung vorgenommen. Je nach Modell sind die Abstände verschieden. In bestimmten Intervallen werden also Stichproben genommen und der Codezeile zugeordnet.

Beim Begriff statistisches Sampling kann wie folgt unterschieden werden:

- **Time–Based–Sampling**: Die Abstände, in denen Messungen vorgenommen werden, sind durch zeitliche Intervalle definiert. Diese Intervalle können einerseits fest sein andererseits einen variablen und eventuell zufälligen Anteil haben. In festen Zeitabständen die Messung vorzunehmen hat den Nachteil, daß bei Programmstrukturen wie Schleifen die Messung häufig an der selben Stelle im Code stattfindet. Damit die gemessenen Ereignisse nicht stets der selben Codezeile zugeordnet werden, kann das Intervall zufällig gewählt werden. Meist geschieht dies durch Vergrößerung, beziehungsweise Verkleinerung des Intervalls um eine zufällige Größe. Der Unterabschnitt 2.3.1 befasst sich mit diesem Thema.
- **Event–Based–Sampling**: Die Messungen werden nach einer definierten Anzahl aufgetretener Ereignisse vorgenommen. Eine zufällige Änderung des Schwellenwertes, ähnlich einem variablen Zeitintervall, ist ebenso vorstellbar und ermöglicht wiederum eine bessere Verteilung der Stichproben über den Code. In Unterabschnitt 2.3.2 wird auf dieses Thema eingegangen.

2.3.1 Time–Based–Sampling

Die Meßmethoden verfälschen das Ergebnis, da die Messung selbst Zeit in Anspruch nimmt. Um den Fehler so gering wie möglich zu halten, benutzen viele Performance–Analyse–Tools für die Messungen das so genannte Time–Based–Sampling (siehe [2, 24]). In regelmäßigen Zeitabständen (zum Beispiel 1ms) sammelt das Tool zum Beispiel die Instruktionsadressen. Die Verfälschung der Messung kann so beim Sampling durch die Wahl des Intervalls eingestellt werden. Nach Beendigung des Programmlaufs werden diese Daten den Modulen etc. zugeordnet, so daß ein Bild davon entsteht, welche Teile des Programms am meisten Zeit verbrauchen und welche Codeadressen am häufigsten angesprungen werden. Dem Programmierer zeigt eine graphische Darstellung, wo im Programm die so genannten Hot–Spots zu finden sind. Vorteil dieser Methode ist, daß die Daten nicht für jede ausgeführte Codezeile aufgezeichnet werden und somit die Anzahl der damit verbundenen (Fest–)-Speicherzugriffe deutlich vermindert und der Overhead, den die Sammlung von Stichproben mit sich bringt verringert wird. Das Verfahren wird durch die Annahme gerechtfertigt, daß sich die Verteilung der gemessenen Werte nur unwesentlich im Vergleich zu einer exakten Messung ändert.

2.3.2 Event–Based–Sampling

Eine andere Herangehensweise stellt das Event–Based–Sampling dar. Hier wird zum Beispiel die aktuelle Codeadresse aufgezeichnet, nachdem eine bestimmte Anzahl von

Prozessor– oder Softwareevents erreicht wurde. Gleichermaßen wie in 2.3.1 ist nach Ablauf des Programms dieser Wert den Modulen zuzuordnen. Die modernen Prozessorarchitekturen unterstützen das Event–Based–Sampling bereits durch die Hardware. In Kapitel 4, Abschnitt 4.1.1.1 werden die konkreten Realisierungen angesprochen.

2.3.3 Präzises attributieren der Samples

Die in den Abschnitten 2.3.1 und 2.3.2 genannten Verfahren haben zur Folge, daß nach bestimmten Zeitintervallen oder einer festgelegten Anzahl von Ereignissen detaillierte Daten über den Zustand der Maschine gesammelt werden können. Diese Vorgehensweise wird hier als präzises Sampling bezeichnet. Für die in Kapitel 3 angestrebte Klassifizierung muß daher eine Datenstruktur gefunden werden, die diese Informationen aufnimmt. Eine detaillierte Datensammlung verbraucht natürlicherweise Zeit, welche das Meßergebnis beinflußt (vergleiche Abschnitt 2.3.1). Zur Unterscheidung von präzisem Sampling und einfachem Sampling spricht man Falle des Time–Based–Sampling vom **P**recise–**T**ime–**B**ased–**S**ampling (**PTBS**), im Falle des Event–Based–Sampling vom **P**recise–**E**vent–**B**ased–**S**ampling (**PEBS**). In Kapitel 4 werden die konkreten Möglichkeiten der Realisierung bei den verschiedenen Architekturen angesprochen. Ist dieser Detailgrad nicht vonnöten, so kann nur der Instruktionszeiger zusammen mit den gemessenen Ereignissen protokolliert werden. Folgende Punkte sind zu betrachten, wenn der Zustand des Rechners zum Zeitpunkt der Messung genau beschrieben werden soll, um eine Rückverfolgung in den Code zu ermöglichen:

- **Die Registerinhalte**: Die Registerinhalte sind für das präzise Sampling von geringerer Bedeutung. Für das Debuggen von Programmen bekommen die Registerinhalte mehr Bedeutung.
- **Die Statusflags**: Die Betrachtung des Prozessorstatusworts gibt Aufschluß über die Bedingungen von Vergleichen und Ähnlichem zum Zeitpunkt des Auftretens des Ereignisses.
- **Der Instruktionszeiger**: Die Adresse, die der Instruktionszeiger enthält, macht die Rückverfolgung in den Quellcode erst möglich.
- **Der Branch–Trace–Buffer**: Im Branch–Trace–Buffer steht die Adresse der Instruktion von der aus die letzte Verzweigung ausging, sowie die Adresse der ersten Instruktion des Verzweigungsziels. Außerdem enthält der Puffer noch ein Bit, welches anzeigt ob diese Verzweigung korrekt vorhergesagt wurde. Die Analyse dieser Struktur zeigt welche Sprünge im Programm durch falsche Vorhersage Performance–Probleme verursachen, so daß Lösungen gezielt gesucht werden können.

Ein Record–Eintrag der Datenstruktur ist in Abbildung 2.1 dargestellt:

Diese Punkte stellen die gemeinsamen Attribute aller Ereignistypen dar. Für bestimmte Klassen von Ereignistypen werden in Kapitel 3 zusätzliche Attribute vorgeschlagen.

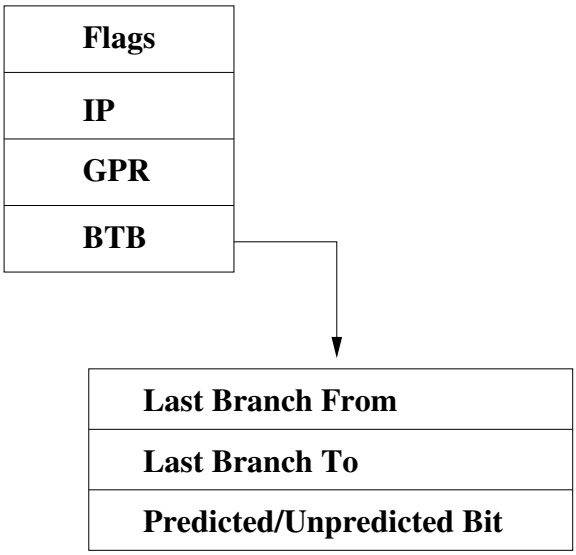


Abbildung 2.1: Die Recordstruktur für präzises Sampling.

2.4 Codeinstrumentierung

Bei der Codeinstrumentierung wird zusätzlicher Code in den Quelltext eines Programms eingefügt. Solche zusätzlichen Codeteile protokollieren während der Ausführung des Programms zum Beispiel die Anzahl der Funktionsaufrufe, welche Funktion von welcher Stelle aufgerufen wurde, oder wie viel Zeit in den einzelnen Programmabschnitten verbraucht wurde. Der Nachteil besteht im Gegensatz zum statistischen Sampling darin, daß der Quelltext des zu untersuchenden Programms verändert werden muß. Auch verfälscht die Instrumentierung das Ergebnis der Analyse, da der Code der Instrumentierung ebenfalls ausgeführt wird. Der Vorteil liegt in der Tatsache, daß durch die exakte Messung ein exaktes Profil des Programms erstellt werden kann. Die Messung der Inklusiv–/Exklusivkosten von Funktionen (vergleiche Kapitel 6) stellt mit Codeinstrumentierung kein Problem dar. Die Instrumentierung von Programmen kann auf den unterschiedlichen Ebenen der Quelltexttransformation stattfinden. Es existieren Programme, die den Code zur Laufzeit verändern, aber auch das manuelle Hinzufügen auf Hochsprachenebene ist möglich. Ein Beispiele für die Instrumentierung zur Laufzeit ist PureCoverage der Firma Rational [19] für die Instrumentierung von Java–Code.

2.5 Profiling

Das Thema des Profiling befasst sich mit der Sammlung von Daten über den Ablauf eines Programms. Für den Anwender steht dabei die Rückverfolgbarkeit in den Quellcode im Vordergrund. Das Profiling verwendet die oben vorgestellten Verfahren und erstellt ein Profil der laufenden Anwendung, um unter Anderem Antworten auf folgende Fragen (vergleiche [36]) zu finden:

- Wie viel Zeit nahm die Ausführung einer Funktion in Anspruch?
- Wie oft wurde eine bestimmte Funktion aufgerufen?
- Welche Zeilen des Sourcecodes benötigten die meiste Zeit zur Ausführung?

Diese Informationen zeigen, welche Programmteile langsamer ablaufen als erwartet und wo Veränderungen und Optimierungen am ehesten Erfolg versprechen. Profiling–Tools wie `gprof` [2] unterscheiden üblicherweise zwischen drei Ausgaben, deren Informationsgehalt verschieden ist. In [2] sind die verfügbaren Ergebnisse eines Profiling–Laufs wie folgt beschrieben:

- **Flat Profile**: Das Flat Profile zeigt, wie viel Zeit in jeder Funktion verbraucht und wie oft diese Funktion aufgerufen wurde. Mit dieser Information ist ersichtlich, welche Funktion die meisten Taktzyklen benötigte. So genannte Hot–Spots im Code, die Häufung der Ausführung bestimmter Codeabschnitte, sind auf diese Weise erkennbar.
- **Call Graph**: Der Aufrufbaum zeigt die Aufrufabhängigkeiten der verschiedenen Funktionen und zeigt an, wie viel Zeit in den Subroutinen einer jeden Funktion verbraucht wurde. Für den Programmierer eröffnet sich daher die Möglichkeit eventuell unnötige Funktionsaufrufe zu erkennen und zu eliminieren. Angaben über die Zeit die eine Funktion und ihre Subroutinen verbraucht stellen allerdings ein Problem dar, auf das in Kapitel 6 eingegangen wird.
- **annotierter Sourcecode**: Die Ausgabe des annotierten Sourcecodes fügt dem Programm in jeder Zeile eine Angabe bei, die darüber Auskunft gibt, wie oft diese Zeile ausgeführt wurde.

Wenn die gesammelten Daten als Eingabe für so genannte Profile–Guided Compiler benutzt werden können, so ist es möglich, automatische Korrekturen durch den Übersetzer und Linker in den Übersetzungsprozeß einfließen zu lassen. Essentiell für das Profiling ist die angesprochene Zurückverfolgbarkeit der gemessenen Ereignisse in den Sourcecode. Ist nicht nur die Häufigkeit des Aufrufs einzelner Codezeilen und Funktionen von Interesse, sondern auch der Grund für gewisse Performance–Probleme, dann ist es wünschenswert, wenn die Hardware den Anwender durch Zählerregister für Ereignistypen unterstützt.

2.6 Simulation

Ein weiteres Verfahren, Daten über den Ablauf eines Programms zu erhalten, stellt die Simulation dar. Bei der Simulation wird das zu untersuchende Programm auf dem Modell einer Hardwarearchitektur ausgeführt. Die Software des Simulators imitiert also einen Prozessor. Es liegt der Vorteil des Verfahrens darin, daß die Messung von Ereignissen nicht mehr an die Ereignistypen und Ereigniszähler des Prozessors gebunden ist, sondern die Simulation diese Typen und deren Messung zur Verfügung stellt. So ist es beispielsweise möglich, Ereignisse, wie Prozedureinsprünge und Prozedurrücksprünge zu messen, die allein mit den Möglichkeit der Hardware nicht zur Verfügung stehen. Der Nachteil liegt zum einen in der Tatsache begründet, daß die Simulation durch ihre eigene Ausführung Overhead erzeugt,

welcher in das Ergebnis einfließt und dieses verfälscht. Andererseits hängen die Ergebnisse auch davon ab, in wie weit die Simulation die imitierte Hardware annähert. Weicht das Verhalten der simulierten Hardware stark von dem der konkreten Architektur ab, so sind die Ergebnisse nicht mehr aussagekräftig.

Der Open–Source Cache–Simulator Valgrind [32] ist ein Beispiel für dieses Verfahren. Das Programm simuliert eine CPU, auf der das zu analysierende Programm ausgeführt wird. Die simulierte CPU stellt dabei Ereignistypen für die Erstellung von Cache– und Heap–Profilen und die Erkennung von Speicher– und Thread–Fehlern zur Verfügung. Auf einer konkreten Maschine ist dies dagegen nicht, oder nur eingeschränkt möglich.

3 Die Klassifizierung von Ereignistypen

Nachdem die Grundbegriffe der Ereignistypen in Kapitel 2 geklärt worden sind, kann die Einteilung der Ereignistypen in Klassen erfolgen. Zunächst gehen die Abschnitte 3.1 – 3.2 von einer theoretischen Betrachtung aus, das heißt, daß die Klassifizierung der Ereignistypen ohne Rücksicht auf die tatsächliche Umsetzung der Hersteller von Prozessoren und Betriebssystemen betrachtet wird. Dahinter steht der Zweck, einen Gesamtüberblick über die Möglichkeiten und den Nutzen verschiedener Ereignistypen zu geben. Die in diesem Abschnitt vorgeschlagene Einteilung gruppiert die Vielzahl von Ereignistypen und stellt deren Zusammenhänge dar. Für einen besseren Überblick bedienen sich die Abschnitte der UML–Notation [12].

3.1 Hardware

Bei der Klassifizierung der in der Hardware auftretenden Ereignisse, ist es sinnvoll sich am prinzipiellen Aufbau einer Rechenanlage zu orientieren, um so die Überklassen zu identifizieren. Grundsätzlich besteht ein Rechner aus dem Prozessor, an den die Peripherie angeschlossen wird. Zunächst sei das Augenmerk auf den Prozessor gelegt, danach wird die Peripherie betrachtet. Bei den schematischen Darstellungen wird zwar auf die jeweilige Architektur hingewiesen, diese jedoch öfters gemischt. Auf der hier betrachteten Ebene spielen die Unterschiede zwischen RISC– und CISC–Systemen keine Rolle, da eine für die Architekturen gleichförmige Klassifizierung gefunden werden soll.

3.1.1 Prozessor

Dieser Abschnitt befasst sich mit der zentralen Einheit, der CPU, einer Rechenanlage. Aus der schematischen Darstellung eines Prozessors soll die Ereignistypklassifizierung abgelesen werden. In Abbildung 3.1 ist ein solcher Aufbau schematisch dargestellt. Es handelt sich hier um die Darstellung eines Pentium–Prozessors der Firma Intel, also eine CISC–basierte Architektur. Die gezeigten funktionalen Einheiten sind jedoch auch in den RISC–Systemen anderer Hersteller vorhanden und unterscheiden sich lediglich in der Implementierung.

Aus der Abbildung können die nachfolgenden fünf Klassen abgelesen werden. Sie bilden übergeordnete Einheiten und daher liegt es nahe die Modellierung in UML als Interface vorzunehmen. Von jedem Interface werden im Verlaufe des Kapitels Klassen abgeleitet, welche die einzelnen Ereignistypen aufnehmen.

- **Branch Predictor**: Die BPU ist für alle Aufgaben zuständig, die mit der Ausführung von bedingten und unbedingten Sprüngen zusammenhängen. Insbesondere wird in dieser Einheit bei den modernen Prozessoren die Sprungzielvorhersage berechnet. Die

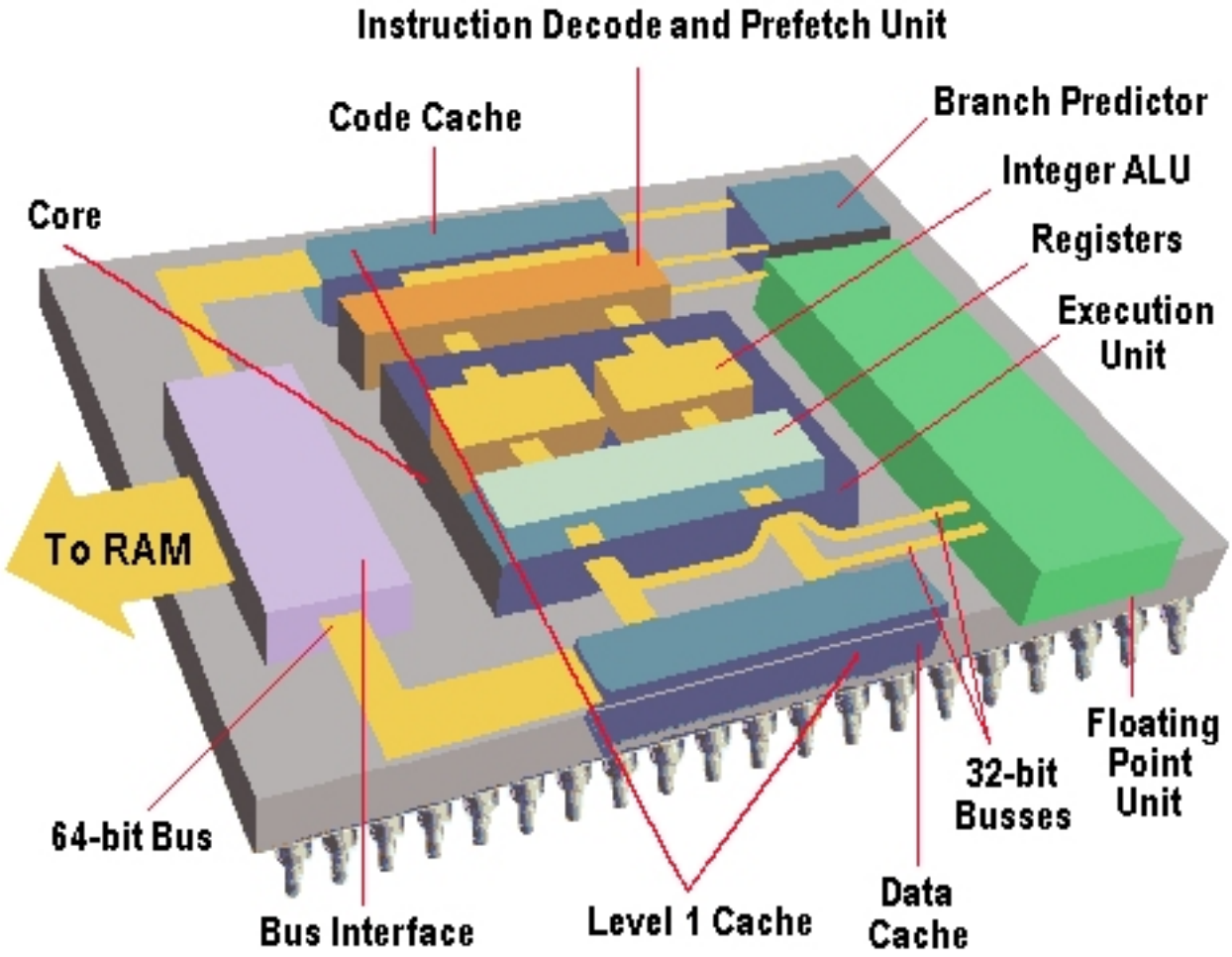


Abbildung 3.1: Schematischer Aufbau eines Prozessors – IA–32 Pentium der Firma Intel.

dazugehörigen Ereignistypen werden durch das Interface `BRANCH_EVENTS` zusammengefasst.

- **Level 1 Cache, Level 2 Cache, Level 3 Cache**: Je nach Hersteller findet sich eine mehr oder weniger stark ausgeprägte Cache–Hierarchie im Prozessor. Meist ist der Level 1 Cache ein so genannter Split–Cache bei dem Daten und Instruktionen getrennt voneinander in verschiedenen Speicherbereichen abgelegt werden. Die Performance eines Programms wird sehr durch die Rate der erfolgreichen Zugriffe auf diese Cache–Hierarchie bestimmt. Alle Ereignistypen die sich mit den Caches zusammenhängen werden mit dem Interface `MEMORY_HIERARCHY` zusammengefasst.
- **Intruction–Decode und Instruction–Prefetch**: Dieser Teil des Prozessors dekodiert Instruktionen, ist aber auch dafür verantwortlich Instruktionen im Voraus zu laden. Das Interface `BASIC_EVENTS` wird Ereignistypen wie zum Beispiel die Anzahl der abgeschlossenen Instruktionen etc. beschreiben.
- **Integer ALU und FPU**: Die Execution–Unit enthält in der Abbildung die Integer–ALU

und die Register. Die dekodierten Instruktionen werden hier verarbeitet. Die FPU gehört zu den die Instruktionen ausführenden Einheiten, so daß auch diesen Teil des Prozessors betreffende Ereignistypen unter dem Interface `INSTRUCTION_EXECUTION` zusammengefasst werden können.

- **Bus Interface**: Die Auslastung der Busse kann Auswirkungen auf die Performance haben, so daß auch hier Ereignistypen und Ereignisraten wichtige Informationen geben können. Das Interface `SYSTEM_EVENTS` enthält aber auch Ereignistypklassen, die unter Anderem die Anzahl der System–Calls, der Interrupts oder der Zugriffe auf den Bus beschreiben.

In 2.2.2 wurde angesprochen, daß es von Vorteil ist, zu wissen wie sich die verbrauchten Zyklen einer Programmausführung auf die Prozessoreinheiten aufteilen. Dem folgend fasst das Interface `CYCLE_ACCOUNTING` Ereignistypen zusammen, die beispielsweise Auskunft geben über die Anzahl der Zyklen während derer die FPU keine Instruktionen verarbeiten konnte.

In den folgenden Abschnitten werden Klassen von den Interfaces abgeleitet. Diese Klassen enthalten die Ereignistypen.

Zur Notation sei gesagt, daß direkt meßbare Ereignisse bestimmter Ereignistypen in Maschinenschrift und Großbuchstaben gekennzeichnet sind (`EREIGNISTYP`). Von diesen Ereignistypen, durch Kombination verschiedener anderer Ereignistypen, abgeleitete Typen werden zusätzlich mit `.d` gekennzeichnet.

3.1.1.1 Das Interface `BRANCH_EVENTS`

Unter dem Interface `BRANCH_EVENTS` werden Ereignistypen zusammengefasst, die eine möglichst genaue Untersuchung der Sprungzielvorhersage ermöglichen sollen. Die Ereignistypen werden in zwei Kategorien aufgeteilt:

1. **`BRANCH_INSTRUCTIONS`**: Die Attribute dieser Klasse modellieren Ereignistypen, die generell die dekodierten und abgeschlossenen Sprungbefehle beschreiben.
2. **`BRANCH_PREDICTION`**: Mit dieser Klasse soll die Sprungzielvorhersage erfasst werden. Um eine genaue Analyse zu ermöglichen, werden Ereignistypen modelliert, die separat die zwei unterschiedlichen Richtungen einer Verzweigung und den Ausgang der Vorhersage beschreiben.

Die Abbildung 3.2 zeigt die Hierarchie der im folgenden beschriebenen Klassen.

3.1.1.2 Die Klasse `BRANCH_INSTRUCTIONS`

Die Ereignistypen dieser Klasse umfassen die dekodierten und die abgeschlossenen Branch–Instruktionen. Durch Instruction–Prefetching ist es möglich, daß mehr Instruktionen dekodiert als abgeschlossen wurden. Unter Umständen ist ein solcher Zustand ein Hinweis für

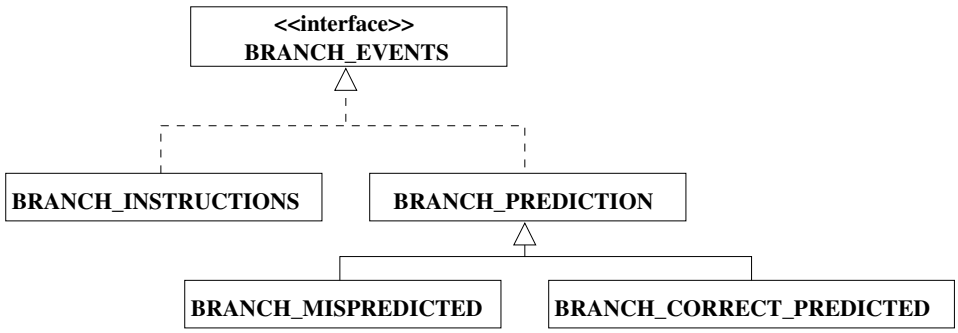


Abbildung 3.2: Die Klassenhierarchie unter dem Interface `BRANCH_EVENTS`.

Pipeline–Stalls oder eine nicht erfolgreiche Sprungzielvorhersage. Das Verhalten eines Programms kann noch genauer untersucht werden, wenn es möglich ist, auch die unbedingten Sprünge zu zählen. Profiling–Tools sind dann in der Lage Prozedureinsprünge besser zuordnen und so den Zeitverbrauch in den Prozeduren bestimmen zu können (siehe Kapitel 6). Sinnvolle Attribute der Ereignistypen dieser Klasse sind diejenigen aus Abschnitt 2.3.3. Vor allem der Branch–Trace–Buffer enthält bei unbedingten Sprüngen Informationen zu Prozedureinsprung und Prozedurrücksprung. Folgende Ereignistypen sind in der Klasse enthalten:

1. **`BRANCH_INSTRUCTIONS_DECODED`**: Ereignisse dieses Typs zählen die Anzahl der dekodierten, aber nicht zwingend abgeschlossenen Instruktionen. Sowohl bedingte als auch unbedingte Sprünge werden gezählt.
2. **`BRANCH_INSTRUCTIONS_RETIRED`**: Die abgeschlossenen Sprungbefehle werden mit Ereignisen dieses Typs gezählt.
3. **`BRANCH_INSTRUCTIONS_UNCONDITIONAL_RETIRED`**: Die abgeschlossenen unbedingten Sprungbefehle werden mit Ereignissen dieses Typs ermittelt.

Die Klasse modelliert zusätzlich die folgenden abgeleiteten Ereignistypen:

1. **`BRANCH_INSTRUCTIONS_CONDITIONAL_RETIRED.d`**: Die Ereignisse dieses Typs berechnen sich nach folgender Vorschrift:

   $$\texttt{BRANCH\_INSTRUCTIONS\_RETIRED} - \texttt{BRANCH\_INSTRUCTIONS\_UNCONDITIONAL\_RETIRED}$$

   Mit dem Ereignistyp wird die Anzahl der bedingten Sprunganweisungen beschrieben.

2. **`BRANCH_INSTRUCTIONS_DECODED_RETIRED_RATIO.d`**: Das Verhältnis von dekodierten zu abgeschlossenen Sprungbefehlen wird wie folgt berechnet:

   $$\frac{\texttt{BRANCH\_INSTRUCTIONS\_DECODED}}{\texttt{BRANCH\_INSTRUCTIONS\_RETIRED}}$$

   Das Verhältnis kann, wie oben erwähnt, auf ungünstiges Instruction–Prefetching oder Pipline–Stalls hinweisen.

3. **BRANCH_INSTRUCTIONS_RETIRED_PER_CYCLE.d**: Die Anzahl der ausgeführten Sprungbefehle pro Taktzyklus geben Aufschluß über die Linearität des Programms und werden nach folgender Formel berechnet:

$$\frac{\texttt{BRANCH\_INSTRUCTIONS\_RETIRED}}{\texttt{CPU\_CYCLES}}$$

3.1.1.3 Die Klasse BRANCH_PREDICTION

Ereignisse der Ereignistypen dieser Klasse treten bei den bedingten Verzweigungen auf. In der Klasse werden folgende Ereignistypen vorgeschlagen:

1. **BRANCH_TAKEN**: Der Ereignistyp beschreibt die `taken`–Branches, die Verzweigungen die im `if`–Zweig enden.
2. **BRANCH_NOT_TAKEN**: Mit diesem Ereignistyp werden die so genannten `not-taken`–Branches, die `then`–Zweige beschrieben.

Die Klasse enthält des weiteren einen abgeleiteten Ereignistyp:

1. **BRANCH_ALL.d**: Die bedingten Sprünge, unabhängig vom Ausgang der Verzweigung können wie folgt beschrieben werden:

$$\texttt{BRANCH\_TAKEN} + \texttt{BRANCH\_NOT\_TAKEN}$$

   In dieser Klasse ist dieser Ereignistyp und der Ereignistyp `BRANCH_INSTRUCTIONS_CONDITIONAL_RETIRED.d` aus Abschnitt 3.1.1.2 identisch.

3.1.1.4 Die Klasse BRANCH_MISPREDICTED

Diese Klasse leitet direkt von der Klasse `BRANCH_PREDICTION` ab. Sie übernimmt daher alle Ereignistypen und den abgeleiteten Typ `BRANCH_ALL.d`. Diese Ereignistypen haben die selbe Bedeutung, sie beschränken sich aber auf Verzweigungen, die von der Branch–Prediction–Unit (vergleiche Abbildung 3.1) nicht korrekt vorhergesagt wurden. Zusätzlich enthält die Klasse zwei Ereignistypen, mit denen der Grund der falschen Vorhersage genauer analysiert werden kann:

1. **BRANCH_MISPREDICTED_WRONG_PATH**: Die Branch–Prediction–Unit hat den falschen Pfad vorhergesagt. Der Ereignistyp beschreibt diese Art der falschen Vorhersage.
2. **BRANCH_MISPREDICTED_WRONG_TARGET**: Wurde das Ziel des bedingten Sprunges falsch vorhergesagt, so treten Ereignisse dieses Typs auf.

3.1.1.5 Die Klasse BRANCH_CORRECT_PREDICTED

Ebenso wie die Klasse BRANCH_MISPREDICTED leitet sich diese Klasse von BRANCH_PREDICTION ab. Alle ererbten Ereignistypen beziehen sich hier auf die korrekte Vorhersage der Branch–Prediction–Unit. Weiter Ereignistypen sind in dieser Klasse nicht enthalten.

3.1.1.6 Das Interface INSTRUCTION_EXECUTION

Unter dem Interface INSTRUCTION_EXECUTION sind Ereignistypen zusammengefasst, die sich mit der Ausführung der Maschinenbefehle beschäftigen – diese gliedern sich in fünf Gruppen. Es wird im Hinblick auf Kapitel 4 auf eine allgemeine Beschreibung Wert gelegt. Da Intel einen Trend zu EPIC angestoßen hat, der Itanium im Moment zwar noch der einzige bekannt Vertreter ist, finden sich die zu Kontroll– und Datenspekulation gehörenden Ereignistypen in der abstrakten Klasse EPIC_EVENTS. Die anderen vier Gruppen enthalten sowohl Ereignistypen der Floating–Point– und der Integer–Berechnung als auch Ereignisse das Laden und Speichern von Registern betreffend und der generellen Instruktionsausführung. In der Abbildung 3.3 ist die Klassenhierarchie dargestellt.

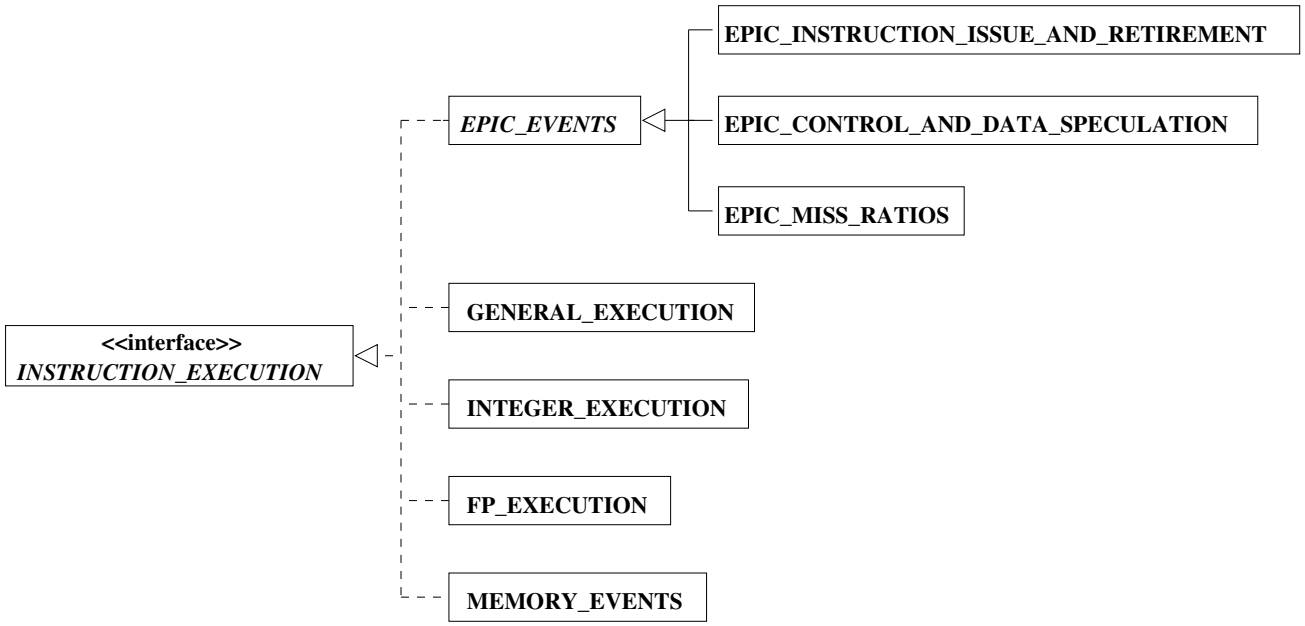


Abbildung 3.3: Die Klassenhierarchie unter dem Interface INSTRUCTION_EXECUTION.

Im folgenden sind die Klassen beschrieben:

- **EPIC_EVENTS**: Diese abstrakte Klasse bündelt die für EPIC–spezifischen Ereignistypen zur Kontroll– und Datenspekulation und gliedert sich daher in die Klassen EPIC_INSTRUCTION_ISSUE_AND_RETIREMENT, CONTROL_AND_DATA_SPECULATION und EPIC_MISS_RATIOS auf.

- **FP_EXECUTION**: Diese Klasse enthält Ereignistypen, die die Ausführung von Floating–Point–Instruktionen betrifft.
- **INTEGER_EXECUTION**: In dieser Klasse befinden sich Ereignistypen der Integer–Einheit.
- **GENERAL_EXECUTION**: Diese Klasse befasst sich mit der Instruktionsausführung im Allgemeinen.
- **MEMORY_EVENTS**: Die Klasse enthält Ereignistypen, die das Laden und Speichern von Daten beschreiben.

3.1.1.7 Die abstrakte Klasse EPIC_EVENTS

EPIC–Architekturen spezialisieren VLIW–Architekturen und versuchen den ILP in jedem Programm zu verbessern. Die größten Hindernisse für einen Compiler dies zu erreichen, sind der häufige Kontrolltransfer und unklare Datenabhängigkeiten [13, 21]. EPIC–Architekturen unterstützen den Compiler durch Daten– und Kontrollspekulation. Folglich sind spezielle Datenstrukturen von Nöten. Von dieser abstrakten Basisklasse leiten sich die Klassen `EPIC_INSTRUCTION_ISSUE_AND_RETIREMENT`, `EPIC_CONTROL_AND_DATA_SPECULATION` und `EPIC_MISS_RATIOS` ab. In diesen Klassen finden sich Ereignistypen, die die Verfügbarkeit der funktionalen Einheiten und die spekulative Ausführung beschreiben. Da der Geschwindigkeitsgewinn eines Programms von der erfolgreichen Spekulation abhängt sind in der Klasse `EPIC_MISS_RATIOS` die Raten der fehlgeschlagenen Spekulationen modelliert.

3.1.1.8 Die Klasse EPIC_INSTRUCTION_ISSUE_AND_RETIREMENT

Mit den Ereignistypen der Klasse werden folgende Ereignisse beschrieben:

- **INSTRUCTIONS_DISPERSED**: Der Ereignistyp beschreibt die Anzahl der Instruktionen, die dem Kern zugeführt wurden und auf die funktionalen Einheiten verteilt (dispersed) werden.
- **EXPLICIT_STOPS**: Das Ereignis dieses Typs tritt auf, wenn sich im abzuarbeitenden Instruktionsstrom eine explizite Stop–Anweisung befindet, so daß die parallele Ausführung von Anweisungen unterbrochen wird. Dies hat Auswirkung auf den ILP und somit auf die Performance des Programms [21].
- **IMPLICIT_STOPS_DISPERSED**: Sind einige Einheiten, etwa durch komplexe Berechnungen nicht verfügbar, so muß die ansonsten parallel ausführbare Menge von Instruktionen angehalten werden. Auch diese Unterbrechungen beeinflußen den ILP und können über Ereignisse dieses Typs identifiziert werden.

3.1.1.9 Die Klasse `EPIC_CONTROL_AND_DATA_SPECULATION`

EPIC–Architekturen wie der Intel Itanium unterstützen das spekulative Laden von Daten auch über Kontrollstrukturen hinweg. Bevor die Ereignisse der Klasse `EPIC_CONTROL_AND_DATA_SPECULATION` behandelt werden können, muß das Verfahren des spekulativen und vorgezogenen Ladens erklärt werden [4].

Die folgende Terminologie bedarf der Unterscheidung und Erklärung:

- **Datenspekulation**: Ermöglicht das Laden eines Datums und die mögliche Verwendung vor schreibende Speicherzugriffe zu verlegen. Bei diesen unklaren Speicherzugriffen wird eine Speicherposition referenziert, von der nicht bekannt ist, welche dadurch indirekt bezeichnete Speicherstelle durch einen Schreibzugriff verändert wird.
- **Kontrollspekulation**: Wird ein zu ladendes Datum und dessen mögliche Verwendung durch eine Verzweigung gesteuert, so bezeichnet man das vorgezogene Laden des des Datums über Verzweigungen hinweg als Kontrollspekulation.

Diese Techniken werden verwendet, um Lade–Latenzzeiten zu verstecken und die Ausführungsgeschwindigkeit zu erhöhen. Dabei die ALAT zum Einsatz. Im Folgenden wird diese Struktur, basierend auf der Implementierung des Intel Itanium beschrieben.

Die Datenspekulation Die Datenspekulation setzt eine spezielle Ladeanweisung (`ld.a`) ein, welche *advanced load* genannt wird. Mit dieser Anweisung ist eine überprüfende Instruktion (`chk.a` oder `ld.c`) verbunden, um die spekulativen Werte zu verifizieren. Abbildung 3.4 zeigt ein einfaches Codestück, bei dem das spekulative Laden von Daten zum Einsatz kommen kann. Es ist der Maschinencode für das folgende Beispielprogramm dargestellt:

```
  unsigned char flag; //global
int test(int *a, *b)
{
if(*a)
flag++;
return(*b-1);
}
```

Die Referenz `*b` kann nun vorgezogen und in ein Register zum schnelleren Zugriff gespeichert werden. Es ist zu beachten, daß `*b` ein Zeiger auf einen Speicherbereich ist und wiederum auf eine Speicherreferenz verweisen kann. Eine Veränderung eines solchen Wertes kann zum Beispiel RAW–Konflikte auslösen. Die Ausführung einer spekulativen Ladeanweisung mit der speziellen Instruktion `ld.a` speichert die Zielregisternummer, einen Teil der Speicheradresse und die Größe des Datums in der ALAT. Dabei findet eine Ersetzung der ursprünglichen Anweisung durch eine Testanweisung – `ld.c` oder `chk.a` statt – für den Fall, daß kein passender Eintrag in der ALAT gefunden werden konnte.

Jeder schreibende Zugriff vergleicht nun die Zieladresse mit jedem Eintrag in der ALAT. Findet sich ein solcher, so wird der entsprechende Eintrag invalidiert – in diesem Falle spricht man von einer Kollision. Bevor ein spekulativ geladenes Datum verwendet werden

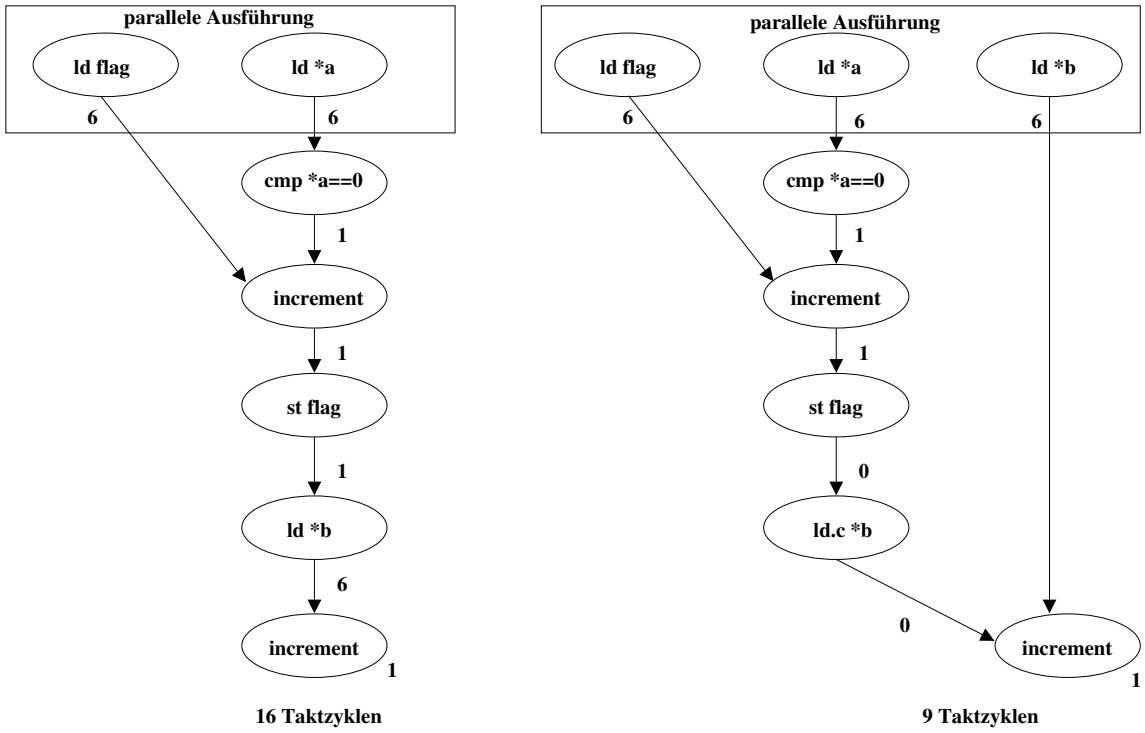


Abbildung 3.4: Beispielcode traditionell abgearbeitet und durch Datenspekulation verändert. Die Angegebenen Zahlen stellen die Minimal–Taktzyklen dar, die die Abarbeitung des Maschinenbefehls benötigt. Befehle auf einer Ebene werden parallel abgearbeitet, so daß nur die maximale Anzahl der angegebenen Taktzyklen verbraucht wird.

kann, wird wiederum mittels *ld.c* oder *chk.a* ein gültiger Eintrag in der ALAT gesucht. Ist ein gültiger, durch die Zielregisternummer indizierter Eintrag vorhanden, so trat keine Kollision auf und das Datum ist direkt zu nutzen. Andernfalls schlägt der Test fehl und das korrekte Datum muß erneut aus dem Speicher geladen werden. Die *ld.c* Instruktion lädt das Datum neu aus dem Speicher, die *chk.a* Instruktion springt zu einer speziellen Behandlungsroutine und führt den vom Compiler generierten Code aus [27].

Die Kontrollspekulation Der folgende Beispielcode zeigt die Verwendung der Kontrollspekulation:

```
conditional_branch zu_Label //bedingter Sprung zu einem
   Label
ld r1=[r5] // Laden des Registers r1 mit dem Inhalt der
   Speicheradresse von r5
add r1,r3 //Addition
```

Der Code kann derart verändert werden, daß die `load`–Anweisung vor den bedingten Sprung verschoben wird. Mittels kontrollspekulativen Ladens (*ld.s*) und Überprüfens (*chk.s*) verändert sich der Code zu:

```
ld.s r1=[r5] //vorgezogenes Laden
weitere Instruktionen
conditional_branch zu_Label
chk.s r1, behandlungsroutine
add r1,r3 //Addition
```

Gleichermaßen wie bei der Datenspekulation schreibt die Instruktion *ld.s* und liest die Instruktion *chk.s* Einträge der ALAT. Daher ist ersichtlich, daß die Performance eines Programms von der erfolgreichen Nutzung der ALAT abhängt, weil Registerzugriffe schneller vonstatten gehen als Hauptspeicherzugriffe. Folgende Ereignistypen beschreiben die ALAT:

1. **ALAT_CAPACITY_OVERFLOW**: Wie oft wurde die beschränkte Größe der ALAT überschritten, das heißt, wie oft wurden Einträge in der Tabelle ersetzt? Damit kann festgestellt werden, ob der Code anders arrangiert werden sollte, um die Anzahl der gültigen Einträge in der ALAT für die bestimmte Codestelle optimal zu halten.
2. **ALAT_CHKA_LDC_RETIRED**: Wie oft wurde eine *chk.a*– oder *ld.a*–Anweisung ausgeführt? Der Ereignistyp beschreibt, wie oft vorgezogen geladen wurde, beschränkt sich aber auf die Datenspekulation.
3. **ALAT_LDA_LDS_RETIRED**: Wie viele *ld.a*– oder *ld.s*– Instruktionen wurden ausgeführt? Ereignisse dieses Typs messen, wie oft ein Datum spekulativ geladen wurde unabhängig von Daten– oder Kontrollspekulation.
4. **ALAT_CHKA_LDC_FAIL**: Wie oft schlug die Spekulation fehl, das heißt, wie oft mußte das Datum mittels *ld.c* erneut geladen werden, bzw. wie oft wurde die Behandlungsroutine über *chk.a* angesprungen?
5. **ALAT_CHKS_SUCCESS**: Wie oft war die Kontrollspekulation erfolgreich?
6. **ALAT_CHKS_FAIL**: Wie oft schlug eine Kontrollspekulation fehl, wie oft mußte die Behandlungsroutine bei *chk.s* ausgeführt werden?

In Hinblick auf das Event–Rate–Monitoring aus 2.2.1 sind folgende abgeleitete Ereignisraten von Interesse:

1. **ALAT_CAPACITY_OVERFLOW_RATIO.d**: Die Rate der ALAT–Überläufe können mittels folgender Formel berechnet werden:
$$\frac{\texttt{ALAT\_CAPACITY\_OVERFLOW}}{\texttt{ALAT\_LDA\_LDS\_RETIRED}}$$
2. **ALAT_CONTROL_SPECULATION_FAIL_RATIO.d**: Mit
$$\frac{\texttt{ALAT\_CHKA\_LDC\_FAIL}}{\texttt{ALAT\_CHKA\_LDC\_RETIRED}}$$
wird die Rate der fehlgeschlagenen Datenspekulation ermittelt.
3. **ALAT_CONTROL_SPECULATION_RATIO().d**: Die Rate der fehlgeschlagenen Kontrollspekulation kann durch
$$\frac{\texttt{ALAT\_CHKS\_FAIL}}{(\texttt{ALAT\_CHKS\_SUCCESS} + \texttt{ALAT\_CHKS\_FAIL})}$$
berechnet werden.

3.1.1.10 Die Klasse FP_EXECUTION

Die Klasse FP_EXECUTION beschäftigt sich mit der Beschreibung von Ereignistypen, die durch die Fließkommaeinheit der CPU (vergleiche Abbildung 3.1) erzeugt werden.

1. **FP_INSTRUCTIONS_RETIRED**: Ereignisse dieses Typs zählen die Anzahl der ausgeführten Fließkommainstruktionen an – die so genannten FLOPS.
2. **FP_MUL_INSTRUCTIONS_RETIRED**: Über Ereignisse dieses Typs kann die Anzahl der Fließkommamultiplikationen ermittelt werden. Da Multiplikationen und Divisionen im Vergleich zu Additionen und Subtraktionen mehr Zyklen beanspruchen, werden die Ereignistypen nach den auszuführenden Operationen unterschieden.
3. **FP_DIV_INSTRUCTIONS_RETIRED**: Der Ereignistyp beschreibt die Anzahl der Fließkommadivisionen.
4. **FP_ADD_INSTRUCTIONS_RETIRED**: Mit diesem Ereignistyp wird die Anzahl der Fließkommaadditionen ermittelt.
5. **FP_SUB_INSTRUCTIONS_RETIRED**: Entsprechend beschreibt dieser Ereignistyp die Anzahl der Fließkommasubtraktionen.

Von Interesse ist auch die FLOP–Rate. Sie kann wie folgt bestimmt werden:

$$FLOPRATE = \frac{FLOP}{CPU\_CYCLES}$$

und ist im zugrunde liegenden Modell ein abgeleiteter Ereignistyp über zwei Klassen hinweg, da der Divisor ein Ereignistyp der Klasse BASIC_EVENTS (siehe Abschnitt 3.1.1.18) ist. Der Ereignistyp **FP_FLOPRATE** wird wie folgt berechnet:

- **FP_FLOPRATE.d**:

$$\texttt{FP\_FLORATE} = \frac{\texttt{FP\_INSTRUCTION\_RETIRED}}{\texttt{CPU\_CYCLES}}$$

3.1.1.11 Die Klasse INTEGER_EXECUTION

Die Ereignistypen, welche in der Klasse INTEGER_EXECUTION enthalten sind, beziehen sich auf die Integereinheit der CPU (vergleiche Abbildung 3.1) und sind denen der Klasse FP_EXECUTION aus Abschnitt 3.1.1.10 ähnlich.

1. **INTEGER_INSTRUCTIONS_RETIRED**: Der Ereignistyp beschreibt alle abgeschlossenen Integerinstruktionen.
2. **INTEGER_ADD_INSTRUCTIONS_RETIRED**: Ereignisse dieses Typs zählen die Anzahl der abgeschlossenen Integer–Additionen.
3. **INTEGER_SUB_INSTRUCTIONS_RETIRED**: Ereignisse dieses Typs treten auf, wenn Integer–Subtraktionen abgeschlossen wurden.

4. **INTEGER_MUL_INSTRUCTIONS_RETIRED**: Der Ereignistyp beschreibt die Integer–Multiplikationen.

5. **INTEGER_DIV_INSTRUCTIONS_RETIRED**: Der Ereignistyp beschreibt die Integer–Divisionen.

Die Aufteilung der Ereignisse auf die verschiedenen arithmetischen Operationen scheint für die Integereinheit ungewöhnlich, da bei den vielen Architekturen alle Befehle innerhalb eines Zyklus abgeschlossen werden können. Andere (ältere) Systeme, wie M88000 oder MIPS bedienen sich der Fließkommaeinheit, da Multiplikation und Division hier mehr Zyklen beanspruchen (vergleiche [10]).

3.1.1.12 Die Klasse GENERAL_EXECUTION

Ereignisse, die während der allgemeinen Instruktionsausführung auftreten, werden durch die Typen dieser Klasse modelliert. Dabei werden die Ausführungsschritte einer Pipeline (vergleiche [10]) zugrunde gelegt. Die Klasse enthält die folgenden Ereignistypen:

1. **INSTRUCTION_PREFETCHES**: Unterstützt die zugrunde liegende Architektur Instruktion–Prefetching, so kann über dieses Ereignis die Anzahl der Anfragen auf die Cache–Hierarchie genauer untersucht werden. In Abschnitt 3.1.1.23 wird dieser Aspekt genauer beschrieben und in Kapitel 4 die Implementierung beleuchtet.

2. **INSTRUCTION_DECODED**: Der Ereignistyp beschreibt die Anzahl der dekodierten Instruktionen.

3. **INSTRUCTION_RETIRED**: Mit Ereignissen dieses Typs ist es möglich die Anzahl der ausgeführten Instruktionen zu bestimmen.

4. **INSTRUCTION_NOP_RETIRED**: Der Compiler oder der Prozessor fügt an bestimmten Stellen "No–Operations", so genannte NOPS ein, um möglicherweise auftretende Pipeline–Stalls zu verhindern. Im Gegensatz dazu steht das so genannte "Interlocking" der Pipelines. Bei diesem Verfahren wird die Pipeline angehalten, um Zyklen verstreichen zu lassen bis der Konflikt behoben ist. Der Verbrauch dieser Zyklen kann Aufschluß über Pipeline–Probleme und einen damit zusammenhängenden Performance–Verlust geben.

Folgende abgeleitete Ereignistypen sind in der Klasse enthalten:

1. **INSTRUCTION_DECODE_RETIRE_RATIO.d**:

$$\frac{\texttt{INSTRUCTION\_DECODED}}{\texttt{INSTRUCTION\_RETIRED}}$$

   Die Rate kann in Verbindung mit Sprungzielvorhersage und Kontroll– und Datenspekulation (vergleiche Abschnitt 3.1.1.7) gebracht werden. Werden mehr Instruktionen dekodiert als in der WB–Phase übernommen, ist die Rate also > 1, so kann dies ein Hinweis auf fehlgeschlagene Sprungzielvorhersage oder fehlerhafte Spekulation sein.

2. **INSTRUCTION_NOP_NONOP_RATIO().d**:

$$\frac{\texttt{INSTRUCTION\_NOP\_RETIRED}}{\texttt{INSTRUCTION\_RETIRED} - \texttt{INSTRUCTION\_NOP\_RETIRED}}$$

   Der Quotient kann auf Pipeline–Stalls – wie oben beschrieben – und ungünstig arrangierten Code hinweisen.

3.1.1.13 Die Klasse MEMORY_EVENTS

Die Klasse der MEMORY_EVENTS befasst sich mit LOAD– und STORE–Operationen der CPU. Bei RISC–Architekturen spielt die LSU eine wesentliche Rolle, aber auch bei CISC–Systemen produzieren unausgerichtete Daten Extrazyklen. Die Klasse enthält die folgenden Ereignistypen:

1. **MEMORY_LOADS_RETIRED**: Ereignisse dieses Typs können dazu verwendet werden, die Anzahl der LOAD–Operationen zu ermitteln. Eine Unterscheidung zwischen Uncacheable–LOADS oder nicht ausgerichteten Daten wird nicht vollzogen.

2. **MEMORY_STORES_RETIRED**: Mit Hilfe der Ereignisse dieses Typs werden die STORE–Operationen gezählt. Wie oben, wird nicht zwischen Uncacheable–Data und nicht ausgerichteten Daten unterschieden.

3. **MEMORY_UC_LOADS_RETIRED**: LOAD–Operationen, die sich auf Daten beziehen, welche nicht zu puffern sind, werden durch diesen Typ beschrieben. Die so genannten Uncacheable–Daten sind zum Beispiel Speicherbereiche der Ein-/Ausgabe.

4. **MEMORY_UC_STORES_RETIRED**: Mit den Ereignissen dieses Typs können STORE–Operationen in nicht zu puffernde Speicherbereich ermittelt werden.

5. **MEMORY_UNALIGNED_LOADS_RETIRED**: Unausgerichtete Speicherzugriffe stellen gerade bei SIMD–Anwendungen in Multimediaanwendungen ein Problem dar, wenn ein Maschinenbefehl verlangt, n Bytes gleichzeitig zu laden oder zu speichern, um mit der Größe des Quell– oder Zielregisters übereinzustimmen. Kann die Ausrichtung der Daten nicht garantiert werden, so müssen Maschinenbefehle verwendet werden, die das Laden unausgerichteter Daten unterstützen, aber langsamer sind. Empirische Untersuchungen [34] haben ergeben, daß eine Ladeoperation von 16 unausgerichteten Bytes, die sich innerhalb einer Cache–Zeile befinden nur ca. 40% langsamer vonstatten geht als eine Ladeoperation auf ausgerichteten Daten. Der Performanceverlust wird allerdings ungleich größer, wenn die Daten über zwei Cache–Zeilen hinweg verteilt sind. Das sogenannte Cache–Zeilen–Splitting kann dann um den Faktor fünf langsamer sein. Der Ereignistyp beschreibt die Zugriffe auf unausgerichtete Daten.

6. **MEMORY_UNALIGNED_STORES_RETIRED**: Auch STORE–Instruktionen verlangen oft das Speichern von Daten an bestimmten Byte–Grenzen. Kann dies nicht garantiert werden, so müssen Instruktionen für unausgerichtetes Speichern von Daten verwendet werden. Mit Hilfe der Ereignisse dieses Typs können diese Instruktionen gezählt werden, um so auf dadurch verursachte Performance–Verlust schließen zu können.

Des weiteren enthält die Klasse folgende abgeleitete Ereignistypen:

1. **MEMORY_UNALIGNED_MEM_REFS.d**: Die Gesamtanzahl von Speicherzugriffen auf unausgerichtete Daten kann wie folgt berechnet werden:

   MEMORY_UNALIGNED_LOADS_RETIRED + MEMORY_UNALIGNED_STORES_RETIRED

2. **MEMORY_ALIGNED_LOADS_RETIRED.d**: Die lesenden Speicherzugriffe auf ausgerichtete Daten können wie folgt berechnet werden:

   MEMORY_LOADS_RETIRED − MEMORY_UNALIGNED_LOADS_RETIRED

3. **MEMORY_ALIGNED_STORES_RETIRED.d**: Die schreibenden Speicherzugriffe auf ausgerichtete Daten können mit folgender Formel berechnet werden:

   MEMORY_STORES_RETIRED − MEMORY_UNALIGNED_STORES_RETIRED

4. **MEMORY_ALIGNED_MEM_REFS.d**: Durch Addition der Werte der Ereignistypen MEMORY_ALIGNED_LOADS_RETIRED.d und MEMORY_ALIGNED_STORES_RETIRED.d ergibt sich die Zahl der Speicherzugriffe auf ausgerichtete Daten.

3.1.1.14 Das Interface CYCLE_ACCOUNTING

In Abschnitt 2.2.2 wurde darauf hingewiesen, daß es sinnvoll ist, den Verbrauch an Taktzyklen den Verursachern zuschreiben zu können. Es wurde angesprochen, daß im Hinblick auf Performance–Engpässe hauptsächlich Stall–/Flush–Bedingungen der Pipelines für derartige Geschwindigkeitsverluste verantwortlich sind. Es sollen die Ereignistypen modelliert werden, die den Verbrauch an Taktzyklen den Hauptaktivitäten während des Programmablaufes zuordnen. Diese Aktivitäten sind die Instruktionsausführung im Allgemeinen, Sprünge im Programm und der Speicherzugriff. Das Interface kapselt daher die Klassen INSTRUCTION_CYCLES, BRANCH_CYCLES und MEMORY_CYCLES. In Abbildung 3.5 ist die Klassenhierarchie dargestellt.

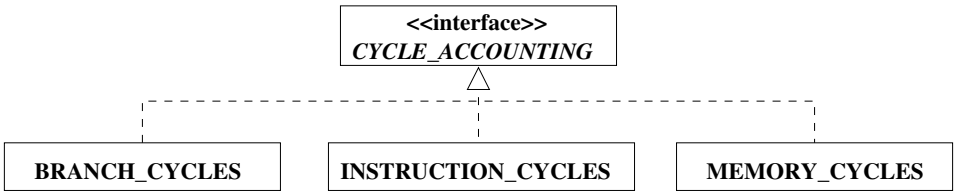


Abbildung 3.5: Die Klassenhierarchie unter dem Interface CYCLE_ACCOUNTING.

3.1.1.15 Die Klasse BRANCH_CYCLES

Wie in Abschnitt 3.1.1.14 angesprochen beschreibt diese Klasse Ereignistypen, die den Verbrauch an Taktzyklen der BPU zuschreiben.

1. **CYCLES_BPU_IDLE**: Der Ereignistyp beschreibt die Anzahl von Taktzyklen während derer die Sprungzielvorhersageeinheit inaktiv war. Der Wert gibt Auskunft über den Programmablauf, das heißt, ob dieser geradlinig ist oder sehr verzweigt.
2. **CYCLES_BPU_MISPREDICTED**: Ein Ereignis dieses Typs gibt an, daß eine Verzweigung fehlerhaft vorhergesagt wurde. Die Pipeline muß danach angehalten beziehungsweise gelöscht werden, um die korrekten Instruktionen nachzuladen.
3. **CYCLES_BPU_TAKEN_BRANCH**: Dieser Ereignistyp beschreibt die Anzahl der Zyklen während derer die Pipeline angehalten wurde, beziehungsweise NOPS eingefügt wurden, um nach einem gewählten Zweig die folgenden Instruktionen zu laden. Ein häufiges Vorkommen solcher Ereignisse korreliert mit dem Auftreten von Ereignisse des Typs `CYCLES_BPU_MISPREDICTED`, da schon vorausgeladenen Instruktionen verworfen werden müssen und die korrekten nachgeladen werden müssen.

Aus den obigen Ereignistypen kann das folgende Ereignis abgeleitet werden:

1. **CYCLES_BPU_STALL.d**: Der abgeleitete Ereignistyp beschreibt, wie viele Stall–Zyklen die BPU verursacht hat. Der Typ wird wie folgt berechnet:

$$\texttt{CYCLES\_BPU\_TAKEN\_BRANCH} + \texttt{CYCLES\_BPU\_MISPREDICTED}$$

3.1.1.16 Die Klasse INSTRUCTION_CYCLES

In dieser Klasse werde Ereignistypen modelliert, die den Verbrauch an Wartezyklen durch das Laden der Instruktionen und die anschließende Verteilung auf die Funktionseinheiten beschreiben.

1. **CYCLES_EXEC_LATENCY**: Wartezeiten bei der Ausführung entstehen durch Funktionseinheiten, die die Instruktionen nicht innerhalb eines Zyklus abarbeiten können. Dazu zählen vor allem die FPU und das Scoreboard. Dieser Ereignistyp beschreibt die Wartezyklen in beiden Einheiten.
2. **CYCLES_FPU_STALL**: Dieser Ereignistyp gibt Auskunft über die Anzahl der Zyklen während derer die FPU angehalten wurde. Gründe dafür können Register–Register–Abhängigkeiten sein, oder Latenzzeiten beim Laden von Operanden.
3. **CYCLES_INSTR_ACCESS**: Kann eine Instruktion nicht im TLB oder im Instruktions–Cache gefunden werden, so verlangsamt sich der Zugriff. Dieser Ereignistyp beschreibt die Anzahl von Zyklen, die unter solchen Bedingungen verbraucht wurden, also die Stall–Zyklen in der IF–Phase.

4. **CYCLES_INSTR_ISSUE**: Können Instruktionen nicht auf die funktionalen Einheiten verteilt werden weil diese belegt sind, so entstehen ebenfalls Wartezyklen. Mit diesem Ereignistyp soll dieser Umstand beschrieben werden.

Durch die Kombination der Ereignistypen, können folgende zwei Ereignistypen abgeleitet werden:

1. **CYCLES_SCOREBOARD_STALL.d**: Die Anzahl der Zyklen während derer sich das Scoreboard in einer Stall–Situation befand, ergibt sich durch:

$$\texttt{CYCLES\_EXEC\_LATENCY} - \texttt{CYCLES\_FPU\_STALL}$$

2. **CYCLES_EXEC_STALL.d**: Die kombinierten Stalls der Instruktionsausführung ergeben sich durch folgende Summe:

$$\texttt{CYCLES\_INSTR\_ISSUE} + \texttt{CYCLES\_EXEC\_LATENCY}$$

   Die Latenzzeiten der Instruktionsausführung, nicht aber die des Zugriffs auf Instruktionen, können so ermittelt werden.

3.1.1.17 Die Klasse MEMORY_CYCLES

Der Ereignistyp dieser Klasse ist für die Ermittlung des Anteils an verbrauchten Wartezyklen durch Stall–Situationen des Speichers bestimmt.

1. **CYCLES_DATA_ACCESS**: Dieser Ereignistyp beschreibt die Wartezyklen, die durch volle Memory–Pipelines, DTLB–Stalls oder RAW–Konflikte entstehen.

3.1.1.18 Das Interface BASIC_EVENTS

Dieses Interface umfasst in der abgeleiteten Klasse INSTRUCTION_DECODE_RETIREMENT Ereignistypen zur Beobachtung der grundlegenden Instruktionsverarbeitung. Die Abbildung 3.6 zeigt die Klassenhierarchie unter dem Interface.

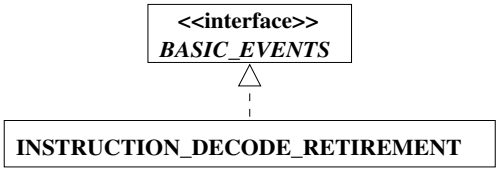


Abbildung 3.6: Die Klassenhierarchie unter dem Interface BASIC_EVENTS.

3.1.1.19 Die Klasse INSTRUCTION_DECODE_RETIREMENT

Diese Klasse beinhaltet Ereignistypen, die grundlegende Informationen zur Instruktionsverarbeitung bereitstellen. Im wesentlichen geben die Ereignistypen über die Anzahl der verstrichenen Taktzyklen und die Anzahl der dekodierten Instruktionen Auskunft.

1. **CPU_CYCLES**: Ereignisse dieses Typs zählen die Anzahl der verstrichenen Taktzyklen. In Verbindung mit der Position des Instruktionszeigers kann ermittelt werden, wo das Programm die meisten Zyklen verbraucht. Diese so genannten Hot–Spots sind der erste Anhaltspunkt für eine genauere Untersuchung des Codes.
2. **INST_DECODED**: Ereignisse dieses Ereignistyps können dazu verwendet werden, die Anzahl der in der ID–Phase befindlichen Instruktionen zu zählen.
3. **INST_RETIRED**: Nicht alle dekodierten Instruktionen werden auch abgeschlossen und gehen in die WB–Phase über. Gründe hierfür sind zum Beispiel falsche Sprungzielvorhersage (vergleiche Abschnitt 3.1.1.4). Der Ereignistyp beschreibt die tatsächlich abgeschlossenen Instruktionen.
4. **INST_DISPATCHED**: Ereignisse dieses Typs können dazu verwendet werden, die Anzahl der auf die Funktionseinheiten verteilten Instruktionen zu ermitteln.

Die Klasse modelliert des weiteren drei abgeleitete Ereignistypen:

1. **INST_DECODED_PER_CYCLE.d**: Die Anzahl der pro Taktzyklus dekodierten Instruktionen kann wie folgt berechnet werden:
$$\frac{\texttt{INST\_DECODED}}{\texttt{CPU\_CYCLES}}$$
2. **INST_RETIRED_PER_CYCLE.d**: Die Anzahl der pro Taktzyklus abgeschlossenen Instruktionen berechnet sich analog nach folgender Vorschrift:
$$\frac{\texttt{INST\_RETIRED}}{\texttt{CPU\_CYCLES}}$$
3. **INST_RETIRED_DECODED_RATIO.d**: Die Rate der abgeschlossenen zu den dekodierten Instruktionen kann nach der folgenden Formel berechnet werden:
$$\frac{\texttt{INST\_RETIRED}}{\texttt{INST\_DECODED}}$$
Je näher die Rate an 1 liegt, um so weniger Instruktionen sind ohne Nutzen dekodiert worden.

3.1.1.20 Das Interface SYSTEM_EVENTS

Das Interface SYSTEM_EVENTS umfasst die zwei Klassen PROCESSOR_SYSTEM_EVENTS und PROCESSOR_TLB_PERFORMANCE. Die Klasse PROCESSOR_SYSTEM_EVENTS kapselt Systemereignistypen, die die Anzahl von System–Calls und Interrupts etc. beschreiben. Die Klasse PROCESSOR_TLB_PERFORMANCE umfasst Ereignistypen, die die Performance des TLB beschreiben. Die Abbildung 3.7 zeigt die Klassenhierarchie der im weiteren beschriebenen Klassen.

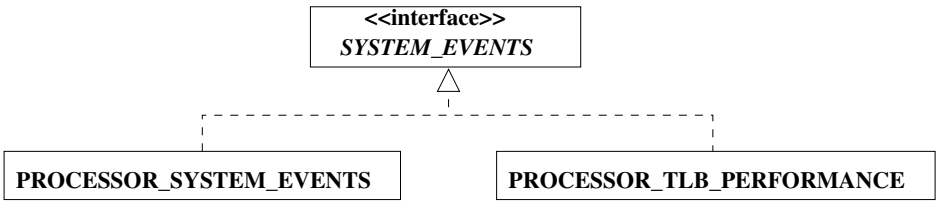


Abbildung 3.7: Die Klassenhierarchie unter dem Interface `SYSTEM_EVENTS`.

3.1.1.21 Die Klasse `PROCESSOR_SYSTEM_EVENTS`

Diese Klasse beinhaltet die folgenden Ereignistypen zur Untersuchung von Systemereignissen:

1. **`PROCESSOR_CPL_CHANGES`**: Mit Ereignissen dieses Typs kann die Anzahl der Privilege–Level–Wechsel gezählt werden. Diese Übergänge werden durch Systemaufrufe aber auch durch die `rfi`–Instruktion[1] verursacht.
2. **`PROCESSOR_SYSTEM_CALLS`**: Ereignisse dieses Typs zählen die Anzahl der System Calls. Dieser Zähler erhöht auch die Anzahl der Ereignisse des Ereignistyps `PROCESSOR_CPL_CHANGES`.
3. **`PROCESSOR_PIPELINE_FLUSHES_BRANCH`**: Der Ereignistyp beschreibt die Anzahl der durch fehlerhafte Sprungzielvorhersage verursachten Invalidierungen der Pipeline. Die Beobachtung dieses Ereignistyps kann den Grund für verminderten Durchsatz liefern.
4. **`PROCESSOR_PIPELINE_FLUSHES_EXCEPTION`**: Durch Exceptions und Interrupts ausgelöste Löschungen der Pipeline werden mit diesem Ereignistyp bestimmt. Ebenfalls liefert die Beobachtung dieses Ereignistyps unter Umständen den Grund für verminderten Durchsatz und Verlust von Performance.
5. **`PROCESSOR_BUS_REQ_OUTSTANDING`**: Ereignisse dieses Typs geben an, wie viele Anforderungen auf Buszugriffe ausstehen und noch nicht behandelt wurden.
6. **`PROCESSOR_BUS_TRAN_MEM`**: Mit Ereignissen dieses Typs kann die Anzahl der abgeschlossenen Speicherzugriffe über den Bus ermittelt werden. In Verbindung mit einer hohen Anzahl von Daten–Cache–Misses lassen sich somit eventuell Performance–Probleme erkennen.
7. **`PROCESSOR_BUS_TRAN_IFETCH`**: Ereignisse dieses Typs zählen die abgeschlossenen Ladevorgänge über den Bus von Instruktionen. Die Anzahl der auftretenden Ereignisse korreliert mit Instruktions–Cache–Misses, da schließlich die Instruktionen aus dem Speicher geladen werden müssen.
8. **`PROCESSOR_BUS_TRAN_IO`**: Dieser Ereignistyp gibt Auskunft über die Anzahl abgeschlossener I/O–Übertragungen.

[1] **R**eturn **F**rom **I**nterrupt: Diese Anweisung reinitialisiert das MSR und setzt die Programmabarbeitung nach einem Interrupt fort.

9. **PROCESSOR_BUS_DATA_RCV**: Ereignisse dieses Typs bestimmen die Anzahl der Zyklen, während derer der Prozessor Daten empfängt.

10. **PROCESSOR_BUS_LOAD**: Mit diesem Ereignistypen kann die Auslastung des Busses bestimmt werden. Ist der Bus vollständig ausgelastet, dann tritt auch vermehrt das Ereignis PROCESSOR_BUS_REQ_OUTSTANDING auf. Dieser Typ ist insbesondere von Interesse, wenn das Programm zu viele Zyklen während des Wartens auf den Bus verbraucht.

11. **PROCESSOR_INTERRUPT_HW**: Ereignisse dieses Typs sind für die Bestimmung der Anzahl von Hardwareinterrupts zuständig.

Aus diesen Ereignistypen können weitere Typen abgeleitet werden. Die folgende Aufzählung erklärt diese abgeleiteten Ereignistypen.

1. **PROCESSOR_PIPELINE_FLUSHES_ALL.d**: Alle Ursachen für Pipeline–-Flushes können einfach durch Summation der aufgetretenen Ereignisse der Typen PROCESSOR_PIPELINE_FLUSHES_BRANCH und PROCESSOR_PIPELINE_FLUSHES_EXCEPTION ermittelt werden.

2. **PROCESSOR_RFI_CPL_CHANGES.d**: Die Anzahl der durch rfi–Instruktionen erzeugten Privilege–Level–Wechsel kann durch folgende Subtraktion beschrieben werden:

   $$\texttt{PROCESSOR\_CPL\_CHANGES} - \texttt{PROCESSOR\_SYSTEM\_CALLS}$$

   Die Anzahl aufgetretener Interrupts kann die Performance eines Programms beeinflussen. Wenn beispielsweise die aufgetretenen Interrupts durch Festspeicherzugriffe ausgelöst werden, dann bremst dies die Programmausführung.

3. **PROCESSOR_INTERRUPTS_IN.d**: Die Anzahl der durch die Peripherie erzeugten Interrupts kann ermittelt werden in dem die Anzahl der System–Calls (PROCESSOR_SYSTEM_CALLS) von der Anzahl der Hardwareinterrupts (PROCESSOR_INTERRUPT_HW) subtrahiert wird. Die Behandlungsroutinen der System–Calls werden über Interrupts angesprungen. Dem stehen die Unterbrechungen des Programmflußes entgegen, die durch die Peripherie verursacht werden. Ereignisse dieses abgeleiteten Ereignistyps zählen diese Unterbrechungen.

3.1.1.22 Die Klasse PROCESSOR_TLB_PERFORMANCE

Mit den Ereignistypen dieser Klasse kann die Leistung des TLB überprüft werden.

1. **PROCESSOR_ITLB_REFERENCES**: Der Ereignistyp beschreibt die Zugriffe auf den Instruktions–TLB.

2. **PROCESSOR_ITLB_MISSES**: Vermittels diese Ereignistyps lassen sich die fehlgeschlagenen Versuche eine Instruktionsadresse über den TLB zu dekodieren beschreiben. Analog zum Ereignistyp PROCESSOR_DTLB_MISSES korrelieren die Ereignisse mit den Instruktions–Cache–Misses.

3. **PROCESSOR_DTLB_REFERENCES**: Analog beschreibt dieser Ereignistyp die Zugriffe auf den Daten–TLB.

4. **PROCESSOR_DTLB_MISSES**: Dieser Ereignistyp dient der Beschreibung der fehlgeschlagenen Versuche eine Datumsadresse über den TLB zu dekodieren. Dieser Zähler korreliert mit den Daten–Cache–Misses [38].

Von diesen Ereignistypen lassen sich weitere Ereignistypen ableiten:

1. **PROCESSOR_DTLB_MISS_RATIO.d**: Die Rate der fehlgeschlagenen Zugriffe auf den Daten–TLB läßt sich wie folgt berechnen:

$$\frac{\texttt{PROCESSOR\_DTLB\_MISSES}}{\texttt{PROCESSOR\_DTLB\_REFERENCES}}$$

2. **PROCESSOR_ITLB_MISS_RATIO.d**: Gleichermaßen läßt sich die Rate der fehlgeschlagenen Zugriffe auf den ITLB bestimmen:

$$\frac{\texttt{PROCESSOR\_ITLB\_MISSES}}{\texttt{PROCESSOR\_ITLB\_REFERENCES}}$$

3. **PROCESSOR_TOTAL_MISSES.d**: Die gesamten erfolglosen Zugriffe auf den TLB ergeben sich aus der Summe von PROCESSOR_DTLB_MISSES und PROCESSOR_ITLB_MISSES.

4. **PROCESSOR_ITLB_HITS.d**: Die erfolgreichen Zugriffe auf den ITLB werden wie folgt ermittelt:

$$\texttt{PROCESSOR\_ITLB\_REFERENCES} - \texttt{PROCESSOR\_ITLB\_MISSES}$$

5. **PROCESSOR_DTLB_HITS.d**: Gleichermaßen bestimmt man die erfolgreichen Zugriffe auf den DTLB:

$$\texttt{PROCESSOR\_DTLB\_REFERENCES} - \texttt{PROCESSOR\_DTLB\_MISSES}$$

3.1.1.23 Das Interface MEMORY_HIERARCHY

Das Interface kapselt die Ereignistypen, welche sich mit der Speicherhierarchie des Prozessors beschäftigen. Daher beinhaltet das Interface L1_CACHE die Klasse L1_CACHE_DATA mit Ereignistypen für den L1–Datencache und die Klasse L1_CACHE_INSTR mit Ereignistypen für den L1–Instruktions–Cache. Der L2–Cache wird im Gegensatz zum L1–Cache gemeinsam für Daten und Instruktionen verwendet, deshalb umfasst die einzelne Klasse L2_CACHE die Ereignistypen des L2–Caches. Manche Systeme besitzen des weiteren einen dritten Cache, den L3–Cache. Die Ereignistypen dieses Caches sind in der Klasse L3_CACHE zusammengefasst. Schließlich sind in der Klasse CACHE-PERFORMANCE Ereignistypen enthalten, die wichtige Raten der Cache–Zugriffe beschreiben. Die Hierarchie der Klassen unter dem Interface MEMORY_HIERARCHY sind in Abbildung 3.8 dargestellt.

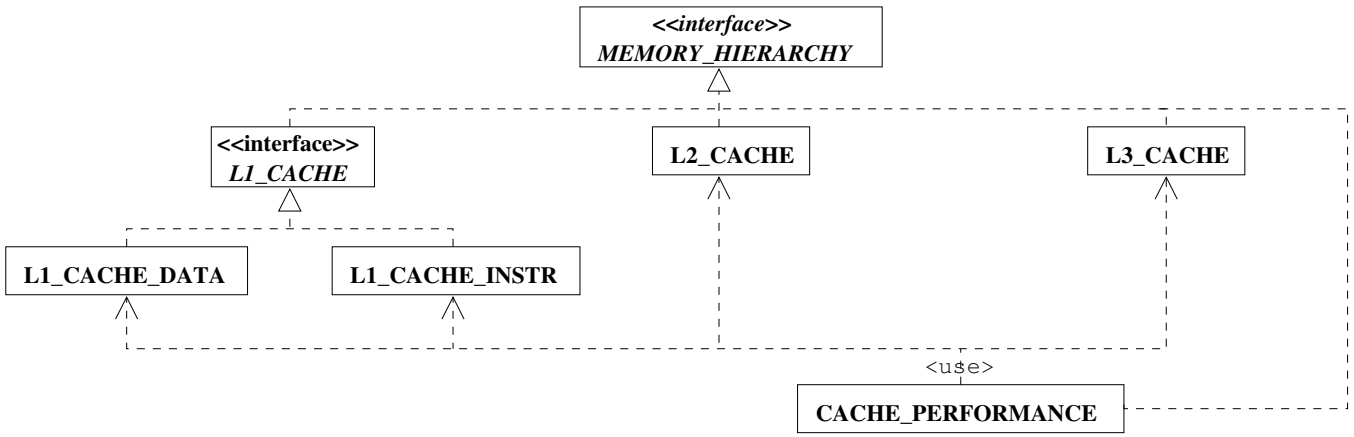


Abbildung 3.8: Die Klassenhierarchie unter dem Interface `MEMORY_HIERARCHY`.

3.1.1.24 Das Interface `L1_CACHE`

Wie oben angesprochen beinhaltet das Interface die Klassen `L1_CACHE_DATA` und `L1_CACHE_INSTR`, um auszudrücken, daß dieser Cache meist ein Split–Cache ist.

3.1.1.25 Die Klasse `L1_CACHE_DATA`

1. **`L1_DATA_REFERENCES`**: Mit diesem Ereignistyp werden die Zugriffe auf den L1–Daten–Cache beschrieben.
2. **`L1_DATA_MISSES`**: Die erfolglosen Zugriffe auf den Cache beschreibt dieser Ereignistyp.

Aus den obigen Ereignistypen kann der folgende Ereignistyp abgeleitet werden:

1. **`L1_DATA_HITS.d`**: Werden die Anzahl der aufgetretenen Ereignisse vom Typ `L1_DATA_REFERENCES` von der Anzahl der Ereignisse des Typs `L1_DATA_REFERENCES` subtrahiert, so ergibt sich die Anzahl der erfolgreichen Zugriffe auf den Datencache.

3.1.1.26 Die Klasse `L1_CACHE_INSTR`

Diese Klasse beschreibt Ereignistypen für den L1–Instruktions–Cache. Die Ereignistypen sind denen des Daten–Cache ähnlich.

1. **`L1_INSTR_REFERENCES`**: Der Ereignistyp beschreibt die Zugriffe auf den Instruktions–Cache.
2. **`L1_INSTR_MISSES`**: Mit diesem Ereignistyp werden die erfolglosen Zugriffe auf diesen Cache beschrieben.

Analog zum Datencache ergibt sich der folgende abgeleitete Ereignistyp:

1. **L1_INSTR_HITS.d**: Durch Subtraktion de Anzahl der aufgetretenen Ereignisse des Ereignistyps L1_INSTR_REFERENCES von der Anzahl der aufgetretenen Ereignisse des Typs L1_INSTR_MISSES ergibt sich der Ereignistyp, welcher die Anzahl der erfolgreichen Zugriffe auf den Cache beschreibt.

3.1.1.27 Die Klasse L2_CACHE

Die Klasse enthält Ereignistypen, die die Funktion des gemeinsamen L2–Daten– und Instruktions–Caches beschreiben.

1. **L2_INSTR_PREFETCHES**: Der Ereignistyp beschreibt die Anfragen auf den Cache Instruktionen vorauszuladen.
2. **L2_INSTR_FETCH**: Dieser Ereignistyp beschreibt erfolgte Lesezugriffe auf Instruktionen des Caches. Da diese Zugriffe nur dann zustande kommen, wenn die Instruktion im L1–Cache nicht gefunden werden konnten, ist der Ereignistyp äquivalent zu dem Ereignistyp L1_INSTR_MISSES.
3. **L2_DATA_REFERENCES**: Mit Ereignissen dieses Typs sind die Zugriffe auf Daten des L2–Caches bestimmbar.
4. **L2_DATA_READS**: Die lesenden Zugriffe auf Daten werden mit diesem Ereignistyp beschrieben.
5. **L2_MISSES**: Erfolglose Zugriffe auf den L2–Cache beschreibt dieser Ereignistyp. Der Ereignistyp ist äquivalent zu dem Ereignistypen L3_REFERENCES in Systemen mit L3–Cache, da erfolglose Zugriffe auf den L2–Cache Zugriffe auf den L3–Cache erzeugen.
6. **L2_DATA_MISSES**: Die erfolglosen Zugriffe auf Daten des Caches werden mit Ereignissen dieses Typs bestimmt.

Von diesen Ereignistypen können die folgenden abgeleitet werden:

1. **L2_INSTR_REFERENCES.d**: Die Anzahl der Zugriffsversuche auf Instruktionen des L2–Caches kann auf zwei Arten berechnet werden:

   $$\texttt{L2\_INSTR\_FETCH} + \texttt{L2\_INSTR\_PREFETCHES}$$

   oder über den Ereignistyp des L1–Caches:

   $$\texttt{L1\_INSTR\_MISSES} + \texttt{L2\_INSTR\_PREFETCHES}$$

2. **L2_TOTAL_REFERENCES.d**: Die Gesamtanzahl von Zugriffen auf den L2–Cache können wie folgt bestimmt werden:

   $$\texttt{L2\_DATA\_REFERENCES} + \texttt{L2\_INSTR\_REFERENCES.d}$$

3. **L2_DATA_WRITES.d**: Der Ereignistyp beschreibt die schreibenden Zugriffe von Daten in den Cache und kann wie folgt berechnet werden:

$$\texttt{L2\_DATA\_REFERENCES} - \texttt{L2\_DATA\_READS}$$

4. **L2_DATA_HITS.d**: Der erfolgreiche Zugriff auf Daten im L2–Cache berechnet sich durch:

$$\texttt{L2\_DATA\_REFERENCES} - \texttt{L2\_DATA\_MISSES}$$

5. **L2_INSTR_MISSES.d**: Der Ereignistyp beschreibt die erfolglosen Zugriffe auf Instruktionen des Caches und wird wie folgt berechnet:

$$\texttt{L2\_MISSES} - \texttt{L2\_DATA\_MISSES}$$

6. **L2_INSTR_HITS.d**: Ähnlich werden die erfolgreichen Zugriffe auf Instruktionen im L2–Cache berechnet:

$$\texttt{L2\_TOTAL\_REFERENCES.d} - \texttt{L2\_MISSES} - \texttt{L2\_DATA\_HITS.d}$$

3.1.1.28 Die Klasse L3_CACHE

Bei einigen Systemen[2] ist ein dritter Cache, der L3–Cache vorhanden. Auch dieser Cache ist wie der L2–Cache ein gemeinsamer Cache für Daten und Instruktionen. Die Klasse enthält ähnliche Ereignistypen wie die anderen Caches.

1. **L3_INSTR_REFERENCES**: Die Lesezugriffe auf Instruktionen im L3–Cache werden durch diesen Ereignistyp beschrieben.
2. **L3_DATA_REFERENCES**: Der Ereignistyp beschreibt Zugriffe auf Daten des L3–Caches.
3. **L3_DATA_READS**: Mit Ereignissen dieses Typs können die lesenden Zugriffe auf den Cache bestimmt werden.
4. **L3_MISSES**: Kann ein Datum nicht im L3–Cache gefunden werden, so treten Ereignisse dieses Typs auf.
5. **L3_READ_MISSES**: Kann auf Daten im L3–Cache nicht lesend zugegriffen werden, so treten Ereignisse dieses Typs auf.
6. **L3_DATA_MISSES**: Zur genaueren Unterscheidung der erfolglosen Zugriffe, beschreibt dieser Ereignistyp die erfolglosen Zugriffe auf Daten.

Durch geeignete Kombination der obigen Ereignisse lassen sich einige weitere Ereignistypen berechnen.

[2] Bisher nur bei den Prozessoren des Herstellers Intel XEON MP und DP, sowie Itanium verfügbar

1. **L3_TOTAL_REFERENCES.d**: Die Anzahl aller Zugriffe auf den L3–Cache läßt sich wie folgt bestimmen:

   $$\texttt{L3\_INSTR\_REFERENCES} + \texttt{L3\_DATA\_REFERENCES}$$

2. **L3_DATA_WRITES.d**: Die schreibenden Zugriffe auf Daten des L3–Caches berechnen sich wie folgt:

   $$\texttt{L3\_DATA\_REFERENCES} - \texttt{L3\_DATA\_READS}$$

3. **L3_DATA_HITS.d**: Die erfolgreichen Zugriffe auf Daten des L3–Caches können wie folgt ermittelt werden:

   $$\texttt{L3\_DATA\_REFERENCES} - \texttt{L3\_DATA\_MISSES}$$

4. **L3_INSTR_MISSES.d**: Die erfolglosen Zugriffe auf Instruktionen des L3–Caches werden durch diesen Ereignistypen beschrieben:

   $$\texttt{L3\_MISSES} - \texttt{L3\_DATA\_MISSES}$$

5. **L3_INSTR_HITS.d**: Der Ereignistyp beschreibt die erfolgreichen Zugriffe auf Instruktionen im L3–Cache und wird wie folgt berechnet:

   $$\texttt{L3\_TOTAL\_REFERENCES.d} - \texttt{L3\_MISSES} - \texttt{L3\_DATA\_HITS.d}$$

6. **L3_WRITE_MISSES.d**: Können Daten nicht in den L3–Cache geschrieben werden, so treten Ereignisse dieses Typs auf. Diese Situation tritt dann auf, wenn Daten im L2–Cache verändert wurden und auch im L3–Cache verändert werden müssen. Ist ein solches Datum bereits aus dem L3–Cache verdrängt worden, so muß es im Speicher über einen Buszugriff verändert werden (Write–Back). Ereignisse dieses Typs werden wie folgt berechnet:

   $$\texttt{L3\_DATA\_MISSES} - \texttt{L3\_READ\_MISSES}$$

3.1.1.29 Die Klasse CACHE_PERFORMANCE

In dieser Klasse werden einige Ereignistypen zusammengefasst, die die Leistung der Caches beschreiben sollen. Alle Ereignistypen sind aus den Ereignistypen der einzelnen Caches abgeleitet. In der Tabelle 3.1 sind diese Ereignistypen dargestellt und in [6] genau beschrieben. Zum Teil sind die Ereignistypen in anderen Systemen anderer Hersteller ebenfalls zu finden, dazu ist Kapitel 4 heranzuziehen.

3.1.2 Prozessorexterner Speicher

Dieser Abschnitt befasst sich mit der Performance–Messung des Hauptspeichers. Die Speichermodule sind im Allgemeinen nicht mit Ereigniszählern ausgestattet, jedoch sollen hier die für die Performance–Analyse nützlichen Ereignistypen theoretisch vorgeschlagen werden. Die Ereignistypen werden in einer Klasse MEMORY_PERFORMANCE zusammengefasst.

Wichtige Cache–Raten

Ereignistyp	Berechnungsvorschrift
L1_INSTR_MISS_RATIO.d	L1_INSTR_MISSES/L1_INSTR_REFERENCES
L1_DATA_MISS_RATIO.d	L1_DATA_MISSES/L1_DATA_REFERENCES
L2_MISS_RATIO.d	L2_MISSES/L2_TOTAL_REFERENCES.d
L2_DATA_MISS_RATIO.d	L2_DATA_MISSES/L2_DATA_REFERENCES
L2_INSTR_MISS_RATIO.d	L2_INSTR_MISSES.d/L2_INSTR_REFERENCES.d
L2_DATA_READ_MISS_RATIO.d	L3_LOAD_REFERENCES/L2_DATA_READS
L2_DATA_WRITE_MISS_RATIO.d	L3_STORE_REFERENCES/L2_DATA_WRITES
L2_INSTRUCTION_FETCH_RATIO.d	L1_INSTR_MISSES/L2_TOTAL_REFERENCES.d
L2_DATA_RATIO.d	L2_DATA_REFERENCES/L2_TOTAL_REFERENCES.d
L3_MISS_RATIO.d	L3_MISSES/L2_MISSES
L3_DATA_MISS_RATIO.d	$\frac{(\text{L3\_READ\_MISSES}+\text{L3\_WRITE\_MISSES.d})}{\text{L3\_DATA\_REFERENCES}}$
L3_INSTR_MISS_RATIO.d	L3_INSTR_MISSES/L3_INSTR_REFERENCES.d
L3_DATA_READ_RATIO.d	L3_LOAD_REFERENCES/L3_DATA_REFERENCES.d
L3_DATA_RATIO.d	L3_DATA_REFERENCES.d/L3_REFERENCES

Tabelle 3.1: Cache–Raten für den L1–, L2– und L3–Cache.

3.1.2.1 Das Interface MEMORY_PERFORMANCE

Das Interface MEMORY_PERFORMANCE bündelt die Klassen MAIN_MEMORY und DISC. In der Klasse werden einige Ereignistypen, den flüchtigen Hauptspeicher betreffend, vorgeschlagen und in der Klasse DISC einige Typen den Festspeicher betreffend angesprochen. In Abbildung 3.9 ist die Hierarchie der Klassen unter dem Interface dargestellt.

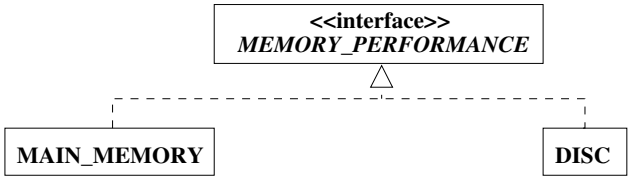


Abbildung 3.9: Die Klassenhierarchie unter dem Interface MEMORY_PERFORMANCE.

3.1.2.2 Die Klasse MAIN_MEMORY

Die Klasse enthält drei Ereignistypen, die Aufschluß über den Datentransfer vom Hauptspeicher zum Prozessor geben könnten.

1. **MEMORY_THROUGHPUT**: Ein Ereignistyp, der den Datendurchsatz in MB/s beschreibt, kann auf Performance–Problem hinweisen.

2. **MEMORY_BURST_MODE**: Ereignisse dieses Typs zählen, wie oft auf den Speicher im so genannten Burst–Mode zugegriffen wurde. Dieser Modus ist im Zugriff schneller als

zum Beispiel der wahlfreie Zugriff auf die Zellen des Hauptspeichers. Der Modus ist detailliert in [18] beschrieben.

3. **MEMORY_PAGE_MODE**: Der Ereignistyp beschreibt, wie oft auf den Speicher im Page–Mode zugegriffen wurde. Zugriffe in diesem Modus gehen ebenfalls schneller vonstatten als wahlfreie Zugriffe, da das Kommando für den Zeilenzugriff nicht wiederholt wird, wenn auf die selbe Zeile zugegriffen wird (siehe hierzu auch [18]).

Meist können Software–Tools wie der Performance–Monitor von Windows 2000/NT der Firma Microsoft diese Angaben messen.

3.1.2.3 Die Klasse DISC

In dieser Klasse wurden zwei Ereignistypen als sinnvoll erachtet. Zum einen ist der Datendurchsatz für die Performance des Systems von Interesse, zum anderen ist es unter bestimmten Umständen wichtig, zu wissen, wie oft ein Interrupt von der Festplatte ausgelöst wurde, da dies den Programmablauf kurzzeitig unterbricht.

Daher kapselt die Klasse die folgenden zwei Ereignistypen:

1. **DISC_THROUGHPUT**: Der Ereignistyp soll den Datendurchsatz in MB/s beschreiben.
2. **DISC_INTERRUPT**: Ereignisse dieses Typs geben an, wie oft der Interrupt der Festplatte ausgelöst wurde.

Diese Ereignistypen fanden sich bei verschiedenen Herstellern nicht als Hardware–Counter implementiert. Performance–Tools, wie der schon erwähnte Performance–Monitor (vergleiche Abschnitt 3.1.2.2), sowie verschiedenste Bench–Marking–Tools können jedoch diese Daten erheben. Es ist allerdings mit diesen Hilfsprogrammen nicht möglich, die Anzahl der Ereignisse den Codeadressen eines in Ausführung befindlichen Programms zuzuordnen und folglich schwierig, den Grund von Performance–Engpässen dieser Art genau zu beschreiben.

3.1.3 Peripherie

In diesem Abschnitt werden mögliche Ereignistypen der übrigen Peripherie eines Rechensystems beleuchtet. Es werden Ereignistypen für

1. Netzwerkadapter und Schnittstellen (Klasse CONNECTION, siehe Abschnitt 3.1.3.1)
2. Graphikadapter (Klasse GRAPHICS, siehe Abschnitt 3.1.3.4)

vorgeschlagen. Die Performance–Eigenschaften eines Programms in speziellen Anwendungsumgebungen sollen durch diese Ereignistypen detaillierter untersucht werden können. In Abbildung 3.10 ist die Klassenhierarchie dargestellt.

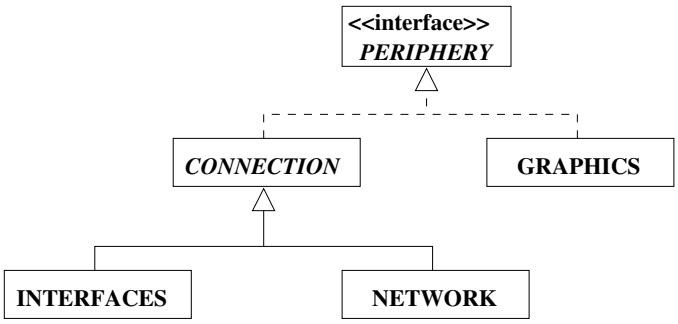


Abbildung 3.10: Die Klassenhierarchie unter dem Interface PERIPHERY.

3.1.3.1 Die abstrakte Klasse CONNECTION

Die Klasse ist die Superklasse der Klassen INTERFACES und NETWORK. Die Vererbungshierarchie wurde so gewählt, da die meisten vorgeschlagenen Ereignistypen sowohl für Peripherieschnittstellen (Klasse INTERFACES, siehe Abschnitt 3.1.3.3), als auch für Netzwerkadapter (Klasse NETWORK, siehe Abschnitt 3.1.3.2) sinnvoll sind. Die Ereignistypen lauten wie folgt:

1. **CONNECTION_BYTES**: Dieser Ereignistyp soll die Bestimmung der gesendeten und empfangenen Bytes beschreiben.
2. **CONNECTION_BYTES_RECEIVED**: Die Anzahl von empfangenen Bytes beschreibt dieser Ereignistyp.

Durch Kombination der Ereignistypen CONNECTION_BYTES und CONNECTION_BYTES_RECEIVED kann der folgende abgeleitete Ereignistyp ermittelt werden:

1. **CONNECTION_BYTES_SENT.d**: Die gesendeten Bytes ergeben sich aus der Subtraktion von CONNECTION_BYTES_RECEIVED von CONNECTION_BYTES.

Der Durchsatz pro Sekunde kann leicht bestimmt werden, wenn die Werte der jeweiligen Ereigniszähler durch die Länge des Sampling–Intervalls geteilt werden.

3.1.3.2 Die Klasse NETWORK

NETWORK ist eine Subklasse von CONNECTION und erbt daher alle dort beschriebenen Ereignistypen. Untersuchungen in [16] zeigen, wie durch Kombination der Ereignistypen dieser Klasse mit dem Ereignistypen PROCESSOR_PROCESSOR_TIME.d in Abschnitt 3.2.1.1 das Antwortzeitverhalten von Servern auf Client–Anfragen gemessen werden kann. Ein weiterer Ereignistyp ist dafür jedoch noch vonnöten:

1. **NETWORK_REQUESTS**: Mit Ereignissen dieses Typs soll die Anzahl von Anfragen, die durch den Netzwerkadapter verarbeitet werden, beschrieben werden.

3.1.3.3 Die Klasse INTERFACES

Diese Klasse ist ebenso wie die Klasse NETWORK Subklasse von CONNECTION. Mit ihr werden die Ereignistypen für die Peripherieschnittstellen modelliert. Sie enthält allerdings keine weiteren Ereignistypen als die ererbten.

3.1.3.4 Die Klasse GRAPHICS

Bei Graphikanwendungen, welche bewegte Bildsequenzen berechnen, sind Daten über die Bildrate pro Sekunde und Anzahl berechneter Objekte von Interesse. Die folgenden Attribute der Klasse werden vorgeschlagen, um auf modernen Graphikadaptern die Performance der Darstellung ermitteln zu können. Die Ereignistypen werden in [28] erwähnt:

- **GRAPHICS_FRAMES**: Der Ereignistyp bestimmt die Anzahl der dargestellten Bilder pro Sekunde.
- **GRAPHICS_OBJECTS**: Die Anzahl der Objekt pro dargestelltem Bild bestimmt die Geschwindigkeit der Bildaufbaus.
- **GRAPHICS_FRAMES**: Die Anzahl der dargestellten Frames bedeutet Rechenaufwand. Mit Ereignissen dieses Typs soll diese Anzahl zu bestimmen sein.

Die Klasse enthält einen weiteren abgeleiteten Ereignistypen.

- **GRAPHICS_OBJECTS_PER_FRAME.d**: Die durchschnittliche Anzahl der dargestellten Objekte pro Frame bestimmt die Geschwindigkeit des Bildaufbaus und berechnet sich wie folgt:

$$\frac{\texttt{GRAPHICS\_OBJECTS}}{\texttt{GRAPHICS\_FRAMES}}$$

3.2 Software

Dieser Abschnitt widmet sich den Ereignistypen beziehungsweise Ereigniszählern, die das Betriebssystem zur Verfügung stellen könnte. Wie auch im vorhergehenden Abschnitt wird eine allgemeine Klassifizierung der vorgeschlagenen Ereignistypen angegeben. In Kapitel 4 wird dann auf die tatsächlich implementierten Typen eingegangen. In Abbildung 3.11 ist die allgemeine Struktur eines Betriebssystems zu sehen [20], anhand derer die Klassifizierung aufgestellt wird.

Es werden vier Interfaces identifiziert, die die Klassifizierung zusammenfassen:

- **Process Management**: Unter dem Interface PROCESS_MANAGEMENT in Abschnitt 3.2.1 befinden sich die Klassen PROCESSOR und die abstrakte Klasse PROCESS_THREAD. In der Klasse PROCESSOR beschreiben die Ereignistypen den Zustand des Prozessors im Allgemeinen. Die Ereignistypen der abstrakten Klasse PROCESS_THREAD beschreiben Eigenschaften der Prozesse und Threads.

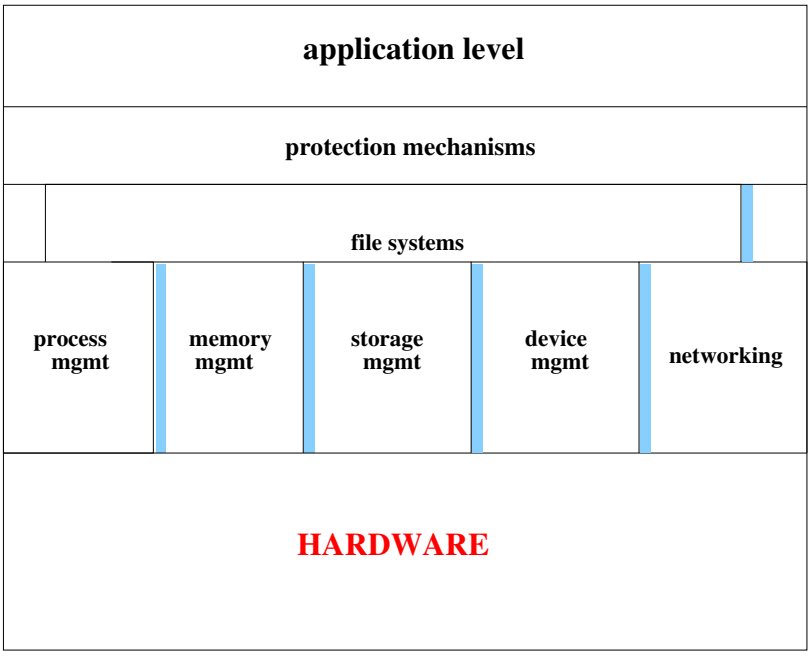


Abbildung 3.11: Allgemeine Struktur eines Betriebssystems.

- **Memory Management**: Ereignistypen für Seitenspeicher und realen Speicher werden in den Klassen `PAGED` und `REAL` unter dem Interface `MEMORY_MANAGEMENT` in Abschnitt 3.2.3 zusammengefasst.
- **Storage Mangement**: Das Interface `STORAGE_DEVICE_MANAGEMENT` in Abschnitt 3.2.2 beherbergt die Klasse `STORAGE_MANAGEMENT`, welche sich mit Ereignistypen für Festplatten oder anderen angeschlossenen Plattenspeichern befasst. Des weiteren steht unter dem Interface die Klasse `DEVICE_MANAGEMENT`, die Ereignistypen beinhaltet, welche Operationen auf Dateisystemen beschreiben.
- **Device Management**: Das Interface `STORAGE_DEVICE_MANAGEMENT` beinhaltet die Ereignistypen dieses Teils des Betriebssystems.
- **Networking**: Mit dem Interface `NETWORK` in Abschnitt 3.2.4 werden Ereignistypen dieses Teils eines Betriebssystems erfasst. Die drei Klassen `READS`, `WRITES` und `CONNECTIONS` implementieren dieses Interface.

In Anhang A.2 sind alle im Folgenden behandelten Klassen unter dem Interface `SOFTWARE` zusammengefasst. Die Abbildung zeigt die Klassenhierarchie in der Gesamtübersicht. Die Abschnitte 3.2.1 bis 3.2.4 behandeln im Folgenden die Teile der Abbildung im Einzelnen.

3.2.1 Das Interface `PROCESS_MANAGEMENT`

Ereignistypen, die unter diesem Interface in den verschiedenen Klassen gesammelt sind, befassen sich mit der Prozessor und Prozessverwaltung eines Betriebssystems. In Abbildung

3.12 ist die Klassenhierarchie abgebildet, welche in den nächsten Unterabschnitten erläutert wird.

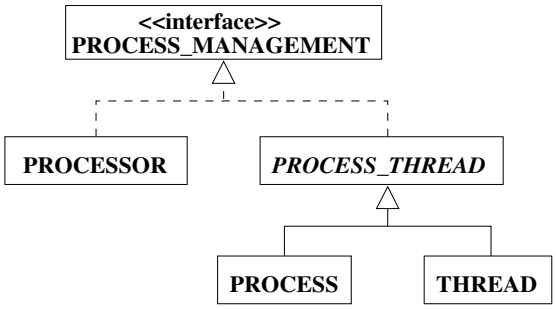


Abbildung 3.12: Die Klassen der Prozessor– und Prozessverwaltung unter dem Interface `PROCESS_MANAGEMENT`.

3.2.1.1 Die Klasse `PROCESSOR`

In dieser Klasse sind Ereignistypen zusammengefasst, die über das Verhalten des Prozessors Auskunft geben. Sie lauten wie folgt:

1. **`PROCESSOR_INTERRUPTS`**: Dieser Ereignistyp beschreibt die Anzahl der vom Prozessor behandelten Interrupts. Diese Interrupts werden dann erzeugt, wenn ein Peripheriegerät Aufmerksamkeit verlangt.

2. **`PROCESSOR_PRIVILEGED_MODE`**: Ereignisse dieses Typs treten auf, wenn der Prozessor im privilegierten Modus operierte. Dazu zählen System–Calls und die Behandlung von Interrupts. Werden sehr viele Ereignisse dieses Typs erzeugt, so kann dies auf ein fehlerhaftes Gerät hinweisen, welches viele Interrupts erzeugt.

3. **`PROCESSOR_PRIVILEGED_TIME`**: Ereignisse dieses Typs berechnen die Gesamtzeit in der sich der Prozessor im privilegierten Modus während des Sampling–Intervalls befand.

4. **`PROCESSOR_USER_MODE`**: Mit diesem Ereignistyp kann ermittelt werden, wie oft sich der Prozessor im Benutzermodus befand, das heißt Benutzerprozesse beziehungsweise Benutzer–Threads ausführte.

5. **`PROCESSOR_USER_TIME`**: Dieser Ereignistyp beschreibt die Zeit, in der sich der Prozessor im Benutzermodus während des Sampling–Intervalls befand.

6. **`PROCESSOR_QUEUE_LENGTH`**: Dieser Ereignistyp beschreibt die Anzahl von rechenbereiten Threads in der Warteschlange. Im Allgemeinen ist ein Wert größer zwei ein Hinweis für die Blockierung der Warteschlange [17].

Aus den oben genannten Ereignistypen lassen sich zwei weitere Ereignistypen ableiten:

1. **PROCESSOR_PROCESSOR_TIME.d**: Die Zeit in der der Prozessor entweder auf Interrupts, beziehungsweise System–Calls reagierte, oder Benutzerprogramme ausführte kann wie folgt beschrieben werden:

   $$\texttt{PROCESSOR\_PRIVILEGED\_TIME} + \texttt{PROCESSOR\_USER\_TIME}$$

   Ereignisse dieses Typs bestimmen damit die Zeit in der nicht der idle–Thread ausgeführt wurde.

2. **PROCESSOR_PROCESSOR_MODE.d**: Wie oft der Prozessor nicht den idle–Thread ausführte kann analog berechnet werden:

   $$\texttt{PROCESSOR\_PRIVILEGED\_MODE} + \texttt{PROCESSOR\_USER\_MODE}$$

3.2.1.2 Die abstrakte Klasse PROCESS_THREAD

Diese abstrakte Klasse beschreibt Ereignistypen, die für Prozesse und Threads gleichermaßen interessant sind. Von ihr erben die Klassen PROCESS und THREAD. In den Unterklassen beziehen sich die Ereignistypen der abstrakten Klasse dann jeweils auf die Beschreibung von Prozessen und Threads im Speziellen, wobei zu beachten ist, daß Prozessereignistypen Werte von Thread–Ereignistypen summieren. Die abstrakte Klasse enthält folgende Ereignistypen:

1. **PROCESS_THREAD_CONTEXT_SWITCHES**: Mit diesem Ereignistyp wird die Anzahl der Kontextwechsel beschrieben. Kontextwechsel bedeuten die Unterbrechung des aktuell laufenden Prozesses oder Threads.

2. **PROCESS_THREAD_SYS_CALLS**: Der Ereignistyp beschreibt die Anzahl der Systemaufrufe in Prozessen oder Threads. Jeder Systemaufruf unterbricht die Ausführung eines Prozesses beziehungsweise eines Threads. Eine gehäufte Anzahl dieser Aufrufe verlangsamt also das laufende Programm.

3. **PROCESS_THREAD_EXCEPTIONS**: Mit diesem Ereignistyp soll bestimmt werden können, wie viele Ausnahmebehandlungen ein Prozess oder Thread behandeln mußte.

4. **PROCESS_THREAD_PAGE_FAULTS**: Der Ereignistyp beschreibt die Seitenzugriffsfehler die durch den Prozess oder Thread verursacht wurden.

5. **PROCESS_THREAD_IO_DATA_OPS**: Über Ereignisse dieses Typs wird die Anzahl der Lese– oder Schreiboperationen auf I/O–Geräte wie Festplatten und Netzwerkkarten bestimmt.

6. **PROCESS_THREAD_IO_READS**: Ereignisse dieses Typs bestimmen die Anzahl der Leseoperationen auf I/O–Geräte.

7. **PROCESS_THREAD_IO_OTHER_OPS**: Mit Ereignissen dieses Typs soll die Anzahl von Operationen auf I/O–Geräten, die keine Schreib– oder Leseoperationen darstellen, bestimmt werden.

Es kann aus den obigen Ereignistypen wiederum ein weiterer Ereignistyp abgeleitet werden:

1. **PROCESS_THREAD_IO_WRITES.d**: Die Anzahl der Schreiboperationen auf I/O–Geräte ergibt sich wie folgt:

$$\texttt{PROCESS\_THREAD\_IO\_DATA\_OPS} - \texttt{PROCESS\_THREAD\_IO\_READS}$$

3.2.1.3 Die Klasse PROCESS

Diese Klasse beschreibt Ereignistypen, die für die Beobachtung von Prozessen wichtig sind. Die Klasse erbt alle Ereignistypen der abstrakten Superklasse und enthält einen weiteren Ereignistyp. Ein wichtiges Attribut der Ereignisse dieses Typs ist die Prozess–ID. Sie macht die Zuordnung des Auftretens von Ereignissen dieses Typs zum Prozess möglich.

1. **PROCESS_THREADS**: Ereignisse dieses Typs geben die Anzahl von Threads in einem Prozess an.

3.2.1.4 Die Klasse THREAD

Diese Klasse fügt den ererbten Ereignistypen der abstrakten Superklasse spezielle Ereignistypen hinzu, die insbesondere für Threads eines Prozesses von Interesse sind. Auch hier ist die Prozess–ID ein wichtiges Attribut der Ereignisse der Ereignistypen dieser Klasse. Mit ihr sind die beobachteten Ereignisse der Threads den Prozessen zuzuordnen.

1. **THREAD_SYNC**: Ereignisse dieses Typs treten auf, wenn zwei oder mehr Threads die Ausführung synchronisieren wollen. Eventuell muß ein Thread auf einen anderen warten, so daß sich die Ausführung verzögern kann.
2. **THREAD_MUTEXES**: Mutexe werden eingesetzt, wenn sich Threads gemeinsame Codeabschnitte, Daten oder Ressourcen teilen, die aber nicht gemeinsam benutzt werden dürfen. Je nach Betriebssystem ist die Implementierung unterschiedlich. Da Mutexe aber in die Ablaufsteuerung der Threads eingreifen, müssen sie vom Betriebssystem zur Verfügung gestellt werden [11]. Ereignisse dieses Typs treten dann auf, wenn das Betriebssystem einen Mutex erzeugt.

3.2.2 Das Interface STORAGE_DEVICE_MANAGEMENT

Die Ereignistypen welche das Interface zusammenfasst, beschreiben Zähler die das Verhalten von Festplatten oder anderen Speichermedien und dem Dateisystem beschreibt. Die Klasse STORAGE_MANAGEMENT soll dabei Ereignistypen für Dateisysteme enthalten. Die Klasse DEVICE_MANAGEMENT hingegen, enthält Ereignistypen, die die Performance der Speichergeräte beschreiben. In Abbildung 3.13 ist die Klassenhierarchie gezeigt.

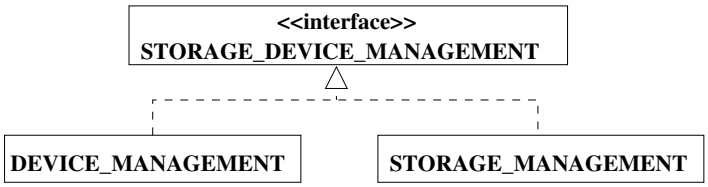


Abbildung 3.13: Die Klassen der Dateisystem und Speichermedien unter dem Interface `STORAGE_DEVICE_MANAGEMENT`.

3.2.2.1 Die Klasse `DEVICE_MANAGEMENT`

Diese Klasse enthält Ereignistypen, die die Performance von Speichergeräten beschreiben. Ein Attribut, welches die Gerätenummer oder Ähnliches enthält, ist sinnvoll, wenn die aufgetretenen Ereignisse den Geräten zugeordnet werden sollen.

1. **`DEVICE_REQUESTS`**: Ereignisse dieses Typs treten auf, wenn Schreib– oder Leseanforderungen an ein Speichergerät gestellt werden.
2. **`DEVICE_REQU_OUTSTANDING`**: Ereignisse dieses Typs beschreiben die Anzahl der noch nicht behandelten Schreib– oder Leseanforderungen. Ereignisse dieses Typs sollten die unbehandelten Anfragen nicht summieren, sondern einen momentanen Zustand wiedergeben, da zwischen den Sampling–Intervallen einige Anfragen behandelt worden sein könnten. In [17] wird ein Maß für die Verzögerung der Bearbeitung angegeben. Ist die Anzahl `spindles` der Platten einer Festplatte bekannt, so ist die Bearbeitungsverzögerung `delay` direkt proportional zu folgender Differenz:

   $$\texttt{delay} \approx \texttt{DEVICE\_REQU\_OUTSTANDING} - \texttt{spindles}$$

   Auch wird angegeben, daß ein Wert kleiner zwei eine geringe Verzögerung bedeutet.
3. **`DEVICE_REQ_READ`**: Werden Anfragen für Lesezugriffe an das Gerät gestellt, so treten Ereignisse dieses Typs auf.
4. **`DEVICE_SPLIT_IO`**: Die Anzahl der Datenübertragungen, die nicht mit einem Zugriff erfolgen konnten, beschreibt dieser Ereignistyp. Ereignisse dieses Typs treten auf, wenn entweder die Datenmenge zu groß war, oder das Speichermedium fragmentiert ist, so daß Positionierungen der Schreib–/Lesemechanik vonnöten waren.

Der aus den genannten Ereignistypen ableitbare Ereignistyp ist im folgenden angegeben.

1. **`DEVICE_REQ_WRITE.d`**: Die Anzahl der Schreibzugriffe kann bestimmt werden durch:

   $$\texttt{DEVICE\_REQUESTS} - \texttt{DEVICE\_REQ\_READ}$$

3.2.2.2 Die Klasse `STORAGE_MANAGEMENT`

In dieser Klasse werden Ereignistypen zusammengefasst, die Zugriffe auf das Dateisystem beschreiben. Somit steht nicht das Gerät wie in Klasse `DEVICE_MANAGEMENT` im Vordergrund, sondern das Dateisystem. Im folgenden sind die Ereignistypen beschrieben.

1. **STORAGE_CONTROL_OPS**: Die Anzahl der Zugriffe die keine Lese– oder Schreibzugriffe sind werden mit Ereignissen dieses Ereignistyps ermittelt.
2. **STORAGE_DATA_OPS**: Die Anzahl aller lesenden und schreibenden Zugriffe auf Daten können durch Ereignisse dieses Typs gezählt werden.
3. **STORAGE_FILE_READ_OPS**: Die Anzahl der lesenden Zugriffe auf Dateien bestimmen Ereignisse dieses Typs.

Aus den oben genannten zwei Ereignistypen läßt sich ein weiterer Typ ableiten:

1. **STORAGE_FILE_WRITE_OPS.d**: Die Anzahl der schreibenden Zugriffe auf Dateien werden wie folgt bestimmt:

$$\texttt{STORAGE\_DATA\_OPS} - \texttt{STORAGE\_FILE\_READ\_OPS}$$

3.2.3 Das Interface MEMORY_MANAGEMENT

Das Interface umfasst Ereignistypen welche die Speicherverwaltung im Betriebssystem beschreiben. Moderne Betriebssysteme bilden den physikalischen Speicher auf virtuelle Speicherseiten ab, von denen weit mehr existieren können als der tatsächlich vorhandene Speicher zulässt. Aus diesem Grund umfasst dieses Interface die Klasse PAGED, welche Ereignistypen für die Seitenverwaltung beinhaltet.

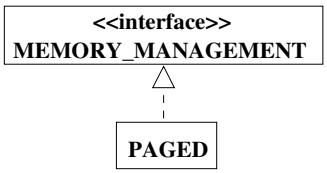


Abbildung 3.14: Die Klasse der virtuellen Speicherverwaltung unter dem Interface MEMORY_MANAGEMENT.

3.2.3.1 Die Klasse PAGED

Ereignistypen dieser Klasse beschäftigen sich mit der Anzahl Ein– und Ausgelagerter Speicherseiten, sowie mit Seitenfehlern. Im Folgenden sind die Ereignistypen beschrieben:

1. **PAGED_PAGE_FAULTS**: Ereignisse dieses Typs treten auf, wenn Zugriffe auf Speicherseiten erfolgen, die sich nicht im aktuellen Arbeitsbereich des Programms befinden. Es werden zwei Arten von Seitenfehlern unterschieden, die so genannten "hard faults" und die "soft faults". Müssen die Seiten vom Festspeicher in den Hauptspeicher übertragen werden, so wird von einem "hard fault" gesprochen. Kann die Seite im Hauptspeicher gefunden werden, so wird ein solcher Seitenfehler als "soft fault" bezeichnet. Ereignisse dieses Typs zählen das Auftreten beider Arten.

2. **PAGED_PAGES_IN**: Ereignisse dieses Typs bestimmen die Anzahl der Seiten, die das Betriebssystem auf der Festplatte unter den ausgelagerten Seiten suchen mußte um einen Seitenfehler behandeln zu können. Diese Ereignisse treten demnach bei "hard faults" auf und sind ein Indikator für systemweite Verzögerungen, da ein Zugriff auf ein langsames Speichermedium erfolgt.

3. **PAGED_PAGES_OUT**: Mit Ereignissen dieses Typs wird die Anzahl der auf ein langsames Speichermedium ausgelagerten Seiten beschrieben. Ereignisse dieses Typs treten dann auf, wenn der physikalische Speicher knapp wird und, je nach Seitenersetzungsalgorithmus, Seiten verdrängt werden müssen, um physikalischen Speicherplatz zur Verfügung zu stellen. Die Auslagerung von Seiten auf langsame Speichermedien nimmt vergleichsweise viel Zeit in Anspruch und verursacht systemweite Verzögerungen.

Durch Kombination der oben genannten Ereignistypen kann ein weiterer abgeleitet werden:

1. **PAGED_PAGE_FAULTS_SOFT.d**: Die Anzahl der "soft faults" ergibt sich wie folgt:

$$\texttt{PAGED\_PAGE\_FAULTS} - \texttt{PAGED\_PAGES\_IN}$$

3.2.4 Das Interface NETWORKING

Die Ereignistypen, welche unter diesem Interface zusammengefasst sind, beschreiben das Verhalten der Netzwerkkomponenten des Betriebssystems. Beispielsweise sollen die Anzahl der empfangenen Bytes oder die Anzahl der nach einem Abbruch wiederholten Verbindungen zu einem Server beobachtet werden können. Vornehmlich sind diese Ereignistypen für beispielsweise Browser–Programme oder FTP–Clients und Server wichtig. Die Abbildung 3.15 zeigt die Hierarchie der Klassen READS, WRITES und CONNECTIONS unter denen die vorgeschlagenen Ereignistypen aufgeteilt sind.

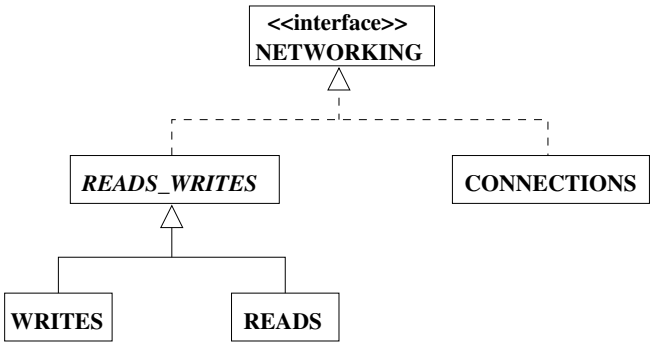


Abbildung 3.15: Die Klassen der Netzwerkkomponente eines Betriebssystems unter dem Interface NETWORKING.

3.2.4.1 Die abstrakte Klasse `READS_WRITES`

Diese abstrakte Klasse enthält Ereignistypen, welche die ein– und ausgehenden Daten des Netzwerksubsystems beschreiben. Von ihr werden die konkreten Klassen `READS` und `WRITES` abgeleitet, welche alle Ereignistypen der abstrakten Superklasse erben. In den konkreten Klassen beschreiben die ererbten Typen die eingehenden respektive die ausgehenden Daten im Speziellen. Folgende Ereignistypen sind in der Klasse enthalten:

1. **`READS_WRITES_DENIED`**: Wird eine Anfrage an den Server gestellt, welche zurückgewiesen wird, so tritt ein Ereignis dieses Typs auf. Diese Situation kann beispielsweise auftreten, wenn ein Client mit einer Leseanfrage die ausgehandelte Puffergröße des Servers überschreitet. Der Server muß dann andere Anfragen abweisen, um Ressourcen für diese zur Verfügung stellen zu können.
2. **`READS_WRITES_EXCEEDING_BUF_SIZE`**: Ereignisse dieses Typs beschreiben die Anzahl der Anfragen an einen Server, welche die ausgehandelte Puffergröße überschreiten. Solche Anfragen können die Ressourcen des Servers überbeanspruchen und zu einem verminderten Antwortzeitverhalten des Servers führen.
3. **`READS_WRITES_UNDERRUN_BUF_SIZE`**: Werden Anfragen gestellt, welche die ausgehandelte Puffergröße unterschreiten, so ist dies ein Indiz für die Verschwendung von Ressourcen des Servers.
4. **`READS_WRITES_BYTES_RECEIVED`**: Die Anzahl der gesendeten oder empfangenen Bytes soll durch diesen Ereignistypen beschrieben werden.

3.2.4.2 Die Klasse `READS`

In dieser Klasse sind Ereignistypen zusammengefasst, die sich mit den eingehenden Daten des Netzwerksubsystems eines Betriebssystems beschäftigen. Von der abstrakten Überklasse werden alle dort beschriebenen Ereignistypen geerbt.

3.2.4.3 Die Klasse `WRITES`

Die ausgehenden Daten des Netzwerksubsystems sollen mit diesen Ereignistypen beobachtbar sein. Die Klasse erbt alle Ereignistypen der abstrakten Überklasse.

3.2.4.4 Die Klasse `CONNECTIONS`

Die Ereignistypen dieser Klasse beschreiben Eigenschaften von Netzwerkverbindungen im Allgemeinen. Folgende Typen sind in der Klasse enthalten:

1. **`CONNECTIONS_REQUESTS`**: Die Ereignisse dieses Typs zählen die Anzahl der gestellten Lese– und Schreibanfragen.
2. **`CONNECTIONS_PACKETS`**: Die Anzahl der übertragenen Pakete soll durch Ereignisse dieses Ereignistyps beschrieben werden.

3. **`CONNECTIONS_SERVER_RECONNECTS`**: Wird die Verbindung zu einem Server abgebrochen und muß wegen einer folgenden Anfrage erneut aufgebaut werden, so tritt ein Ereignis dieses Typs auf. Eine erneute Verbindungsaufnahme wird als teure Operation bezeichnet [17].

4 Implementierung der Klassen

Die vorgeschlagene Klassifizierung von Ereignistypen in Kapitel 3 hat zum Ziel, für verschiedene Prozessoren und Peripheriegeräte auf der Seite der Hardware und verschiedene Betriebssysteme auf der Seite der Software eine allgemeine Einteilung der Ereignistypen darzustellen. Diese Kapitel zeigt in wie fern sich die Ergebnisse des vorherigen Kapitels auf konkrete Prozessortypen und Betriebssysteme anwenden lassen. Auf der Seite der Hardware werden die Prozessoren Itanium und P6 des Herstellers Intel und der PowerPC 604e des Herstellers IBM betrachtet. Die Wahl fiel auf diese Prozessoren, um ein möglichst breites Spektrum der Technologien abzudecken. Die drei Prozessoren lassen sich in drei Architekturklassen einteilen, auf die die Klassifizierung anwendbar ist. Auf der einen Seite stehen die CISC–Architekturen mit einer Vielzahl unterschiedlich langer und komplexer Befehlswörter. Der Maschinencode eines Programms auf diesen Prozessoren ist folglich sehr kompakt, die Ausführung bedarf aber eines sehr komplexen Leit– und Rechenwerkes. Auf der anderen Seite stehen die RISC–Architekturen mit einem einfachen und kleinen Befehlssatz. Da eine CISC–Instruktion mit mehreren RISC–Instruktionen vergleichbar ist, ist der Maschinencode der Programme größer, jedoch wird kein komplexes Leit– und Rechenwerk benötigt. Die Seite der CISC–Architekturen vertritt in diesem Kapitel der Intel P6, die Seite der RISC–Architekturen der IBM PowerPC 604e. Beide Seiten werden in einer dritten Architekturklasse durch den 64–Bit Prozessor Itanium vereint – einem Gemeinschaftsprojekt der Hersteller Intel und HP. Die Hardware–Performance–Zähler diese Prozessors bildeten den Ausgangspunkt der aufgestellten Klassifizierung.

Die in Kapitel 3, Abschnitt 3.2 aufgestellte Einteilung der Ereignistypen der Hardware wird an den Betriebssystemen Windows 2000 der Firma Microsoft und Linux der Distribution Debian untersucht.

4.1 Vorstellung der betrachteten Prozessoren

In den nachfolgenden Abschnitten werden die betrachteten Prozessoren kurz vorgestellt. Dabei liegt das Hauptaugenmerk auf den Fähigkeiten der Prozessoren zur Performance–Messung.

4.1.1 Intel IA–64 Itanium

Der Itanium–Prozessor stellt vier 32–Bit Performance–Zähler zur Verfügung, mit denen insgesamt über 50 Ereignistypen zu messen sind. Des Weiteren stellt dieser Prozessor verschiedene erweiterte Möglichkeiten der Ereignisüberwachung zur Verfügung, auf die im Folgenden eingegangen wird.

Die folgenden Unterabschnitte beschreiben kurz die Zählerregister und ihre Besonderheiten, um einen Überblick über die Fähigkeiten des Prozessors zu geben. Die Beschreibung ist jedoch nicht umfassend. Für eine vollständige Beschreibung sei auf [5, 6] hingewiesen.

4.1.1.1 Die Performance Counter Register

Die folgende Liste stellt in aller Kürze die Registerpaare für die hardwareunterstützte Ereignismessung dar.

- Es existieren vier Registerpaare: PMC/PMD[4,5,6,7].
- Die Register haben eine Länge von 64 Bit.
- Das jeweilige Register PMC dient der Konfiguration des dazugehörigen Registers PMD, welches die Ereignisse zählt. Bei zählenden Registern werden nur die unteren 32 Bit verwendet, ein carry–out von Bit 31 ist als Überlauf definiert.
- Der Überlauf eines Zählers kann mittels bestimmter Bits des zugehörigen Konfigurationsregisters einen Interrupt erzeugen oder nicht. Die Anzahl der bis zum Überlauf gemessenen Ereignisse kann durch Initialisierung des Zählerregisters eingestellt werden. Der Initiaisierungswert berechnet sich wie folgt:

  $$i = n - c + 1$$

  Wobei n die Breite des Zählerregisters und c die Anzahl zu messender Ereignisse bis zum Überlauf ist. Diese Eigenschaft unterstützt in besonderer Weise das Event–Based–Sampling (siehe Abschnitt 2.3.2).
- Die Zähler sind nicht symmetrisch, das heißt nicht jedes überwachbare Ereignis kann mit jedem Zähler gemessen werden. Die Registerpaare PMC/PMD[4,5] zählen Ereignisse, die höchstens sieben Mal pro Taktzyklus auftreten können, die Registerpaare PMC/PMD[6,7] zählen hingegen nur solche Ereignisse, die höchstens drei Mal pro Taktzyklus auftreten.

4.1.1.2 Überlauf des Ereigniszählers

Im vorherigen Abschnitt wurde erwähnt, daß überlaufende Zähler derart konfiguriert werden können, daß ein Interrupt erzeugt wird. Der Itanium besitzt die vier Statusregister PMC[0,1,2,3] die den Überlauf der Zähler näher beschreiben. Nur das Register PMC[0] wird benutzt, alle anderen ignoriert.

Das Register PMC[0] hat folgende Eigenschaften:

- Das Register ist 64 Bit breit.
- Das Bit 0 ist das so genannte "freeze–bit". Es wird gesetzt, wenn ein Register überläuft und dieses konfiguriert ist, einem Interrupt zu erzeugen. Das Setzten des Bits unterbricht die Zählung weiterer Ereignisse in diesem Register. Das Bit wird durch die Hardware gesetzt, die Software ist für die Löschung verantwortlich.

- Die Bits 4 bis 7 geben an, welches spezielle der vier Zählerregister übergelaufen ist. Die Werte der Bits bleiben erhalten, so daß mehrere Bits gleichzeitig gesetzt sein können.
- Die übrigen Bits werden ignoriert.

4.1.1.3 Ereignisqualifizierung

Die Performance–Überwachung kann beim Itanium–Prozessor auf eine Teilmenge aller Ereignisse eingeschränkt werden. Wie in Abbildung 4.1 dargestellt, kann die Zählung von Ereignissen auf Codebereiche, Instruktionsbereiche, Datenbereiche und die Privilege–Level eingeschränkt werden [6].

- **Das Überwachen spezieller Codebereiche:** Mit dem Itanium–Prozessor ist es möglich, die Ereigniszählung auf bestimmte Codebereiche einzuschränken. Diese Codebereiche werden in den vier "Instruction Breakpoint Registers" (IBR) eingestellt. Tritt ein Ereignis auf, so wird die Adresse der Instruktion mit den Werten dieser vier Register verglichen. Das Ereignis wird nur dann gezählt, wenn die die Instruktionsadresse innerhalb der Adressbereiche liegt, welche die IBRs angeben. Mit Hilfe des Registers PMC[13] kann diese Funktion durch Löschen oder Setzen des ersten Bits an– oder abgeschaltet werden. Mit dieser Eigenschaft kann die Ereignismessung zum Beispiel auf einzelne Module eingeschränkt werden, sie ist allerdings nur für IA–64–Instruktionen verfügbar. IA–32–Instruktionen werden ignoriert.
- **Das Überwachen einzelner Instruktionen:** Die "Opcode Match Register" PMC[8,9] beschränken die Ereigniszählung auf einzelne Instruktionen oder Instruktionstypen. Übereinstimmungen mit diesen Registern können ihrerseits in den Zählerregistern als Ereignisse gezählt werden. So ist es einerseits möglich, Histogramme der Ausführungshäufigkeiten bestimmter Befehle über den gesamten Code oder – in Verbindung mit der Einschränkung auf Codebereiche – über Abschnitte dessen zu erstellen. Andererseits ist es mit dieser Funktion möglich das Auftreten von Ereignissen den Instruktionen zuzuordnen, indem zum Beispiel nur die erfolglosen Zugriffe auf den L1–Cache gezählt werden, die durch eine bestimmte Instruktion erzeugt wurden. Es ist mit dem Itanium–Prozessor möglich jede einzelne Instruktion zu verfolgen.
- **Die Beschränkung der Ereignismessung auf Datenbereiche:** Der Itanium–Prozessor erlaubt die Einschränkung der Ereignisüberwachung für Speicherzugriffe auf bestimmte Datenbereiche. Damit ist es zum Beispiel möglich erfolglose Zugriffe auf den Daten–Cache für spezifische Datenstrukturen zu erfassen. Die Adressbereichsprüfung wird zum Beispiel auf alle Ladeoperationen und Semaphor–Instruktionen angewendet.
- **Die Auswahl von Teilereignissen:** Vermittels der "Event Specific Unit Mask" ist es möglich Details über das beobachtete Ereignis in Erfahrung zu bringen. Als Beispiel diene das Ereignis `BR_PATH_PREDICTION`. Dieser Ereignistyp zählt die Verzweigungen basierend auf der Richtung (taken/not taken) und dem Ausgang der Vorhersage. Über die "Event Specific Unit Mask" können verschiedenste Kombinationen ausgewählt werden, zum Beispiel die Zählung aller Taken–Branches, welche nicht korrekt

vorhergesagt wurden. Die "Event Specific Unit Mask" ist nicht für alle beobachtbaren Ereignisse verfügbar.

- **Beschränkung der Ereigniszählung auf das Privilege–Level:** Die Zählung von Ereignissen kann auf das aktuelle Privilege–Level beschränkt werden. Ohne Unterstützung des Betriebssystems ist es möglich die Ereigniszählung auf den Benutzermodus oder den Systemmodus zu beschränken.
- **Beschränkung der Ereigniszählung auf den Instruktionssatz:** Der Itanium–Prozessor unterstützt die Ausführung sowohl von IA–64 Instruktionen als auch von IA–32 Instruktionen. Durch die Möglichkeit der getrennten Ereigniszählung für diese Instruktionsmengen ist es möglich, die Verteilung der ausgeführten Befehle oder aufgetretenen Ereignisse auf den jeweiligen Instruktionssatz zu ermitteln.

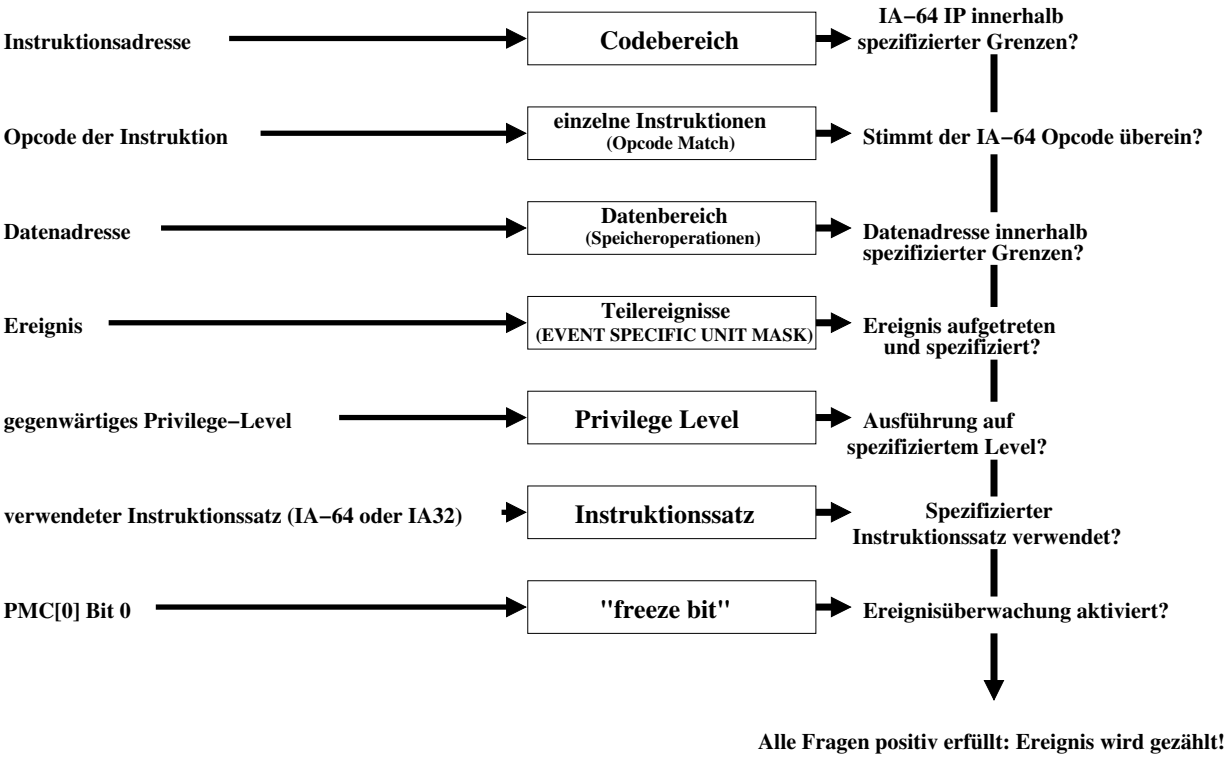


Abbildung 4.1: Möglichkeiten der Ereignisqualifizierung mit dem Intel Itanium.

4.1.2 Intel IA–32 P6

Die Intel P6 Prozessorfamilie besitzt zwei 40 Bit Zählerregister, mittels derer zwei Ereignistypen gleichzeitig beobachtet werden können. Die Möglichkeiten der Ereigniszählung entsprechen im Wesentlichen denen des Itanium (siehe dazu Abschnitt 4.1.1). Im Folgenden werden die Möglichkeit kurz aufgeführt und erklärt. Eine detaillierte Beschreibung ist in [3] zu finden.

4.1.2.1 Die Performance Counter Register

Die Ereignismessung wird in der Hardware beim Intel P6–Prozessor durch die im Folgenden erläuterten Register ermöglicht.

- Es existieren zwei Registerpaare PerfEvtSel/PerfCtr[0,1].
- Die Register PerfEvtSel[0,1] dienen der Konfiguration der dazugehörigen Zählerregister PerfCtr[0,1] und kontrollieren die Bedingungen für die Ereigniszählung. Die Registerbreite beträgt 32 Bit.
- Die Zählerregister PerfCtr[0,1] haben eine Breite von 40 Bit. Nur die niederwertigen 32 Bit können mit Werten beschrieben werden, die oberen 8 Bit geben das Vorzeichen in Bezug auf Bit 31 an. So ist es möglich den Registern positive und negative Zahlen zuzuweisen. So ist es möglich die Register nach einer einstellbaren Anzahl von Ereignissen überlaufen zu lassen (siehe dazu Abschnitt 4.1.2.2).

4.1.2.2 Überlauf des Ereigniszählers

Über die PerfEvtSel–Register kann bei Überlauf des jeweiligen Zählerregisters ein Interrupt generiert werden. Analog zur Eigenschaft des Itanium–Prozessors kann die Anzahl der bis zum Überlauf gemessenen Ereignisse durch Initialisierung des Zählerregisters eingestellt werden. Die Ereignismessung kann beim Überlauf eines Zählers noch weitere Daten liefern. Dazu kann für die Ereignisse der detaillierte Zustand der Maschine in einem Speicherbereich abgelegt werden (vergleiche hierzu Abschnitt 2.3.3).

4.1.2.3 Ereignisqualifizierung

Die Möglichkeiten der Ereignisqualifizierung sind im Vergleich zu denen des Intel Itanium eingeschränkter. Folgende Möglichkeiten bestehen:

- **Die Auswahl von Teilereignissen:** Vermittels der "Event Specific Unit Mask" können unter der Vielzahl der Auswalbaren Ereignistypen Teilereignisse bestimmt werden, um die Ereigniszählung auf Details einzuschränken. Die "Unit Mask" wird in den Konfigurationsregistern PerfEvtSel[0,1] angegeben.
- **Beschränkung der Ereigniszählung auf das Privilege–Level:** Über die Konfigurationsregister kann die Ereigniszählung auf Privilege–Level eingeschränkt werden. Die Unterscheidung findet zwischen User–Mode mit den Privilege–Levels 1,2 oder 3 und dem Operating–System–Mode mit Privilege–Level 0 statt.

Die Einschränkung der Ereignismessung auf Instruktionen, Instruktionssätze, Code– und Datenbereiche wird nicht unterstützt (siehe Abschnitt 4.1.1.3).

4.1.3 IBM RISC PowerPC 604e

Die RISC–Architektur des IBM PowerPC 604e unterstützt durch die Hardware Ereigniszähler in ähnlichem Umfang und mit den ähnlichen Möglichkeiten wie die Prozessoren des Herstellers Intel. Die Folgenden Abschnitte beschreiben die verwendeten Register und die nutzbaren Möglichkeiten. Für eine detaillierte Darstellung ist [9] heranzuziehen.

4.1.3.1 Die Performance Counter Register

Die Ereigniszähler des PowerPC 604e benutzen die folgenden Register:

- Es existieren vier 32–Bit Zählerregister – PMC[1,2,3,4].
- Die Ereigniszählung wird durch die "Monitor Mode Control" Register MMCR[0,1] bestimmt und konfiguriert.
- Die Register SIA und SID enthalten die "Sampled Instruction Address" und die "Sampled Data Address", wenn die Zählerregister eingestellt sind einen Interrupt bei Überlauf zu erzeugen. Die exakte Instruktions– und Datenadresse der verursachenden Instruktion wird allerdings nur bei vier speziellen Ereignissen, die im Register PMC1 gezählt werden erfasst. Bei allen anderen Ereignissen wird die Adresse der letzten ausgeführten Instruktion und des letzten benutzten Operanden gespeichert. Meist stehen daher SIA und SDA nicht miteinander in Beziehung. Die Register SIA und SID protokollieren den Zustand der Maschine. Diese Eigenschaft ist nicht so detailliert wie beim P6 (vergleiche Abschnitt 4.1.2.2), entspricht aber der detaillierten Attributierung, wie sie in Abschnitt 2.3.3 beschrieben ist.

4.1.3.2 Überlauf des Ereigniszählers

Wie bei den Prozessoren des Herstellers Intel ist es möglich bei Überlauf der Zählerregister einen Interrupt zu erzeugen. Dazu müssen Bits des Konfigurationsregisters MMCR0 geeignet gesetzt sein. Wird das höchstwertige Bit eines Zählerregisters gesetzt, dann wird diese Situation als Überlauf definiert.

4.1.3.3 Ereignisqualifizierung

Der IBM PowerPC 604e unterstützt ebenfalls die eingeschränkte Ereigniszählung.

- **Beschränkung der Ereigniszählung auf das Privilege–Level:** Der PowerPC 604e kann Ereignisse die im Supervisor–Modus und im User–Modus stattfinden getrennt zählen.
- **Beschränkung der Ereigniszählung auf markierte Prozesse:** Es ist möglich eine Prozess zu markieren, so daß nur die in diesem Prozess auftretenden Ereignisse gezählt werden. Dazu kann die Analysesoftware das Bit 29 des MSR[1] setzen. Ereignisse werden dann nur gezählt, wenn dieses Bit gesetzt ist. So ist, unterstützt durch

[1] Hier ist das Model–Specific–Register gemeint.

die Hardware, die Erstellung eines Per–Prozess–Profils möglich. Die Privilege–Level–Einschränkungen und die Möglichkeit der Markierung der Prozesse können auf jede beliebige Art kombiniert werden.

Eine bedingte Ereigniszählung für bestimmte Code– und Datenbereiche, sowie für bestimmte Instruktionen oder Instruktionssätze wird nicht unterstützt.

4.2 Implementierung der Klassifizierung in den betrachteten Prozessoren

Die Umsetzung der vorgeschlagenen Ereignistypen in den betrachteten Prozessoren ist tabellarisch in Anhang B aufgeführt. Die Auswahl der zu messenden Ereignisse und die Programmierung der Zählerregister unterscheidet sich von Prozessor zu Prozessor – ausführliche Anleitungen sind in den bereits zitierten Quellen [3, 6, 9] zu finden.

4.3 Verfügbarkeit der Ereignistypen der übrigen Hardware

Dieser Abschnitt beschäftigt sich mit der Implementierung der Ereignistypen der übrigen Hardware, welche unter den Interfaces `MEMORY_PERFORMANCE` in Abschnitt 3.1.2.1 und `PERIPHERY` in Abschnitt 4.3.2 zusammengefasst ist.

4.3.1 Das Interface `MEMORY_PERFORMANCE`

Beginnend mit dem prozessorexternen Speicher werden hier die Klassen `MAIN_MEMORY` aus Abschnitt 3.1.2.2 und `DISC` aus Abschnitt 3.1.2.3 nochmals aufgegriffen.

4.3.1.1 Die Klasse `MAIN_MEMORY`

Um festzustellen, ob die Ereignistypen durch Zählerregister in der Hardware unterstützt werden, wurden Handbücher der Hersteller Infineon, Kingston und Samsung herangezogen. Bei keinem dieser Hersteller konnten Hinweise auf Zählerregister für Ereignisse der vorgeschlagenen Ereignistypen gefunden werden (vergleiche dazu [30, 23]).

4.3.1.2 Die Klasse `DISC`

Die technischen Handbücher der Hersteller Seagate und Hitachi wurden zu Rate gezogen, um Hinweise auf Zählerregister für die Ereignistypen `DISC_THROUGHPUT` und `DISC_INTERRUPT` zu finden. Es wurden die Festplatten Seagate Cheetah und Hitachi Deskstar 120 GXP betrachtet.

- **`DISC_THROUGHPUT`**: In [31] fand sich der Hinweis auf die so genannte "transfer rate negotiation message" der Seagate Festplatte, welche die Dauer des Transfers und die

Transferrate angibt. Die S.M.A.R.T. Funktion der Hitachi Festplatten erzeugt eine 512–Byte große Datenstruktur, in der ein so genannter Attributeintrag Auskunft über die "troughput performance" gibt [22].

- **DISC_INTERRUPT**: Bei keiner betrachteten Festplatte ist es möglich die Anzahl der ausgelösten Interrupts zu zählen.

4.3.2 Das Interface PERIPHERY

Die Implementierung der Ereignistypen der Peripherie, wie sie in Abschnitt vorgeschlagen sind, wird in den folgenden Abschnitten 4.3.2.1 und 4.3.2.2 behandelt.

4.3.2.1 Die abstrakte Klasse CONNECTION

Um die Implementierung von Zählerregistern in Netzwerkadaptern und Schnittstellen für die in Abschnitt 3.1.3.2 und Abschnitt 3.1.3.3 behandelten Ereignistypen der Klasse NETWORK zu untersuchen, wurde versucht Hinweise auf die Ereignistypen in Handbüchern verschiedener Hersteller zu finden. Die vorgeschlagenen Ereignistypen waren jedoch bei keinem bekannten Hersteller in der Hardware der Adapter umgesetzt.

Für die Klasse INTERFACES in Abschnitt 3.1.3.3 konnten keine technischen Unterlagen gefunden werden, die auf eine Implementierung von Zählerregistern für Ereignisse der vorgeschlagenen Ereignistypen hindeuten.

4.3.2.2 Die Klasse Graphics

In der Klasse GRAPHICS in Abschnitt 3.1.3.4 werden Ereignistypen vorgeschlagen, die die Performance des Graphikadapters beschreiben sollen. Es konnten keine technischen Unterlagen Herstellern von Graphikadaptern gefunden werden, die auf eine Unterstützung der vorgeschlagenen Ereignistypen hindeuten.

4.4 Vorstellung der betrachteten Betriebssysteme

Die folgenden Abschnitte widmen sich der vorgeschlagenen Klassifizierung der Ereignistypen der Software (siehe Abschnitt 3.2). An den Betriebssystemen Windows 2000 der Firma Microsoft und dem Betriebssystem Linux wird untersucht, ob die vorgeschlagenen Ereignistypen in diesen Betriebssystemen implementiert sind. Die Abschnitte 4.4.1 und 4.4.2 behandeln die Möglichkeiten der Ereignismessung der jeweiligen Betriebssysteme im Allgemeinen. In Abschnitt 4.5 wird die Umsetzung der Klassifizierung aus Abschnitt 3.2 beleuchtet.

4.4.1 Windows 2000

Das Betriebssystem Windows 2000 stellt mit dem Platform SDK:Windows Management Instrumentation eine Sammlung von Performance Counter Klassen zur Verfügung [17]. Diese Sammlung von Klassen teilt sich in zwei Gruppen:

- Formatted Performance Counter Classes
- Raw Performance Counter Classes

Die Formatted Performance Counter Classes beziehen die Daten vom so genannten Cooked Counter Provider. Das System Monitor Utility von Windows bezieht ebenso die Daten von dieser Quelle, so daß die Werte der Attribute in den Formatted Performance Counter Klassen die selben sind. Der Cooked Counter Provider berechnet die Werte der Attribute mit vordefinierten Sampling–Intervallen und Formeln. Dem entgegen stehen die Raw Performance Counter Classes, deren Attribute die ursprünglichen und unbehandelten Performance–Daten enthalten. Die Quelle für diese Daten ist der Performance Counter Provider. In den folgenden Abschnitten werden die Attribute der Raw Performance Counter Classes behandelt.

Die abstrakte Klasse `Win32_PerfRawData` bildet die Superklasse für eine Vielzahl von Klassen, die mit ihren Attributen Zähler für alle denkbaren Ereignisse des Betriebssystems bereitstellen. Die Möglichkeiten reichen von der Zählung von Ereignissen des Active Server Pages Moduls bis zur Zählung von Ereignissen die die Prozessoraktivität betreffen. In [17] sind alle Klassen und ihre Attribute detailliert beschrieben.

4.4.2 Linux

Das Betriebssystem Linux verfügt im Gegensatz zu Windows 2000 nicht über eine derartige Bibliothek. Eine ähnlich Funktion erfüllt allerdings das /proc–Dateisystem, welches in einer Hierarchie von Ordnern und Dateien Informationen über den momentanen Zustand des Kernels speichert. Bei der Betrachtung der Implementierung der vorgeschlagenen Ereignistypen wurden die Einträge in diesem Dateisystem als Informationsquelle verwendet. Detaillierte Beschreibungen der Struktur des /proc–Dateisystems finden sich in [29], [14] und den man–Pages [7] des Dateisystems.

Der Unterscheidung zwischen Formatted Performance Counter Classes und Raw Performance Counter Classes des Betriebssystems Windows 2000 ähnlich, existieren auch im /proc–Dateisystem formatierte und unformatierte Dateien. Die Daten der unformatierten Dateien werden durch verschiedenste Programme für den Benutzer aufbereitet. Beispielsweise zeigt der Befehl `ifconfig` die Einträge der /proc–Dateien /proc/net/socket, /proc/net/dev und /proc/net/if_inet6 als Liste an, die leichter für den Benutzer lesbar ist.

4.5 Implementierung der Klassifizierung in den Betriebssystemen Windows 2000 und Linux

Die Implementierung der in Abschnitt 3.2 vorgeschlagenen Klassifizierung ist in Anhang C tabellarisch dargestellt. Dort werden die Ereignistypen einzeln aufgelistet und jeweils deren Verfügbarkeit angegeben.

5 Die Spezifikation abgeleiteter Ereignisse

Die allgemeine Einteilung von Ereignistypen in Klassen (siehe Kapitel 3) legt die Spezifikation,auch abgeleiteter Ereignistypen, durch eine allgemeine Beschreibungssprache nahe. Dem Benutzer soll damit das Mittel an die Hand gegeben werden, selbst definierte Heuristiken, und damit Erfahrungen allgemeiner beschreiben zu können. Eine derart angereicherte und aufbereitete Ereignismessung kann als Datenbasis für Visualisierungsprogramme wie KCachegrind [1] dienen.

Unter 5.1 werden zunächst prinzipielle Überlegungen für den Entwurf einer solchen Spezifikationssprache behandelt. In Abschnitt 5.2 findet sich die syntaktische Darstellung in EBNF [35] mit Erklärung der Sprachelemente. Der Abschnitt 5.3 stellt schließlich einige einfache Beispiele dar.

5.1 Entwurf

Beim Entwurf der Spezifikationssprache wird ein Aufbau angestrebt, welcher die abstrakte Definition eines Ereignistyps und der Attribute ermöglicht. Eine Bibliothek von Grundereignistypen bildet die Basis der abgeleiteten Ereignistypen, welche gleichermaßen definiert werden. Es ist somit möglich, die Menge der Grundereignistypen stetig zu erweitern und auch abgeleitete Ereignistypen für die Berechnung anderer Ableitungen heranzuziehen. Dahinter steht der Gedanke, zunächst in abstrakter Weise die Datenbasis zu definieren, auf der dann vermittels Filterfunktionen Heuristiken angewendet werden können. Der Entwurf lehnt sich stark an `HATF`, einer Spezifikationssprache für die Darstellung der Spuren (Traces) bei der Allokation von Heap–Speicherplatz [15] und der `PA language` an, welche die Angabe von logischen Ausdrücken zur Ereignisqualifizierung unterstützt [39].

Da während des Entwurfs die Beschreibung der abgeleiteten Ereignisse im Vordergrund stand, bot sich bei der Namensfindung folgendes Akronym an: **D**erived **E**vent **S**ecification **L**anguage, kurz **DESL**.

In den folgenden Abschnitten werden einige Definitionen vorgenommen, um Ereignismengen charakterisieren und in ein gegenseitiges Verhältnis bringen zu können.

5.1.1 Ereignismengen und Ereignisströme

In DESL wird davon ausgegangen, daß mit jedem Auftreten eines Ereignisses ein Eintrag erzeugt wird, in dem die gemessenen Daten des Ereignisses gespeichert sind. Diese Daten sind zum einen der Wert des Zählers, zum anderen die Attribute des Ereignisses beziehungsweise des Ereignistyps. Tritt also ein Ereignis auf, so werden die Felder des Eintrags gefüllt. Daran anschließend können Berechnungen auf diesen Feldern stattfinden. Da jeder Eintrag

einen Identifikator besitzt, ist es möglich, auf die einzelnen Felder qualifiziert zuzugreifen. Der Wert, der in den jeweiligen Feldern abgespeichert wird, ist durch seine Interpretation bestimmt. Das heißt, daß bei der Definition der abgeleiteten Ereignisse Metriken einfließen, mit Hilfe derer die vom Grundereignis gelieferten Rohdaten verarbeitet werden. Es ist daher wichtig, daß die Sprache arithmetische, logische und relationale Operationen unterstützt. Beim Entwurf wurden unter anderem Funktionen wie die Berechnung der Summe, des Durchschnitts, die Operatoren `UND`, `ODER` und Vergleiche wie `GRÖSSER` und `KLEINER` berücksichtigt.

5.1.1.1 Ereignismengen

In diesem Abschnitt soll der Begriff des Ereignistyps in Mengenschreibweise dargestellt werden. Es fällt vor allem später bei der Sprachdefinition leichter, mit Hilfe dieser Mengen argumentieren zu können. Dabei wird von folgenden Annahmen ausgegangen:

1. Die Menge der aufgetretenen Ereignisse eines Ereignistyps ist eine Menge von Einträgen r_i. Die einzelnen Felder eines Eintrags stellen die Attribute des aufgetretenen Ereignisses dieses Typs dar.
$$\Gamma_e = \{r_1, r_2, \ldots, r_n\}$$
Γ_e ist also die Menge aller Ereignisse eines Ereignistyps e einer Messung

2. Die Menge der Felder eines Eintrages eines Ereignistyps e heiße Φ_e. Die Anzahl $|\Phi_e|$ der Felder eines Eintrages eines Typs e ist für alle Einträge des Typs e gleich groß.

3. Das Ergebnis, das heißt die gemessenen Ereignisse der Grundereignistypen werden protokolliert. Sie liegen demnach nach einem Messlauf vor. Die Menge dieser Ereignistypen heiße Γ, so daß
$$\Gamma = \bigcup_e \Gamma_e$$

4. Mit Filterfunktionen können die Ereignisse der Typen verarbeitet werden. Eine Filterfunktion f_i berechnet aus den Feldern eines oder mehrerer Ereignisse verschiedener Ereignistypen das Ergebnisfeld i eines abgeleiteten Ereignistyps a. Die Menge der für alle Felder eines abgeleiteten Ereignistyps a angewendeten Funktionen heiße F_a:
$$F_a = \bigcup_{0 \leq i \leq |\Phi_a|} f_i$$

5.2 Sprachdefinition von DESL

Die Spezifikation von Ereignistypen in DESL hat folgenden prinzipiellen Aufbau:

Definition der Eingabedaten
`input{`
`eventtype` `eventtype_name`**`{`**
Angabe der Eingabedatei der Ausgangsdaten

```
File name = file_name;
```
Angabe der Felder, die von den Eingabedaten gefüllt werden
```
type attribute_1;
type attribute_2;
⋮
}
}
```

Spezifikation der abgeleiteten Ereignisse
```
heuristics{
eventtype eventtype_name{
type attribute_1 = Filterausdruck;
type attribute_2 = Filterausdruck;
⋮
}
}
```
Auflistung der als Ergebnis gewünschten Ereignisse
```
output{
eventtype_name_1;
eventtype_name_2;
⋮
eventtype_name_n;
}
```
Die folgenden Abschnitte stellt die Definition von DESL in EBNF[1] dar. Die Bedeutung der Sprachelemente wird an den jeweiligen Stellen angegeben.

5.2.1 Spezifikationsstruktur in DESL

Die äußersten Elemente in der Blockstruktur eines DESL–Programms werden in diesem Abschnitt erläutert. Das Programm teilt sich grundsätzlich in drei Abschnitte – die Definition der Eingabedaten und die Definition der anzuwendenden Heuristik und die Definition der auszugebenden Daten.

Syntax:

specification = *input{* {basic_eventtype}* *},*
heuristics{ {derived_eventtype}* *},*
output{ {eventtype_id}* *};*

[1] In EBNF erfolgt die Kennzeichnung von Schlüsselsymbolen durch Anführungszeichen. Zur besseren Lesbarkeit werden die Schlüsselsymbole abweichend vom Standard in *Kursivschrift* angegeben.

Semantik:

- `input`: Wie oben angegeben wird die Ereignistypspezifikation in DESL durch die Definition der Eingabedaten eingeleitet. Die innerhalb des `input`–Blocks spezifizierten Ereignistypen bilden die Grundmenge Γ der Ereignistypen, auf denen dann Heuristiken angewendet werden, um abgeleitete Ereignistypen zu berechnen. Die definierten Felder werden aber in dieser Umgebung nicht mit Filtern belegt. Da die Eingabedaten von einer Quelle stammen, muß eine Felddefinition vom Typ `File` (siehe Abschnitt 5.2.2) sein und den Pfad zur Quelle enthalten.
- `heuristics`: Alle in der `heuristics`–Umgebung definierten Ereignistypen stellen nun Berechnungen auf den Grunddaten an und bilden die abgeleiteten Ereignistypen.
- `output`: Innerhalb dieser Struktur werden die auszugebenden Ereignistypen, das heißt diejenigen, die für den Benutzer von Interesse sind, aufgelistet. Dies ist dann von Belang, wenn zur Berechnung einer Heuristik Hilfsereignistypen definiert werden, deren Ergebnisse für die Ausgabe aber nicht relevant sind. Im `output`–Block können Ereignisse des `input`–Blocks, sowie Ereignistypen des `heuristics`–Block vorkommen.

5.2.2 Datentypen in DESL

Das Ergebnis einer Verarbeitung des Ereignisstroms kann von unterschiedlichem Typ sein. So kann der Benutzer nur an einem bestimmten Wert, zum Beispiel der Summe aller ausgeführten Instruktionen, interessiert sein, an einer geordneten Menge von Ereignissen, die vor einem bestimmten Ereignis stattfanden, oder an allen Ereignissen, die eine gewisse Bedingung erfüllen. Daher werden in DESL der Listentyp, der Zahltyp und ein Typ für die Aufnahme von Buchstaben(–ketten) definiert. Der Typ `Line` beschreibt den Bezug eines aufgetretenen Ereignisses zum Quellcode auf abstrakter Ebene.

Syntax:

type = *List* | *Number* | *String* | *File* | *Line*;

Semantik:

- `List`: Felder oder Berechnungsergebnisse dieses Typs enthalten eine Menge von Einträgen. Der Listentyp ist mit allen anderen Typen verträglich, insbesondere können auch Records gespeichert werden (siehe Abschnitt 5.2.3).
- `Number`: Dieser Typ dient zur Speicherung von Zahlen, eine Beschränkung des Zahlenraums auf beispielsweise die natürlichen Zahlen ist nicht definiert.
- `String`: Felder des Typs String speichern alphanumerische Buchstaben(–ketten)
- `File`: Derart typisierte Felder stellen die Dateiinhalte dar. Die Datei wird geöffnet, gelesen und deren Daten den spezifizierten Feldern zugewiesen. Dieser Typ ist nur in der `input`–Umgebung zulässig und für den Bezug zu den Eingabedaten bestimmt.

- `Line`: Dieser Typ ist für die Speicherung der Quellcodezeile, in der das Ereignis auftrat, zu verwenden.

5.2.3 Recorddefinition

Der Event–Record stellt die Umgebung für den zu messenden Ereignistyp dar. In der Recordumgebung werden die Felder für die gewünschten Daten definiert und deren Interpretation angegeben. Die Interpretation beeinflußt den Wert des Feldes mittels arithmetischer, logischer oder vergleichender Ausdrücke. Der Strom der durch das Auftreten von Ereignissen erzeugten Records wird auf diese Weise gefiltert.

Syntax:

```
basic_eventtype = header{,
                  File file_name = path;,
                  {type field}*; };
derived_eventtype = header {,
                    {globalVar eventtype_id,}*;
                    {metadata}+ [constraint] [group by {eventtype_attribute_qualifier}*]};
metadata          = type attribute [selfreference name] = filter;;
constraint        = constraint(log_fct | rel_fct | referenceExpr | runExpr);;
header            = eventtype eventtype_id;
globalVar         = attribute;
attribute         = Konstante vom Typ String;
```

Semantik:

- `header`: In der Umgebung *eventtype* wird der Ereignistyp gekapselt. Zur Unterscheidung der einzelnen Ereignistypen trägt jede Umgebung einen Namen, die `eventtype_id`.
- `basic_eventtype`: In dieser Umgebung werden die Basisereignistypen angegeben. Im Gegensatz zu einem abgeleiteten Ereignistyp beschreiben diese Spezifikationen die Datenbasis. Daher werden die Felder hier ohne verarbeitende Filter angegeben. Wichtig ist im *input*–Block die Angabe des Pfadnamens zur Datenquelle (siehe Abschnitte 5.2.1 und 5.2.2).
- `derived_eventtype`: In dieser Umgebung werden die abgeleiteten Ereignistypen spezifiziert. Da hier Berechnungen auf den Daten der aufgetretenen Ereignisse der Grundereignistypen stattfinden sollen, wird nach der Angabe des Feldnamens ein verarbeitender Filter, welcher den Ausdruck darstellt, angegeben.
- `metadata`: Deklaration der Felder eines Eintrages. Der Wert des Feldes `attribute` ist das Rohdatum oder der Wert den der Filter `filter` aus den Rohdaten berechnet. Wenn das Schlüsselwort `selfreference` gefolgt von einem Namen angegeben wird,

bedeutet dies, daß sich Feldbezeichnungen im Filterausdruck auf das aktuelle Feld beziehen. Man stelle sich vor, daß zur Berechnung des Filters Ergebnisse aus vorhergegangenen Berechnungen benötigt werden. Im *constraint*–Ausdruck muß jedes Feld im Filterausdruck, welches einen Verweis auf das aktuelle Feld besitzt, aufgeführt werden.

- `globalVar`: Mit den globalen Variablen kann auf frühere Spezifikationen von Ereignistypen zugegriffen werden. Mit diesem Konstrukt ist es auch möglich auf ein und denselben Ereignistyp über verschiedene Variablen zugreifen zu können.
- `attribute`: Die Daten, die ein aufgetretenes Ereignis zur Verfügung stellen soll oder kann, werden in den Feldern des Eintrages gespeichert. Diese Recordfelder stellen die Attribute des Ereignisses dar und sind je nach Ereignis unterschiedlich in Anzahl und Typ.
- `constraint`: In abgeleiteten Ereignistypen werden Berechnungen auf den Feldern der Records der Grundereignistypen vorgenommen. Das Schlüsselwort *constraint* schränkt die Menge der für die Berechnung zu verwendenden Felder ein. Zur Erklärung des Schlüsselwortes werden nun die folgenden Definitionen vorgenommen.

Variablen und Terme in DESL

Zunächst werden die Variablen und Terme in DESL definiert.

1. Die im Ausdruck vorkommenden Recordfelder $v_{i,j,k}$ der Records, also die Attribute des Ereignisses, sind Variablen. Dabei bezeichnet zum Beispiel der Ausdruck $v_{0,5,3}$ innerhalb dieser Definition[2] das dritte Attribut des fünften Records des ersten spezifizierten Ereignistyps.
2. Funktionen in DESL sind diejenigen arithmetischen Ausdrücke, die durch die Produktion `arith_expr` gebildet werden und die arithmetischen Funktionen, welche vermittels `arith_fct` entstehen. Wie in Abschnitt 5.2.4 zu sehen ist, sind die Funktionssymbole ein– oder zweistellig.
3. Des weiteren sind Terme wie folgt definiert:
   a) Alle Variablen aus 1 sind Terme.
   b) Wenn $t_0 \ldots, t_k$ Terme sind und f ein Funktionssymbol aus 2 ist, dann ist auch $f(t_0, \ldots, t_k)$ ein Term, wobei f k–stellig ist.

Formeln in DESL

Nachdem nun die Variablen und Terme definiert sind, können im weiteren induktiv die Formeln festgelegt werden. Nur innerhalb der `constraint`–Umgebung sind diese Formeln anzugeben.

[2]Die Bezeichnung eines Attributs mit $v_{i,j,k}$ ist äquivalent zu einer Referenzierung des Attributs mittels event_id.fieldname und wird hier aus Platzgründen und der Allgemeinheit halber verwendet.

1. Sind t_1, t_2 Terme wie oben definiert, dann sind alle Ausdrücke der Produktionen `rel_fct`, `rel_list_fct`, `log_fct`, `referenceExpr` und `runExpr` (siehe Abschnitt 5.2.4) die auf ein oder zwei Terme angewendet werden Formeln.
2. Formeln können in DESL nur innerhalb der `constraint(.)`–Umgebung definiert werden, wobei `constraint` implizit über die Variablen der Formel allquantifiziert.

- `group by`: Werden Werte durch Summenbildung etc. aggregiert und ist diese Aggregation nur über gleichartige Attribute gewünscht, so sind alle Attribute bis auf die aggregierten in der `group by`–Klausel anzugeben. Das folgende Beispiel soll die Bedeutung deutlich machen. Wird beispielsweise ein abgeleitetes Ereignis $a_{sumcycle}$ betrachtet, welches ein Attribut `cyclesum` enthält. Dieses Attribut berechnet sich aus der Summe des Attributs `cycles` eines Grundereignistyps e_k aus Γ. Der Grundereignistyp besitze zusätzlich noch das Attribut `function`. `cyclesum` soll nun nur die Summe der Attributwerte von `cycles` berechnen, für die die zugehörigen Attributwerte von `function` untereinander den selben Wert haben. Eben dieser Attributname ist dann in der `group by`–Klausel aufzuführen.

5.2.4 Definition eines Filters

Über die Angabe von Filtern soll die Berechnung der Felder eines Records in Ableitungen ermöglicht werden. Meist reicht es aus, arithmetische Ausdrücke für die Berechnung der Ereignisse eines abgeleiteten Ereignistyps anzugeben. In DESL sollen aber diese Möglichkeiten erweitert werden, so daß auch logische Vergleiche und Ähnliches spezifizierbar sind.

Syntax:

filter = filter_expr | *none*;
filter_expr = arith_expr | arith_fct | rel_list_fct;
arith_expr = arith_expr **+** X | arith_expr **-** X | X;

X = const | X * Y | X / Y | Y;
Y = *(* arith_expr *)* | arith_fct | eventtype_attribute_qualifier [ω]
| globalVar.field [ω];
ω = ·[+ | **-**][*0,...,9*]*.[*0,...,9*]*;

rel_fct = *l(*[arith_expr | arith_fct], a*)* | *g(*[arith_expr | arith_fct], a*)*
| *le(*[arith_expr | arith_fct], a*)* | *ge(*[arith_expr | arith_fct], a*)*
| *eq(*[arith_expr | arith_fct], a*)*;
rel_list_fct = @ rel_fct;
arith_fct = *avg(*[arith_expr | arith_fct]*)*
| *max(*[arith_expr | arith_fct]*)*
| *min(*[arith_expr | arith_fct]*)*
| *sum(*[arith_expr | arith_fct]*)*
| *inc(*[arith_expr | arith_fct], const*)*
| *num(*eventtype_attribute_qualifier | globalVar.attribute*)*;

log_fct = *not(*[rel_fct | log_fct]*)*
| *and(*[rel_fct | log_fct | referenceExpr | runExpr], [rel_fct | log_fct | referenceExpr]*)*
| *or(*[rel_fct | log_fct | referenceExpr]*)*;
referenceExpr = name.attribute*(*rel_fct[, epshandler]*)*;
runExpr = *run(*eventtype_id*)* = [*bottom_up* | *top_down*];
epshandler = const;
a = const | eventtype_attribute_qualifier | eventtype_id | globalVar**.**attribute;

Semantik

- `filter`: Wie oben erwähnt, sondert der angegebene Filter die anfallenden Daten aus. Im Unterschied zur *constraint*–Umgebung wird mit der Filterdefinition die Heuristik angegeben. Der im Filter spezifizierte Ausdruck nimmt die Berechnung auf der durch den *constraint*–Ausdruck bestimmten Teilmenge von Einträgen beziehungsweise deren Attribute vor.
- `none`: Es werden nur die Rohdaten ohne weitere Verarbeitung in das Feld gespeichert.
- `filter_expr`: Die Definition erlaubt die Konstruktion arithmetischer Ausdrücke.
- `rel_fct`: Stellt den Syntax der vergleichenden Funktionen dar.
- `rel_list_fct`: Das Zeichen @ leitet bei den Vergleichsfunktionen den Listenkontext ein. Derartige Ausdrücke haben in einer Liste alle diejenigen Records zum Ergebnis, welche dem im Ausdruck spezifizierten Vergleich genügen.
  Es ist darauf zu achten, daß das Ergebnis einem typkompatiblen Feld des Records zugewiesen wird, das heißt, das entsprechende Feld muß vom Typ *List* sein.
- `arith_fct`: Diese Funktionen komprimieren den Bestand der erzeugten Einträge.
  - *avg*: Berechnet den Durchschnitt eines Ausdrucks.
  - *sum*: Berechnet die Summe eines Ausdrucks.
  - *min*: Bestimmt das Minimum eines Ausdrucks.
  - *max*: Bildet das Maximum eines Ausdrucks.
  - *inc*: Erhöht die angegebenen Variablen um den Wert der Konstante `const`.
  - *num*: Zählt das Vorkommen aller Records.
  - *log_fct*: Erlaubt die Konstruktion logischer Ausdrücke.
- `referenceExpr`: Dieses Konstrukt ist immer dann anzuwenden, wenn sich Berechnungen auf Felder des aktuellen Records der Ereignistypspezifikation beziehen. Im *constraint*–Ausdruck ist der Name der Selbstreferenz anzugeben und, gefolgt von einem Punkt, das Feld welches den Selbstbezug erfährt. In Klammern folgt eine Bedingung die den Bezug zu einem früher berechneten Feld herstellt. Wenn das selbstbezügliche Feld einen undefinierten Wert hat, dann kann für diesen Fall im `epshandler` ein Default–Wert angegeben werden.

- `runExpr`: Dieses Konstrukt legt die Traversierung der Records einer Ereignistypspezifikation fest. Die Ereignisse des Ereignistyps können von unten nach oben (*bottom_up*) oder von oben nach unten (*top_down*) durchlaufen werden.
- `epshandler`: In einer Selbstreferenzbehandlung stellt dieser Ausdruck einen Standardwert für das rückbezügliche Feld ein.

5.2.5 Konstanten– und Basisereignisdefinition

Dieser Abschnitt erläutert die Definition von Konstanten für die Identifizierung von Ereignistypen und der Attribute der Ereignisse.

Syntax

const = Konstante vom Typ Number;
eventtype_id = Konstante vom Typ String;
eventtype_attribute_qualifier = eventtype_id.attribute;

Semantik

- `const`: Eine beliebige Zahlkonstante des Typs `Number`.
- `eventtype_id`: Eine eindeutige alphanumerische Zeichenkette des Typs `String`, der den Ereignistyp benennt.
- `eventtype_attribute_qualifier`: Um auf die Attribute der einzelnen Ereignisse eines Typs zugreifen zu können, muß der Name des Ereignistyps und des gewünschten Feldes angegeben werden. Erfolgt der Zugriff auf über die `eventtype_id` alleine, so sind alle Records und alle Attribute des Records betroffen.

5.3 Beispiele in DESL

Dieser Abschnitt stellt anhand einiger einfacher Beispiele die Verwendung der vorgeschlagenen Spezifikationssprache vor.

1. **First–Level Cache–Misses**: Es wird davon ausgegangen, daß der Prozessor einen Grundereignistyp `l1_misses` zur Verfügung stellt, welcher die fehlgeschlagenen Zugriffe auf den L1–Cache beschreibt. Die gemessenen Ereignisse der fehlgeschlagenen Zugriffe auf den Cache seien in der Datei "L1_MISSES.out" abgespeichert. Der Grundereignistyp `l1_misses` habe das Attribut `nr` und speichere die Zeilennummer der Quellcodezeile, in der das Ereignis auftrat – dies kann auch bedeuten, daß eine bestimmte Anzahl von Ereignissen dieses Typs überschritten wurde (vergleiche Abschnitt 2.3). Der abgeleitete Ereignistyp `num_l1_misses` soll für jede Quellcodezeile

nun die Anzahl der aufgetretenen erfolglosen Zugriffe auf den Cache bestimmen. In DESL sieht die Beschreibung folgendermaßen aus:

```
input{
eventtype l1_misses{
File = "L1_MISSES.out";
Line nr;
}
heuristics{
eventtype num_l1_misses{
Line addr = l1_misses.addr;
Number num = num(l1_misses.nr);
group by l1_misses.nr;}
}
output{
num_l1_misses;
}
```

2. **Prozeduraufrufe**: In diesem Beispiel soll die Anzahl der Aufrufe von Prozeduren berechnet werden. Dafür wird davon ausgegangen, daß der Prozessor einen Ereignistyp `calls` zur Verfügung stellt, welcher beim Auftreten eines Ereignisses dieses Typs die Unterprogrammaufrufe mit Adresse des Aufrufers und Aufgerufenen speichert. In der Datei "CALLS.out" sollen die Adressen des Aufrufers und des Aufgerufenen in den Attributen `caller`, `callee` abgespeichert sein. Die entsprechende Spezifikation sähe dann folgendermaßen aus:

   ```
   input{
   eventtype calls{
   File f = "CALLS.out";
   Line caller;
   Line callee;
   }
   heuristics{
   eventtype num_counts{
   Line callee = calls.callee;
   Number count = num(calls.caller);
   group by callee;}
   }
   output{
   num_counts;
   }
   ```

   Das abgeleitete Ereignis `num_counts` zählt alle Attribute `caller` des Grundereignistyps `calls`. Da die Attribute durch die `group by`–Klausel in Gruppen nach `callee` geordnet sind, wird pro Gruppe ein Ergebnis in `count` produziert (siehe Abschnitt 5.2.3).

3. **Der abgeleitete Ereignistyp L3–DATA–MISS–RATIO**: Verfügt der Prozessor über die Möglichkeit die erfolglosen Lese– und Schreibzugriffe, sowie die Anzahl der Zugriffe auf Daten im L3–Cache zu zählen, so läßt sich die Rate der fehlgeschlagenen Zugriffe

auf Daten in diesem Cache bestimmen (siehe Abschnitt 3.1.1.29, Tabelle 3.1). Es werden drei Grundereignistypen spezifiziert. Ein Ereignis des Typs `l3_load_misses` mit dem Attribut `addr` trete auf, wenn eine festgelegt Anzahl (vergleiche Abschnitt 2.3) von fehlgeschlagenen Lesezugriffen auf den L3–Cache erfolgt ist. Im Attribut `addr` werde die Quellcodezeilennummer gespeichert. Analog treten Ereignisse des Typs `l3_store_misses` mit dem Attribut `addr` dann auf, wenn Schreibzugriffe auf den L3–Cache fehlschlagen. Ähnlich treten Ereignisse des spezifizierten Typs `l3_data_references` immer dann auf, wenn im Allgemeinen Daten oder Instruktionen des L3–Caches referenziert werden. Auch dieser Ereignistyp enthält das Attribut `addr`. In der Datei "L3_LOAD.out", "L3_STORE.out" und "L3_REF.out" seien die Daten zu den Ereignistypen `l3_load_misses`, `l3_store_misses` und `l3_data_references` abgespeichert. Die Spezifikation des abgeleiteten Ereignistyps `l2_data_miss_ratio` sieht in DESL folgendermaßen aus:

```
input{
/*L3-LOAD-Misses*/
eventtype l3_load_misses{
File = "L3_LOAD.out";
Line addr;}
/*L3-STORE-MISSES*/
eventtype l3_store_misses{
File = "L3_STORE.out";
Line addr;}
/*L3-REFERENCES*/ eventtype l3_data_references{
File = "L3_REF.out";
Line addr;}
}
heuristics{
eventtype l3_data_miss_ratio{
Line addr = l3_misses.addr;
Number ratio = (num(l3_load_misses.addr) +
num(l3_store_misses.addr)) / num(l3_data_references.addr);
group by addr;
constraint (eq(eq(l3_load_misses.addr, l3_store_misses.addr),
l3_data_references.addr));}
}
output{
l3_data_miss_ratio;
}
```

Wie in Abschnitt 5.2.3 beschrieben, muß jedes nicht aggregierte Feld in die group–by Klausel aufgenommen werden – in diesem Fall ist dies das Feld `addr` des Typs `l3_data_miss_ratio`. Die `constraint`–Anweisung besagt, daß für die Berechnung die Felder `l3_load_misses.addr`, `l3_store_misses.addr` und `l3_data_references.addr` den gleichen Inhalt haben müssen. Sonst würde das Ergebnis aus zufälligen Einträgen in den Dateien L3_LOAD.out, L3_STORE.out und L3_REF.out berechnet. Es wäre dann nicht gesichert, daß die Quellcodezeilen übereinstimmen. Die Berechnung findet aber nur dann statt, wenn auch für alle drei Attribute `addr` der

Ereignistypen `l3_load_misses`, `l3_store_misses` und `l3_data_references` ein Eintrag vorhanden ist.

6 Anwendung

Die in Kapitel 5 vorgeschlagene Spezifikationssprache wird in diesem Kapitel dazu verwendet werden, um die Inklusiv–/Exklusivkostenberechnung bei Prozeduraufrufen zu beschreiben. Es werden zwei Heuristiken für dieses Problem vorgestellt und mit den exakten Werten einer Instrumentierung verglichen. Solche Heuristiken sind immer dann vonnöten, wenn Messungen die Inklusivkosten nicht liefern können, oder auf die Messung dieser verzichtet wurde. Der Abschnitt 6.1 stellt zunächst das Problem im Allgemeinen dar. In Abschnitt 6.2 wird die im Profiling–Tool `gprof` verwendete Heuristik [2] mit DESL spezifiziert, bevor in Abschnitt 6.3 eine alternative Heuristik – die Quotientenheuristik – vorgeschlagen und behandelt wird. Im konkreten Anwendungsfall wurde eben jene als Modul für das Visualisierungsprogramm KCachegrind [1] implementiert. Der Abschnitt 6.4 stellt abschließend die Ergebnisse der Heuristiken im Vergleich zur Instrumentierung dar.

6.1 Die Berechnung der Inklusiv–/Exklusivkosten bei Prozeduraufrufen

Bei der Performance–Analyse von Programmen ist die Frage nach dem Zeitverbrauch eines Programms von zentraler Bedeutung, um ineffiziente Codeabschnitte im Programm identifizieren zu können. Es stellt sich sich die Frage, zu welchem Anteil sich die Summe der gemessenen Ereignisse auf die Prozeduren des Programms verteilt. Die Summe der gemessenen Ereignisse eines Ereignistyps wird im Folgenden als Kosten bezeichnet. Auf die in Abschnitt 1.1 erwähnten Begriffe Inklusivkosten und Exklusiv– oder Selbstkosten eines Prozeduraufrufes soll hier genauer eingegangen werden. Folgendes Beispiel (siehe auch Abbildung 6.1) verdeutlicht den Unterschied:

- eine Prozedur `a` rufe die Prozeduren `b` und `c` auf
- die Prozedur `b` rufe `d` und `e` auf
- `c` rufe `d` und `f` auf
- die Prozeduren `d`, `e` und `f` rufen keine weiteren Prozeduren auf

Wie Abbildung 6.1 zeigt, handelt es sich bei den Exklusiv– oder Selbstkosten um den Anteil der Kosten, den eine Prozedur ohne die Kosten, die durch Aufrufe anderer Prozeduren entstehen, verursacht. Etwaige Aufrufe anderer Prozeduren werden hier also nicht mitgezählt. Dagegen fassen die Inklusivkosten die Selbstkosten und die Kosten, die durch Aufrufe in anderen Funktionen entstanden sind zusammen. In der Abbildung sind diese Kosten an die Aufrufpfeile geheftet. Bei Prozeduren, welche keine Aufrufe tätigen, sind

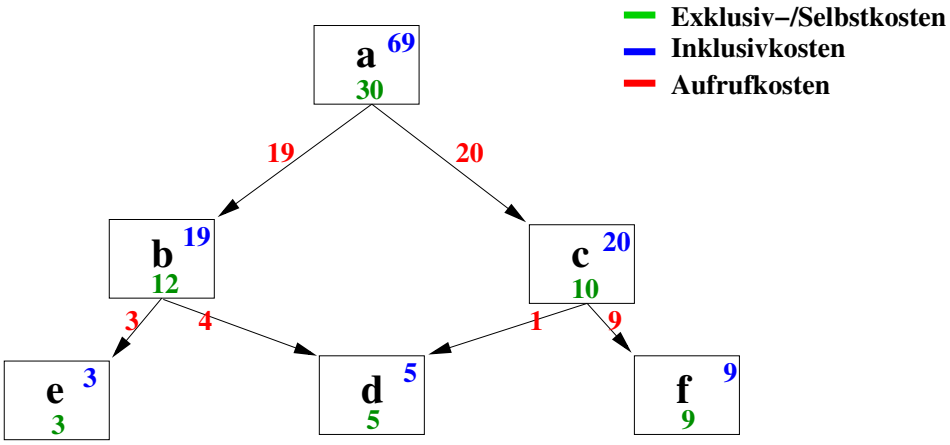


Abbildung 6.1: Ein Aufrufbaum für Prozeduren a, b, c, d, e und f.

demnach Inklusiv– und Selbstkosten identisch – diese Prozeduren werden im Folgenden als ϵ–Prozeduren bezeichnet.

Die Bestimmung dieser zwei Kostentypen ist von unterschiedlich schwieriger Natur. Grundsätzlich ist die exakte Messung der Inklusivkosten nur vermittels Codeinstrumentierung möglich. Diese in Kapitel 2, Abschnitt 2.4 erwähnte Methode der Performance–Analyse hat den Vorteil, daß alle benötigten Daten[1] gemessen werden können. Der zusätzlich eingefügte Code beeinflußt durch seine Ausführung jedoch das Ergebnis der Messung. Im Gegensatz dazu steht das statistische Sampling, welches in bestimmten Intervallen Messungen vornimmt (vergleiche Abschnitt 2.3). Dieses Verfahren beeinträchtigt das Meßergebnis in geringerem Maße, da der Meßfehler durch die Wahl des Sampling–Intervalls einstellbar ist. Der Nachteil des Verfahrens liegt darin, daß Aufrufbeziehungen nicht ermittelt werden können, es daher auch unmöglich ist Angaben über die Inklusivkosten, oder die Aufrufkosten zu machen. An diesem Punkt kann eine Heuristik von Vorteil sein, welche aus den Ergebnissen einer Simulation, die fehlenden Daten des Samplings errechnet. Damit die Anwendung einer solchen Heuristik erfolgreich ist, muß das analysierte Programm einen deterministischen Ablauf haben und das Sampling und die Simulation mit der selben Arbeitslast vorgenommen worden sein. Hat das zu analysierende Programm keinen deterministischen Ablauf, so beziehen sich Sampling und Simulation auf unterschiedliche Abläufe, die nicht miteinander verglichen werden können. Ist die Arbeitslast bei den zwei Messungen unterschiedlich, können die Daten ebenfalls nicht miteinander verglichen werden.

Unterschiedliche Heuristiken verwenden unterschiedliche Daten zur Berechnung dieser Inklusivkosten. In Abschnitt 6.2 wird ein Verfahren dargestellt, welches die Anzahl von Aufrufen einer Funktion als Abschätzung verwendet, wohingegen im Abschnitt 6.3 das Verhältnis zwischen Inklusivkosten der aufgerufenen Funktion und den Aufrufkosten als Grundlage verwendet wird. In diesem Abschnitt werden die Daten von zwei Messungen verwendet. Die Daten der einen Messung stammen aus einer Simulation, die die Inklusiv– und Exklu-

[1] Mit der Instrumentierung des Codes können zum Beispiel Prozedureinsprünge und die Rückkehr aus Prozeduraufrufen etc. genau den aufrufenden Prozeduren zugeordnet werden, daher ist auch die Ausgabe eines Aufrufbaumes mit Inklusiv– und Exklusivkosten möglich. Da während des Samplings nur die Codeadresse zum Zeitpunkt der Messung bekannt ist, kann diese Zuordnung nicht vorgenommen werden.

sivkosten liefert. Aus diesen Daten werden die Verhältnisse zwischen Inklusivkosten der aufgerufenen Funktion und Aufrufkosten berechnet. Die Daten der anderen Messung liefern nur die Exklusivkosten und stammen von einer Sampling–Messung.

6.2 Die Heuristik von `gprof`

Um die Inklusivkosten zu berechnen, können die Kosten eines Aufrufs entsprechend der Anzahl der Aufrufe aufgeteilt werden. Im folgenden Beispiel ist dieses Verfahren dargestellt. Das Tool `gprof` schätzt die in Prozeduraufrufen verbrauchte Zeit ab. Es wird davon ausgegangen, daß der durchschnittliche Zeitverbrauch in jedem Aufruf einer beliebigen Funktion `c` nicht mit dem Aufrufer der Funktion `c` korreliert [2]. Wenn nun die Funktion `c` insgesamt 5 Sekunden der gesamten Zeit verbraucht und $2/5$ aller Aufrufe von `a` kommen, dann werden 2 Sekunden der Inklusivzeit von `c` hinzugerechnet. Die Annahme, daß sich die verbrauchte Zeit entsprechend der Anzahl der Aufrufe verteilt, ist oft korrekt, es gibt jedoch einige Fälle, in denen sie nicht zutrifft. Ein Bespiel dazu wäre, daß mit einer bestimmten Parameterkonstellation von `c` der Aufruf schnell zurückkehrt – beispielsweise könnte `c`, aufgerufen mit 0, schneller ein Ergebnis liefern, als mit einem anderen Parameterwert.

Die DESL–Spezifikation

In diesem Abschnitt wird die Spezifikation mittels DESL vorgeschlagen. Für die Darstellung der Heuristik werden folgende Annahmen gemacht:

1. Es existiert ein nicht spezifiziertes Ereignis `fn` mit den Attributen `Line start` und `Line end`. Das Attribut `start` sei die Anfangsadresse einer Funktion, und damit auch ihr Symbolname, das Attribut `end` stellt die Endadresse einer Funktion dar. Das Ereignis `fn` soll also alle Funktionen eines Programms und deren Ausdehnung über den Code beinhalten. In `gprof` existiert eine ähnliche Abbildung, die Codeadressen Funktionsnamen zuordnet.

2. Es existiert ein gemessenes Ereignis `lbl_cost`, welches ein Histogramm über den Code beinhaltet. Dieses Ereignis habe die Attribute `Line addr` und `Number hit`. Mit diesem Ereignis wird während des statistischen Samplings aufgezeichnet, wie oft eine bestimmte Codeadresse besucht wurde. Somit errechnen sich die Exklusivkosten einer Funktion f folgendermaßen:

$$excl\_cost = \sum_{\substack{lbl\_cost \\ f.start \leq lb\_cost.addr \leq f.end}} lbl\_cost.hit$$

3. Es existiert ein gemessenes Ereignis `calls`, welches die Aufrufbeziehungen ermittelt. Dieses Ereignis besitzt drei Attribute: die Adresse des Aufrufers (`Line caller`) und die des Aufgerufenen (`Line callee`), sowie die Anzahl (`Number times`) der Aufrufe.

4. Es existiere eine Ordnung über den gemessenen Ereignissen, so daß die Ereignisse entsprechend der Anzahl der Aufrufe, die eine Funktion tätigt geordnet sind. Die Funktionen sollen von vielen getätigten Aufrufen zu wenig getätigten Aufrufen geordnet sein.

5. Es wird angenommen, daß vor der Spezifikation in DESL eine Zyklenerkennung stattgefunden hat, um Rekursionen aufzulösen. Wäre dies nicht der Fall, so würde die Berechnung quasi in einer Endlosschleife enden.

Beginnend mit dem `input`–Block werden die Ereignistypen `calls` und `lbl_cost` spezifiziert. Die Daten der Grundereignisse liegen in einer Datei Namens "gmon.out".

```
input{
eventtype calls{
File = "gmon.out";
Line caller;
Line callee;
Number times;}

eventtype lbl_cost{
File = "gmon.out";
Line addr;
Number hit;}
}
```

Diese zwei Ereignistypen stellen den gesamten `input`–Block dar und sind die Datenbasis, auf welcher der nun folgende `heuristics`–Block Berechnungen vornimmt. Als erster abgeleiteter Ereignistyp werden nun die Exklusivkosten aller im zu untersuchenden Programm vorhandenen Funktionen ermittelt.

```
heuristics{
eventtype excl_cost{
Line f = fn.start;
Number eC = sum(lbl_cost.hit);
constraint(and(ge(fn.start, lbl_cost.addr), le(fn.end,
lbl_cost.addr)));
group by f;}
⋮
} /*Ende des heuristics-Blocks*/
```

Wie in Punkt 2. der DESL–Spezifikation angesprochen, summiert das Ereignis `excl_cost` die Häufigkeiten der angesprungenen Codeadressen auf. Um die Gesamtzahl der Aufrufe einer Funktion, unabhängig von deren Aufrufer zu bestimmen, wird nun das Ereignis `total_times_called` im `heuristics`–Block definiert.

```
eventtype total_times_called{
Line f = calls.callee;
Number total = sum(calls.times);
```

```
group by f;}
```

Der dritte Ereignistyp des `heuristics`–Blocks berechnet den Quotienten aus Gesamtaufrufen einer Funktion und der Anzahl der Aufrufe einer speziellen Funktion. Das Attribut `quot` enthält diesen Quotienten.

```
heuristics{
eventtype quotient{
v1 calls, v2 total_times_called;
Line caller = v1.caller;
Line callee = v1.callee;
Number quot = v1.times / v2.total;
constraint(eq(v1.callee, v2.f));
}
```

Schließlich berechnet der abgeleitete Ereignistyp `incl_cost` die Inklusivkosten einer Funktion indem die Kosten der Funktion selbst und die mit dem Quotienten gewichtete Summe der Inklusivkosten der aufgerufenen Funktionen addiert.

```
eventtype incl_cost{
v1 excl_cost, v2 excl_cost;
Line caller = calls.caller;
Line callee = calls.callee;
Number incl = v1.eC + callCost;
Number self = v2.eC;
Number callCost selfreference r = quotient.quot * (sum(incl - self)
+ self);
constraint( and( and( and( and( and( eq(v1.caller, callee)),
eq(v2.caller, caller)), eq(quotient.caller, caller),
r.incl(eq(callee, incl_cost.caller), 0))), r.self(eq(callee,
incl_cost.caller))), run(calls) = bottom_up));
}
```

Mit diesem Ereignistyp ist der `heuristics`–Block vollständig. Ihm folgt der `output`–Block, in welchem die auszugebenden Ereignisnamen aufzuführen sind.

```
output{
incl_cost;
}
```

Alle anderen spezifizierten Ereignisse dienten nur der schrittweisen Berechnung und können als Hilfsereignisse angesehen werden, an deren Ergebnis der Benutzer in der Ausgabe nicht interessiert ist.

6.3 Eine alternative Heuristik – die Quotientenheuristik

Wie in Abschnitt 6.1 angesprochen, stellt eine Simulation alle benötigten Daten zu Verfügung. Insbesondere sind die Inklusiv– und Exklusivkosten bekannt, aber auch die Kosten der jeweiligen Prozeduraufrufe, während beim Sampling nur Exklusivkosten ermittelt werden können. In Abbildung 6.2 und 6.3 ist abermals der Aufrufbaum für die Prozeduren `a`, `b`, `c`, `d`, `e` und `f` dargestellt. Im oberen Bild sei das Ergebnis eines Samplinglaufes zu sehen, im unteren das einer Simulation.

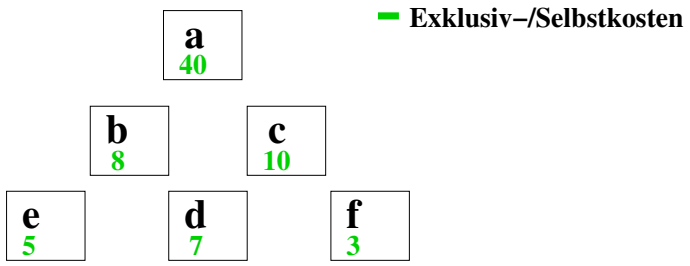


Abbildung 6.2: Die Meßergebnisse für Prozeduren `a`, `b`, `c`, `d`, `e` und `f` nach Abschluß des Samplings.

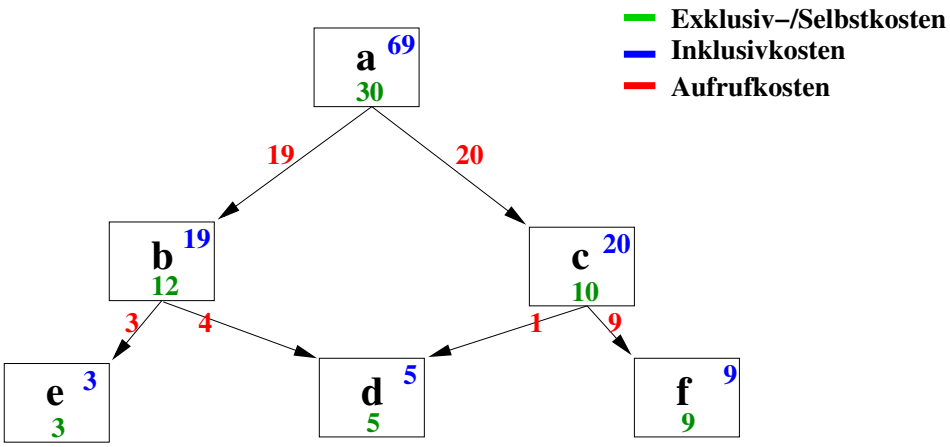


Abbildung 6.3: Die Meßergebnisse für Prozeduren `a`, `b`, `c`, `d`, `e` und `f` nach Abschluß eines Simulationslaufes.

Sieht man nun die Ergebnisse der Simulation als korrekte Referenzwerte an, so arbeitet die Heuristik wie folgt:

1. Berechnung des Verhältnisses zwischen den Aufrufkosten und den Inklusivkosten aller aufgerufenen Prozeduren in der Simulation.

2. Multiplikation des aus der Simulation errechneten Quotienten mit den Inklusivkosten der zugehörig aufgerufenen Prozedur. Die Inklusivkosten stammen ebenfalls aus der Simulation

3. Addition des Ergebnisses zu den Selbstkosten der aufrufenden Funktion zur Errechnung der Inklusivkosten.

Im Beispiel aus Abbildung 6.3 ergibt sich demnach für Schritt 1:

Berechnung der Quotienten in der Simulation

Aufrufer	**Aufgerufener**					
	a	b	c	d	e	f
a	-	$19/19$	$20/20$	-	-	-
b	-	-	-	$4/5$	$3/3$	-
c	-	-	-	$1/5$	-	$9/9$
d	-	-	-	-	-	-
e	-	-	-	-	-	-
f	-	-	-	-	-	-

Tabelle 6.1: In den Zellen steht der Quotient aus Aufrufkosten und Inklusivkosten der Prozeduren der Simulation.

Ist der jeweilige Quotient berechnet, so können die Inklusivkosten des Samplinglaufes berechnet werden. Zunächst werden die Aufrufkosten ermittelt. Dabei ist zu beachten, daß mit den ϵ–Prozeduren der Simulation begonnen werden muß, also der Aufrufbaum in der Simulation von unten nach oben durchgelaufen wird. Wie in Abschnitt 6.2 wird dazu davon ausgegangen, daß die Funktionen bereits in der Simulation nach der Anzahl getätigter Aufrufe geordnet sind und eine Zyklenerkennung durchgeführt wurde. Mit einer solchen Bottom–Up–Traversierung des Baumes ergibt sich für die Inklusivkosten des Samplings:

Berechnung der Inklusivkosten im Sampling

Aufrufer	**Inkl.–Kosten Aufrufer**	**Aufgerufener**					
		a	b	c	d	e	f
a	$40 + 18\frac{3}{5} + 14\frac{2}{5} = 73$	-	$19/19 \cdot 18\frac{3}{5}$	$20/20 \cdot 14\frac{2}{5}$	-	-	-
b	$8 + \frac{28}{5} + 5 = 18\frac{3}{5}$	-	-	-	$4/5 \cdot 7$	$3/3 \cdot 5$	-
c	$10 + \frac{7}{5} + 3 = 14\frac{2}{5}$	-	-	-	$1/5 \cdot 7$	-	$9/9 \cdot 3$
d	7	-	-	-	-	-	-
e	5	-	-	-	-	-	-
f	3	-	-	-	-	-	-

Tabelle 6.2: In den Zellen stehen die jeweiligen Anteil von Aufrufkosten und Eigenkosten der Prozeduren. Aufsummiert ergeben sich die Inklusivkosten des Sampling.

Nach Anwendung der Heuristik wurden zwei Ergebnisse erzielt. Zum einen wurde das Ergebnis des Samplings um die Inklusivkosten der Prozeduren erweitert, zum anderen sind nun die Aufrufbeziehungen ermittelbar, so daß der Aufrufbaum entsprechend erstellt werden kann. Die Abbildung 6.4 zeigt das Ergebnis der Berechnung.

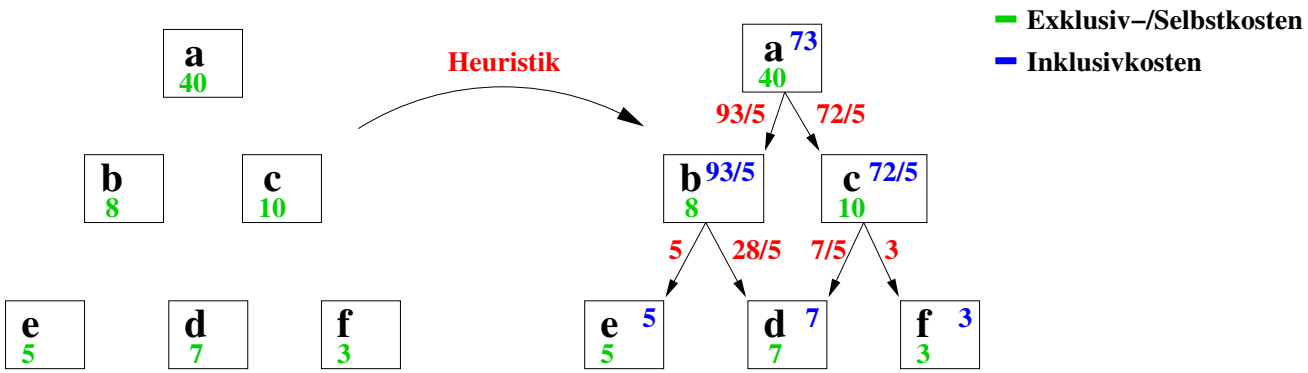


Abbildung 6.4: Die durch die Heuristik angereicherten Daten des Samplings.

Die DESL–Spezifikation

Der Darstellung der Heuristik von `gprof` folgend, stellt dieser Abschnitt die alternative Heuristik in der Spezifikationssprache DESL dar. Folgende Annahmen werden für die Darstellung gemacht:

1. **Simulationsdaten:** Die Simulationsdaten liegen in einer Datei namens "`simulation.out`" vor, welche gemäß Tabelle 6.3 geordnet sind (vergleiche Abbildung 6.3). Außerdem sind die Daten nach den Aufrufen geordnet. Das heißt, daß Prozeduren, welche keine Aufrufe tätigen, am Ende der Tabelle erscheinen. Diese ϵ–Prozeduren haben identische Inklusiv– und Selbstkosten, alle übrigen Felder sind mit dem speziellen Wert ϵ belegt.

Datenschema der Simulationsdaten

Aufrufer	Aufgerufener	Inkl.–Kosten Aufrufer	Selbstkosten Aufrufer	Aufrufkosten
A	B	69	30	19
A	C	69	30	20
⋮	⋮	⋮	⋮	⋮
D	ϵ	5	5	ϵ

Tabelle 6.3: Die Tabelle enthält für jeden Aufruf eine Zeile, in der die Inklusiv– und Selbstkosten des Aufrufers gespeichert sind, sowie die aufgerufene Prozedur und die Kosten, die der Aufruf verursachte.

2. **Datenschema der Daten des Samplings:** Die Daten des Samplings liegen in der Datei "`sampling.out`" vor und sind wie in Tabelle 6.4 organisiert (vergleiche Abbildung 6.2).

Mit diesen Annahmen wird nun zuerst der `input`–Block spezifiziert. Der Block besteht aus den zwei Ereignistypen `simulation` und `sampling`. Der erste Ereignistyp spezifiziert die Simulationsdaten und hat in DESL folgende Gestalt:

Datenschema der Sampling–Daten

Funktion	Inklusiv–/Selbstkosten
A	40
⋮	⋮
F	3

Tabelle 6.4: Prozeduren und ermittelten Kosten des Samplings.

```
input{
eventtype simulation{
File = "simulation.out";
Line caller;
Line callee;
Number incl;
Number self;
Number callCost;
}
//es folgt die Spezifikation der Sampling-Daten
⋮
}
```

Nachdem die Simulationsdaten spezifiziert worden sind, folgt im `input`–Block die Spezifikation der Sampling–Daten.

```
eventtype sampling{
File = "sampling.out";
Line fct;
Number cost;
}
```

Damit ist der `input`–Block vollständig und es kann mit der Definition des `heuristics`–Blocks begonnen werden. Der `heuristics`–Block besteht aus den zwei Ereignistypen `quotient` und `incl_cost`. Der erste Ereignistyp `quotient` berechnet den Quotienten aus Aufrufkosten und Inklusivkosten der aufgerufenen Prozedur aus den Simulationsdaten.

```
heuristics{
eventtype quotient{
v1 simulation, v2 simulation;
Line caller = v1.caller;
Line callee = v1.callee;
Number quot = v1.callCost / v2.incl;
constraint(eq(v1.callee, v2.caller));
}
/*weitere Ereignisspezifikationen*/
⋮
}
```

Die `constraint`–Bedingung hat den Zweck, daß die Inklusivkosten der aufgerufenen Prozedur verwendet werden und nicht die Inklusivkosten des Aufrufers. Nach der Berechnung ergibt sich die Tabelle 6.5.

Aus der Simulation berechnete Quotienten

Aufrufer	Aufgerufener	Quotient
A	B	1
⋮	⋮	⋮
D	ϵ	1

Tabelle 6.5: Aufrufer und aufgerufene Prozedur mit Aufrufquotienten.

Das letzte Ereignis des `heuristics`–Blocks berechnet die Inklusivkosten gemäß der zuvor errechneten Quotienten für die Prozeduren in `sampling`.

```
eventtype incl_cost{
v1 sampling, v2 sampling;
Line caller = simulation.caller;
Line callee = simulation.callee;
Number incl = v1.cost + callCost;
Number self = v2.cost;
Number callCost selfreference r = quotient.quot * (sum(incl - self)
+ self);
constraint( and( and( and( and( and( eq(v1.caller, callee)),
eq(v2.caller, caller)), eq(quotient.caller, caller),
r.incl(eq(callee, incl_cost.caller), 0))), r.self(eq(callee,
incl_cost.caller))), run(simulation) = bottom_up));
}
```

6.4 Vergleich der Heuristiken

In diesem Abschnitt werden an einem einfachen Beispielprogramm (siehe Anhang D) die Heuristiken von `gprof` (vergleiche Abschnitt 6.2), die Quotientenheuristik (vergleiche Abschnitt 6.3) und die Daten der Simulation mit den Daten einer Instrumentierung verglichen.

6.4.1 Qualitätsmaß für den Vergleich der Ergebnisse

Um die Güte einer Heuristik anzugeben, bietet sich an, den Abstand der gemessenen Werte zu den Werten der Instrumentierung zu berechnen. Der Abstand Δ eines berechneten Wertes b_f einer Funktion f zum realen Wert r_f der Instrumentierung der selben Funktion f werde definiert als:

$$\Delta_f = \sqrt{(b_f - r_f)^2}$$

Der Abstand von zwei Funktionen f und g ist dann gegeben durch:

$$\Delta_{f,g} = \sqrt{(b_f - r_f)^2 + (b_g - r_g)^2}$$

Entsprechend wird der Abstand von drei und mehr Funktionen berechnet. Je geringer der Wert von Δ ist, desto genauer ist die jeweilige Heuristik.

Datenerfassung

Der Abschnitt beschäftigt sich mit der Datenerfassung für die Heuristiken. Die Messdaten wurden auf einem Dual–Prozessor–System Intel Pentium II (Deschutes) mit einem Hauptspeicher von 512MB RAM unter dem Betriebssystem Debian Linux erstellt.

- **Referenzdaten**: Die Referenzdaten wurden durch die Instrumentierung des Beispielprogramms gewonnen. Im Code wurden die Inklusiv– und Exklusivkosten durch die Bibliotheksfunktion `clock()` ermittelt. Die gemessenen Werte sind Millisekunden.
- **Simulationsdaten**: Um die Daten der Simulation zu erhalten, wurde das Programm `calltree` [1] verwendet. Dieser Cache–Simulator stellt die Daten für den Aufrufbaum zur Verfügung. Es ist möglich, den erzeugten Datensatz direkt in das Visualisierungsprogramm `KCachegrind` zu laden und die Visualisierung anzeigen zu lassen. Es wurden für den Vergleich die abgeschlossenen Instruktionen verwendet. Es stellte sich heraus, daß auf der verwendeten Maschine pro Millisekunde circa 1,87 Instruktionen abgeschlossen werden. Der Faktor wurde aus der Gesamtlaufzeit des Programms und den gesamten, während dieser Zeit abgeschlossenen Instruktionen berechnet. Der Faktor konnte auch bei den einzelnen Funktionsaufrufen bestätigt werden. Um Die Ergebnisse der Quotientenheuristik mit der Instrumentierung vergleichen zu können, mußten die Simulationsdaten mit dem Faktor normiert werden.
- **Daten der `gprof`–Heuristik**: Mit dem Profiling–Tool wurden die Daten für die in Abschnitt 6.2 beschriebene Heuristik erzeugt.
- **Sampling–Daten für die Quotientenheuristik**: Die Daten des Samplings sind mit dem Programm `OProfile` [26] erzeugt worden. Bei `OProfile` handelt es sich um ein systemweites Profiling–Tool für Linux. `OProfile` verwendet die Methode des Event–Based–Sampling (siehe Abschnitt 2.3.2) und läßt sich mit einer graphischen Oberfläche konfigurieren. Die Messung am Beispielprogramm zählte ebenfalls die abgeschlossenen Instruktionen (vergleiche dazu Abschnitt 3.1.1.12). Das Programm `OProfile` wurde so konfiguriert, daß jeweils nach 6000 aufgetretenen Ereignissen ein Ereignisrecord ausgegeben wird. Nach der Datenerfassung war es noch vonnöten, mittels des Perl–Scripts `op2caltree`[2] ein für `KCachegrind` lesbares Format zu erzeugen. Es ist wichtig zu bemerken, daß die angezeigten Prozeduraufrufe in der Simulation und im Sampling nicht identisch sein müssen. Einige Prozeduren tauchen im Sampling nicht auf, da an solchen Stellen der vorausgesetzte Determinismus des Programms zum Beispiel durch Systemaufrufe nicht vorhanden ist. Das behandelte Beispielprogramm ist jedoch deterministisch.

In der Abbildung 6.5 ist der zentrale Aufrufbaum mit Aufrufanzahlen des Beispielprogramms zu sehen.

[2] Teil des KCachegrind–Paketes.

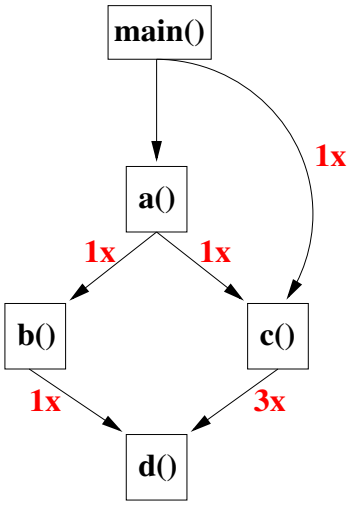


Abbildung 6.5: Der zentrale Aufrufbaum des Beispielprogramms.

Im Beispielprogramm wurde die Arbeitslast allerdings so verteilt, daß, obwohl die Funktion `b()` die Funktion `d()` nur ein Mal aufruft, die Funktionen `c()` und `b()` insgesamt etwa die gleiche Arbeit leisten. Die verbrauchte Zeit in diesen beiden Funktionen ist also in etwa gleich.

Ergebnisse

In diesem Abschnitt werden die Ergebnisse von Simulation, Quotientenheuristik und der Heuristik von `gprof` mit den Daten der Instrumentierung verglichen. In der nachfolgenden Tabelle 6.6 sind die gemessenen und die durch die Heuristiken berechneten Inklusivkosten der in Abbildung 6.5 dargestellten Funktionen angegeben. Die für die Quotientenheuristik gemessenen abgeschlossenen Instruktionen wurden durch den Faktor 1.87 geteilt. Wie oben erwähnt, entsprechen die Werte dann Millisekunden. Die Normierung ist nötig, um die Qualität der Quotientenheuristik angeben zu können. Des weiteren werden in der Tabelle nicht Millisekunden, sondern Sekunden angegeben.

Gemessene und berechnete Inklusivkosten für den Aufrufbaum

	main()	a()	b()	c()	d()
Instrumentierung	199,270000	145,530000	90,340000	96,830000	166,750000
Simulation	197,479250	144,092540	89,642320	96,077068	166,481140
`gprof`	199,270000	124,6437500	48,407500	138,572500	166,750000
Quotientenheuristik	199,270000	145,502810	90,662188	96,793456	167,78517

Tabelle 6.6: Die Tabelle zeigt die Ergebnisse der Messung und die Berechnung der Inklusivkosten des zentralen Aufrufbaumes.

Die Güte der jeweiligen Methode zur Messung der Inklusivkosten in der Simulation und der Berechnung der Inklusivkosten mit Quotientenheuristik und `gprof` wird im Weiteren

verglichen. Dazu wird der Abstand $\Delta_{main,a,b,c,d}$ zur Instrumentierung aller Funktionen für jeweils jede Methode berechnet. In Tabelle 6.7 sind die Ergebnisse angegeben.

Vergleich der Güte der Heuristiken und der Simulation

Methode	$\Delta_{main,a,b,c,d}$
Simulation	2,4149782
`gprof`	62,74224
Quotientenheuristik	1,0851068

Tabelle 6.7: In der Tabelle ist die Qualität der Simulation und der Heuristiken angegeben.

Abschließend kann festgehalten werden, daß die Quotientenheuristik ein exaktes Mittel zur Anreicherung von Sampling–Daten mit Inklusivkosten darstellt. Es fällt vor allem auf, daß in den Fällen, in denen die in Funktionsaufrufen verbrauchte Zeit nicht mit der Anzahl der Aufrufe korreliert, die Quotientenheuristik wesentlich bessere Ergebnisse liefert, als die Heuristik von `gprof`.

7 Zusammenfassung

Die vorgeschlagene Klassifizierung von Ereignistypen für die Performance–Messung in Rechensystemen gibt dem Benutzer einen geordneten Einblick in die Möglichkeiten der Performance–Analyse von Programmen. Die Komplexität der Computer– und Betriebssysteme stellt dem Benutzer eine große Auswahl an Ereignistypen zur Verfügung. Diese Vielfalt wurde mittels der Klassifizierung auch bildlich einer Ordnung unterworfen. Darüber hinaus wurde die Klassifizierung allgemein gewählt, so daß die Anwendung in unterschiedlichen Rechensystemen vereinfacht wird. In Abschnitt 7.1 werden die Ergebnisse der Arbeit diskutiert und der mögliche Nutzen für den Benutzer beleuchtet. Der Abschnitt widmet sich auch den Problemen bei der Entwicklung der Spezifikationssprache DESL und der Implementierung der Heuristik zur Bestimmung der Inklusivkosten von Prozeduraufrufen. Sowohl Vor– wie auch Nachteile dieser Punkte werden kritisch beleuchtet. In Abschnitt 7.2 werden schließlich mögliche zukünftige Projekte, basierend auf der vorgestellten Klassifizierung und der Spezifikationssprache DESL angesprochen.

7.1 Ergebnisse

Die stetig steigende Komplexität von Prozessoren und Programmen führt dazu, daß das Verhalten der in Ausführung befindlichen Software schwer nachzuvollziehen ist. Erkennt der Entwickler, daß das Programm nicht den Erwartungen in puncto Geschwindigkeit entspricht, so ist er gezwungen den Sourcecode einer Prüfung zu unterziehen. Die modernen Rechensysteme unterstützen diesen Prozeß in Hardware dahingehend, daß Ereigniszähler für nahezu alle denkbaren Ereignisse, die während einer Programmausführung auftreten können, zur Verfügung gestellt werden. Die Werte dieser Zähler geben ein Ablaufprofil des Programms und erleichtern so die Suche nach ineffizienten Codeteilen. Allerdings ist die Vielzahl solcher Zähler und ihre Bedeutung schwer zu durchschauen, mangelt es auch oft an einer übersichtlichen Einteilung.

Die vorliegende Arbeit sollte im wesentlichen drei Hauptanforderungen genügen. Erstens soll eine allgemeine Unterteilung von Ereignistypen vorgeschlagen werden, um einen Überblick über die Möglichkeiten der Performance–Messung zu ermöglichen und die Bedeutung der Ereignistypen zu verdeutlichen. Der Abschnitt 7.1.1 widmet sich dem Ergebnis dieser Aufgabe.

Die Kombination der Information der Ereignistypen erweitert die Möglichkeiten der Analyse von Programmen. Daten von unterschiedlichen Messungen könne verwendet werden, um mit Hilfe von Heuristiken nicht gemessene Daten zu ergänzen und so das Profil zu vervollständigen. Zu diesem Zwecke ist zweitens die Beschreibungssprache DESL vorgeschlagen worden, welche die Entwicklung von Heuristiken zur Kombination von Ereignistypen erleichtern soll. In Abschnitt 7.1.2 werden die Ergebnisse dieser Anforderung besprochen.

Eine konkrete Heuristk zu implementieren und deren Qualität zu untersuchen stellte die dritte wesentliche Anforderung an diese Arbeit dar. Zu diesem Zweck wurde eine Zusatzfunktion für das Visualisierungsprogramm `KCachegrind` entwickelt. Die Qualität der Heuristik wurde anschließend an einem einfachen Beispielprogramm untersucht. In Abschnitt 7.1.3 werden die Ergebnisse nochmals diskutiert.

7.1.1 Ergebnisse der Klassifizierung von Ereignistypen

Nachdem in Kapitel 2 die Begriffe der Ereignismessung erläutert wurden, sind in Kapitel 3 die Komponenten eines allgemeinen Rechensystems untersucht worden. Beginnend mit der Hardware wurden Ereignistypen, ihre Bedeutung und ihr Nutzen für die Performanzanalyse vorgeschlagen. Jede einzelne Komponente eines Prozessors, sowie wichtige Peripherie wurden untersucht. Anhand der allgemeinen Struktur eines Betriebssystems wurde im Weiteren die Seite der Software in gleicher Weise beleuchtet. Durch die Verwendung der Modellierungssprache UML entstand eine allgemeine Einteilung von Ereignistypen, welche auch bildlich dem Anwender einen Überblick gibt und die Möglichkeiten der Kombination der Ereignistypen veranschaulicht. Bei der Wahl der Klassifizierung wurde eine Bindung an einen bestimmten Prozessor– und Betriebssystemtyp vermieden, so daß der Anwender beim Studium der Ereignistypen eines konkreten Systems die Klassifizierung als Stütze verwenden kann. Um zu zeigen, daß die aufgestellte Einteilung der Ereignistypen in der Realität anwendbar ist, wurden zum einen die Vertreter Intel P6, Intel Itanium und IBM PowerPC 604e der drei unterschiedliche Prozessorarchitekturen CISC, VLIW/EPIC und RISC behandelt. Zum Anderen ist die Anwendbarkeit der Klassifizierung auf der Seite der Software an den Betriebssystemen Microsoft Windows 2000 und Linux untersucht worden. Es zeigte sich, daß die meisten der vorgeschlagenen Ereignistypen von den jeweiligen konkreten Systemen unterstützt werden und der Klassifizierung folgend eingeteilt werden können.

Aufgrund der zeitlichen Rahmenbedingungen und um Verwirrung zu vermeiden, ist bei der Untersuchung der konkreten Systeme in Kapitel 4 allerdings darauf verzichtet worden, die Namen der Ereignistypen, beziehungsweise den Zugriff auf die Ereigniszähler zu nennen – allein die Verfügbarkeit wurde angegeben. Im konkreten Anwendungsfall geben die technischen Handbücher der jeweiligen Systeme detaillierte Informationen über den Gebrauch der Ereignistypen, welche anhand der vorgeschlagenen Klassifizierung durchschaubar zugeordnet werden können.

7.1.2 Ergebnisse der Spezifikationssprache DESL

Die Klassifizierung der Ereignistypen machte unter anderem deutlich, daß die geeignete Kombination von Ereignistypen zu weiteren, abgeleiteten Ereignistypen führt. Die Kombination bewirkt, daß mehr Information über den Ablauf eines zu untersuchenden Programms erlangt werden kann, als dies die alleinige Betrachtung der Ereignistypen ermöglicht. Zu diesem Zweck wurde die Spezifikationssprache DESL vorgeschlagen, die Rohdaten von Profilmessungen einer geeigneten Verarbeitung unterziehen kann. Die Beschreibungssprache orientiert sich dabei an der Klassifizierung in der Weise, daß die Ereignistypen durch Datenstrukturen modelliert werden und abgeleitete Ereignistypen auf die Felder dieser Datenstrukturen zugreifen können. Die Beschreibungssprache wurde in EBNF angegeben und

einige Beispiele verdeutlichten die Anwendung. In Kapitel 6 sind zwei Heuristiken zur Berechnung von Inklusiv–/Exklusivkosten von Prozeduraufrufen in DESL spezifiziert worden, so daß die Anwendbarkeit von DESL gezeigt werden konnte. Es zeigten sich beim Entwurf von DESL außerdem Parallelen zu SQL und HATF [15], so daß eine tatsächliche Implementierung möglich ist.

Zwei Punkte sind allerdings kritisch zu bemerken. Zum ersten wurde DESL nur in EBNF vorgeschlagen, so daß über die Praktikabilität der tatsächlich implementierten Sprache keine Aussagen gemacht werden können. Zweitens zeigte sich, daß die Spezifikation der Heuristiken in DESL in Kapitel 6 sehr kompliziert, wenn auch nicht unmöglich ist. Weitergehende Untersuchungen an DESL sind vonnöten, um die Anwendung zu vereinfachen. Die Wahl eines funktionalen Berechnungsschemas für die Spezifikation von Ereignistypen und Heuristiken könnte eine vereinfachte Anwendung zu Folge haben. Derartige Untersuchungen können in weiterführenden Arbeiten zu einer tatsächlichen Implementierung der Spezifikationssprache führen.

7.1.3 Ergebnisse der Implementierung der Quotientenheuristik

Das Kapitel 6 beschäftigte sich mit der konkreten Umsetzung einer Heuristik zur Bestimmung von Inklusivkosten von Prozeduraufrufen. Zu Beginn des Kapitels wurde die Methode des Profiling–Tools `gprof` dargestellt und in DESL beschrieben. Im Anschlussß wurde die Quotientenheuristik erklärt. Diese Heuristik berechnet die Kosten-Verteilung anhand des Quotienten der Kosten des Aufrufs und der Inklusivkosten der aufgerufenen Prozedur. Die Quotienten werden aus Simulationsdaten berechnet und bei Sampling–Daten angewendet, so daß diese unvollständigen Daten vermittels der Quotientenheuristik angereichert und vervollständigt werden können – auch diese Heuristik wurde mit DESL beschrieben. Zum Zwecke der konkreten Anwendung wurde ein Zusatzmodul für das Visualisierungsprogramm `KCachegrind` [1] programmiert. Es zeigte sich, daß die in der Simulation fehlenden Daten zuverlässig hinzugefügt wurden. Die Anwendung des Programms auf Daten einer Simulation zeigte das erwartete Ergebnis und ergab, daß die berechneten Werte zu denen der Simulation identisch sind. Angewendet auf Sampling–Daten wurden gute Ergebnisse erzielt. Es konnte gezeigt werden, daß Messungen mit Samplingtools wie `OProfile` [26] vermittels der Quotientenheuristik zuverlässig angereichert werden können. Es steht also ein Mittel zur Verfügung, welches die zeit– und speicherplatzsparende Profilmessung ermöglicht, gleichzeitig jedoch durch die Quotientenheuristik auf detaillierte Aussagen über den Programmablauf nicht verzichtet werden muß. Da die Quotientenheuristik zur Vervollständigung der Sampling–Daten Daten einer Simulationsmessung benötigt, muß der Ablauf des zu untersuchenden Programms jedoch deterministisch sein. Kann dies nicht vorausgesetzt werden, so sind die jeweiligen Messergebnisse nicht in Zusammenhang zu bringen und die Quotientenheuristik ist dann nicht mehr anwendbar. Auch dieser Punkt kann in nachfolgenden Arbeiten genau untersucht werden.

Bei der Programmierung des Zusatzmoduls traten einige Probleme auf. Da das Visualisierungsprogramm `KCachegrind` über keine definierte Schnittstelle zur Anbindung von Plug–Ins verfügt, gestaltete sich die Einbindung des Zusatzmoduls gelegentlich schwierig. An verschiedenen Stellen war es nötig den Code des Programms zu verändern, damit das Modul aufgerufen werden konnte. `KCachegrind` wurde allerdings während der Programmierung des Zusatzmoduls restrukturiert und mit einer Plug–In–Schnittstelle ausgestattet.

Das Modul konnte zum Zeitpunkt der Entwicklung diese Schnittstelle noch nicht nutzen. Der Algorithmus zur Berechnung der Quotientenheuristik ist im Hinblick auf Effizienz der Berechnung zu verbessern, da nicht alle Möglichkeiten von `KCachegrind` ausgenutzt werden konnten. Es muß auch noch darauf hingewiesen werden, daß der Algorithmus in eine Endlosschleife gerät, wenn das zu untersuchende Programm Rekursionen enthält. Die Lösung für dieses Problem liegt in der Zyklenauflösung, einer Funktion von KCachegrind. Aus zeitlichen Gründen konnte bei der Programmierung diese Funktionalität nicht mehr berücksichtigt werden.

7.2 Ausblick

Die vorgeschlagene Klassifizierung von Ereignistypen erfüllt nicht nur den Zweck der besseren Übersicht, sondern kann als Basis für weitere Projekte dienen. In Abschnitt 7.2.1 werden einige Vorschläge für den weiteren Einsatz angesprochen. Die Spezifikationssprache DESL muß bis zur tatsächlichen Implementierung weiter untersucht werden. Einige Vorschläge, die bei der Implementierung zu bedenken sind, werden in Abschnitt 7.2.2 besprochen. Schließlich widmet sich Abschnitt 7.2.3 der Quotientenheuristik. Das Zusatzmodul stellt einen Prototypen dar und kann in mehreren Punkten erweitert werden. Der Abschnitt behandelt die offenen Punkte der Implementierung und schlägt Verbesserungen des Moduls vor.

7.2.1 Klassifizierung der Ereignistypen

Wie im vorigen Abschnitt erwähnt, gestattet die Klassifizierung der Ereignistypen einen geordneten Überblick über die Möglichkeiten der Ereignismessung in einem Rechensystem. Die vorgeschlagene Einteilung erfolgte auf abstrakter Ebene, um die unterschiedlichen Rechensysteme zu erfassen. Zukünftige Arbeiten auf diesem Gebiet können die Entwicklung einer allgemeinen Bibliothek zur Performance–Messung mit wohldefinierter Schnittstelle zum Inhalt haben. Eine Performance–Analyse–Bibliothek kann sich die vorgeschlagene Klassifizierung zunutze machen und die Struktur der Bibliothek von ihr ableiten. Auf der Seite der Software existiert diese Bibliothek bereits durch die Performance Counter Klassen [17] des Betriebssystems Windows 2000. Bibliotheken wie die Performance Assertion Library PAL [39] und das /proc–Dateisystem stellen die Basis für eine solche Bibliothek des Betriebssystems Linux zur Verfügung. Die Bibliothek PAL gestattet die Anwendung von Metriken bei der Codeinstrumentierung. Für das /proc–Dateisystem existieren jedoch nur Hilfsprogramme wie `procinfo`, welche die Informationen für den Menschen lesbar machen, so daß auch hier eine Bibliothek von Nutzen ist, die den Zugriff innerhalb von Programmen zur Instrumentierung ermöglicht. Weiterhin ist es denkbar, in eine solche Bibliothek Performance–Analyse–Assistenten zu integrieren, die dem Benutzer Hinweise geben, an welcher Stelle das Programm geändert werden sollte, um eine Effizienzsteigerung zu erreichen. Mit dem VTune Performance Analyzer der Firma Intel [24] sind derartige Ansätze bereits in die Tat umgesetzt. Ein allgemeine Bibliothek diente also der Instrumentierung des Sourcecodes, wie auch der standardisierten Entwicklung von Analyseprogrammen.

7.2.2 DESL

Die Spezifikationssprache DESL wurde in EBNF angegeben und kann in zukünftigen Arbeiten implementiert werden. DESL muß dazu gründlich untersucht werden, um die Komplexität der Spezifikationsausdrücke zu verringern. Es wird vorgeschlagen, die Spezifikation der Ereignistypen als Objekte zu implementieren und den Zugriff auf Felder eines anderen Ereignisses über die Objekt–ID und den Feldnamen zu qualifizieren. Hinter DESL stak die Vorstellung, daß die Daten der Ereignistypen in Tabellen gespeichert sind, auf deren Einträge durch die Spezifikation von Berechnungsvorschriften zugegriffen wird. Dieser Zugriff kann mit Hilfe eines Datenbankmoduls implementiert werden. Die Datenbankabfragesprache SQL könnte genügend Funktionalität zur Verfügung stellen. DESL würde dann als Adapter für den Datenbanktreiber fungieren und die Kommandos des Treiber kapseln. In Abschnitt 7.1.2 wurde erwähnt, daß DESL ein funktionales Berechnungsschema zugrunde gelegt werden könnte. Die Beschreibung von Ereignistypen als Funktionen hätte dann auch den Vorteil, daß die Einträge, welche die aufgetretenen Ereignisse darstellen nicht mehr verändert werden müssten.

7.2.3 Quotientenheuristik

Es wird vorgeschlagen, in weiterführenden Arbeiten das Zusatzmodul einer Revision zu unterziehen, so daß die mittlerweile verfügbare Plug–In–Schnittstelle genutzt werden kann. Des weiteren nutzt die Heuristik zum gegenwärtigen Zeitpunkt nur den ersten verfügbaren Ereignistyp, den `KCachegrind` anbietet, zur Berechnung. Hier ist es sinnvoll, das Programm so zu verändern, daß der Ereignistyp zur Berechnung der Inklusivkosten vom Benutzer ausgewählt werden kann und auch unterschiedliche Ereignistypen, durch Formeln verknüpft herangezogen werden können. Da sich die Qualität der Quotientenheuristik bewies, ist es von Vorteil, daß die angereicherten Sampling–Daten gespeichert werden können. Das Zusatzmodul unterstützt diese Funktion noch nicht. Die Entwicklung einer Funktion zur Speicherung der angereicherten Daten im Format von `KCachegrind` ermöglicht die Vergleichbarkeit der angereicherten Daten, gerade wenn unterschiedliche Ereignistypen als Datenbasis herangezogen werden.

Die Erstellung von Programmprofilen ist für den Anwendungsentwickler von besonderer Bedeutung, wenn das Programm in Hinblick auf Effizienz beziehungsweise Ausführungsgeschwindigkeit nicht den Erwartungen entspricht. So ist der Entwickler daran interessiert Leistungsengpässe zu identifizieren und die Gründe dafür in den Programmcode zurückverfolgen zu können. Bis heute ist für diese Analyse von Programmen sehr viel Erfahrung und oftmals Glück vonnöten, können die Ursachen doch vielfältiger Natur sein. Die vorliegende Arbeit hilft durch die Klassifizierung der Ereignistypen, also der Indikatoren für Leistungsengpässe dem Glück auf die Sprünge. Die übersichtliche Darstellung relevanter Ereignistypen macht diese Basis der Profilanalyse durchschaubarer. Die vorgeschlagene Spezifikationssprache soll einen Anstoß geben, wie einerseits Heuristiken in die Profilmessung einfließen können, zum anderen soll sie auch den Grundstein für weiterführende Arbeiten legen, die Ereignismessung zu vereinfachen. Die Quotientenheuristik stellt schließlich ein konkretes Beispiel dar, wie die Kombination von Ereignistypen durch eine Heuristik unvollständige Informationen anreichern kann. Zusammengefasst wird dem Programmierer das

Thema der Profilmessung näher gebracht, so daß die Erstellung eines Profils durchschaubarer wird.

Speed, it seems to me, provides the one genuinely modern pleasure.

ALDOUS HUXLEY

A Klassifizierung der Ereignistypen

A.1 Hardware

Die Abbildung A.1 zeigt die vorgeschlagene Klassifizierung der Ereignistypen des Abschnitts 3.1 im Gesamtüberblick.

A.2 Software

Die Abbildung A.2 zeigt die vorgeschlagene Klassifizierung der Ereignistypen des Abschnitts 3.2 im Gesamtüberblick.

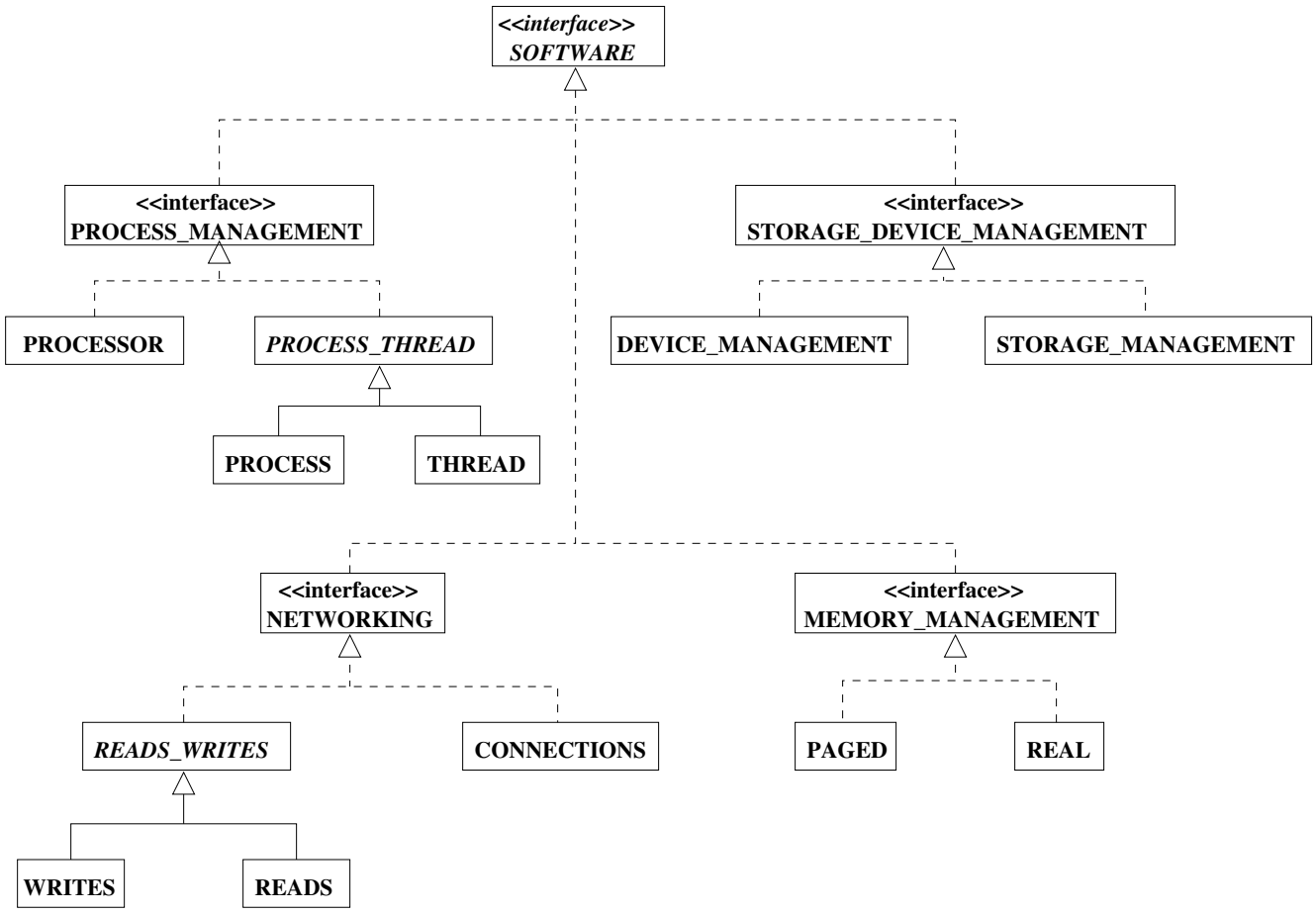


Abbildung A.2: Überblick über die Klassifizierung der Ereignistypen in der Software.

B Realisierung der Ereignistypklassen in der Hardware

In den folgenden Abschnitten wird die Umsetzung der Klassifizierung aus Kapitel 3 in der Hardware beleuchtet. Beginnend mit den Prozessoren geben die Tabellen in den Abschnitten B.1.1 bis B.1.6 an, ob Ereignisse eines bestimmten Ereignistyps im jeweiligen Prozessor durch einen Hardwarezähler gemessen werden können oder nicht. Für die Vorgehensweise zur Programmierung des jeweiligen Zählers ist [3, 6, 9] heranzuziehen.

B.1 Prozessoren

Die Umsetzung der Ereignistypen in der Hardware der in Kapitel 4, Abschnitt 4.1 vorgestellten Prozessoren wird in diesem Abschnitt behandelt. Jeder Unterabschnitt stellt die vorgeschlagene Schnittstelle dar, unter der in Tabellen die Namen der Ereignistypen aufgeführt sind. Ein $\surd$ bedeutet, daß der jeweilige Prozessor diesen Ereignistyp unterstützt, der Eintrag $\times$ hingegen bedeutet, daß dieser Typ nicht unterstützt wird. Die jeweiligen Prozessoren enthalten eine Vielzahl weiterer, hier nicht behandelter Ereignistypen, die meist architekturabhängig sind. Ausführlich werden diese Details in den bereits erwähnten Quellen [3, 6, 9] beschrieben.

B.1.1 Das Interface `BRANCH_EVENTS`

Das Interface `BRANCH_EVENTS` fasst Ereignistypen der Sprungzielvorhersage zusammen (siehe Abschnitt 3.1.1.1). Die Klassen `BRANCH_INSTRUCTIONS`, `BRANCH_PREDICTION`, `BRANCH_MISPREDICTED` und `BRANCH_CORRECT_PREDICTED` werden in den Tabellen B.1, B.2 und B.3 behandelt.

B.1.1.1 Die Klasse `BRANCH_INSTRUCTIONS`

Die Ereignistypen zur Überwachung bedingter und unbedingter Verzweigungen sind in der Klasse `BRANCH_INSTRUCTIONS` in Abschnitt 3.1.1.2 zusammengefasst.

B.1.1.2 Die Klasse `BRANCH_PREDICTION`

Die Ereignistypen der Klasse `BRANCH_PREDICTION` beschreiben die bedingten Verzweigungen. Der Abschnitt 3.1.1.3 erklärt die Bedeutung der Ereignistypen.

Ereignistypen der Klasse `BRANCH_INSTRUCTIONS`

Ereignistyp	P6	Itanium	PowerPC 604e
BRANCH_INSTRUCTIONS_DECODED	×	×	×
BRANCH_INSTRUCTIONS_RETIRED	×	×	×
BRANCH_INSTRUCTIONS_UNCONDITIONAL_RETIRED	×	×	×
BRANCH_INSTRUCTIONS_CONDITIONAL_RETIRED.d	√	√	√
BRANCH_INSTRUCTIONS_DECODED_RETIRED_RATIO.d	×	×	×
BRANCH_INSTRUCTIONS_RETIRED_PER_CYCLE.d	×	×	×

Tabelle B.1: Ereignistypen der Klasse BRANCH_INSTRUCTIONS und ihre Verfügbarkeit.

Ereignistypen der Klasse `BRANCH_PREDICTION`

Ereignistyp	P6	Itanium	PowerPC 604e
BRANCH_TAKEN	√	√	×
BRANCH_NOT_TAKEN	√	√	×
BRANCH_ALL.d	√	√	√

Tabelle B.2: Ereignistypen der Klasse BRANCH_PREDICTION und ihre Verfügbarkeit.

B.1.1.3 Die Klasse `BRANCH_MISPREDICTED`

In der Klasse BRANCH_MISPREDICTED finden sich Ereignistypen, die eine falsche Sprungzielvorhersage beschreiben. Tabelle B.3 zeigt, welche Prozessoren die Ereignistypen messen können.

Ereignistypen der Klasse `BRANCH_MISPREDICTED`

Ereignistyp	P6	Itanium	PowerPC 604e
BRANCH_MISPREDICTED_WRONG_PATH	×	√	×
BRANCH_MISPREDICTED_WRONG_TARGET	×	√	×

Tabelle B.3: Ereignistypen der Klasse BRANCH_MISPREDICTED und ihre Verfügbarkeit.

Die übrigen Ereignistypen, welche von BRANCH_PREDICTION ererbt sind, sind bei allen Prozessoren verfügbar.

B.1.1.4 Die Klasse `BRANCH_CORRECT_PREDICTED`

Die Klasse BRANCH_CORRECT_PREDICTED enthält Ereignistypen, die die korrekte Sprungzielvorhersage beschreiben. Die von BRANCH_PREDICTION ererbten Ereignistypen sind bei allen Prozessoren implementiert.

B.1.2 Das Interface `INSTRUCTION_EXECUTION`

Mit dem Interface werden die Ereignistypen zur Beschreibung der Instruktionsausführung zusammengefasst. Aus den folgenden Tabellen ist zu erkennen, ob die jeweiligen Prozessoren die Ereignistypen implementieren.

B.1.2.1 Die Klasse EPIC_EVENTS

Die Ereignistypen der Klasse EPIC_EVENTS, sowie die Typen der Subklassen EPIC_INSTRUCTION_ISSUE_AND_RETIREMENT, EPIC_CONTROL_AND_DATA_SPECULATION und EPIC_MISS_RATIOS des Abschnitts 3.1.1.7 sind nur bei dem Intel Itanium zu finden.

B.1.2.2 Die Klasse GENERAL_EXECUTION

Die Implementierung der Ereignistypen der Klasse GENERAL_EXECUTION ist in Tabelle B.4 dargestellt. In Abschnitt 3.1.1.12 sind die Ereignistypen zur Überwachung der allgemeinen Instruktionsausführung beschrieben.

Ereignistypen der Klasse GENERAL_EXECUTION

Ereignistyp	P6	Itanium	PowerPC 604e
INSTRUCTION_PREFETCHES	×	×	×
INSTRUCTION_DECODED	√	√	√
INSTRUCTIONS_RETIRED	√	√	√
INSTRUCTION_NOP_RETIRED	×	√	×
INSTRUCTION_DECODE_RETIRE_RATIO.d	√	√	√
INSTRUCTION_NOP_NONOP_RATIO.d	×	√	×

Tabelle B.4: Ereignistypen der Klasse GENERAL_EXECUTION und ihre Verfügbarkeit.

B.1.2.3 Die Klasse INTEGER_EXECUTION

In der Klasse INTEGER_EXECUTION werden Ereignistypen vorgeschlagen, die Ereignisse der Integereinheit beschreiben. Die Tabelle B.5 zeigt, bei welchem Prozessor diese Ereignistypen zu finden sind.

Ereignistypen der Klasse INTEGER_EXECUTION

Ereignistyp	P6	Itanium	PowerPC 604e
INTEGER_INSTRUCTIONS_RETIRED	×	√	√
INTEGER_ADD_INSTRUCTIONS_RETIRED	×	√	×
INTEGER_SUB_INSTRUCTIONS_RETIRED	×	√	×
INTEGER_MUL_INSTRUCTIONS_RETIRED	×	√	×
INTEGER_DIV_INSTRUCTIONS_RETIRED	×	√	×

Tabelle B.5: Ereignistypen der Klasse INTEGER_EXECUTION und ihre Verfügbarkeit.

B.1.2.4 Die Klasse FP_EXECUTION

Die Ereignistypen der Klasse FP_EXECUTION beschreiben Ereignisse der Fließkommaeinheit der CPU. In Tabelle B.6 ist angegeben, welche Prozessoren das Auftreten von Ereignisse der betreffenden Ereignistypen mit Hardwarezählern messen können.

Ereignistypen der Klasse `FP_EXECUTION`

Ereignistyp	P6	Itanium	PowerPC 604e
`FP_INSTRUCTIONS_RETIRED`	√	√	√
`FP_MUL_INSTRUCTIONS_RETIRED`	√	√	×
`FP_DIV_INSTRUCTIONS_RETIRED`	√	√	×
`FP_ADD_INSTRUCTIONS_RETIRED`	×	√	×
`FP_SUB_INSTRUCTIONS_RETIRED`	×	√	×
`FP_FLOPRATE.d`	√	√	√

Tabelle B.6: Ereignistypen der Klasse `FP_EXECUTION` und ihre Verfügbarkeit.

B.1.2.5 Die Klasse `MEMORY_EVENTS`

Die Klasse `MEMORY_EVENTS` in Abschnitt 3.1.1.13 beschreibt Ereignistypen, welche sich auf Ereignisse der LSU beziehen. In Tabelle B.7 ist angegeben, mit welchem Prozessor derartige Ereignisse gezählt werden können.

Ereignistypen der Klasse `MEMORY_EVENTS`

Ereignistyp	P6	Itanium	PowerPC 604e
`MEMORY_LOADS_RETIRED`	×	√	√
`MEMORY_STORES_RETIRED`	×	√	√
`MEMORY_UC_LOADS_RETIRED`	×	√	×
`MEMORY_UC_STORES_RETIRED`	×	√	×
`MEMORY_UNALIGNED_LOADS_RETIRED`	×	√	√
`MEMORY_UNALIGNED_STORES_RETIRED`	×	√	√
`MEMORY_UNALIGNED_MEM_REFS`	√	√	√
`MEMORY_ALIGNED_LOADS_RETIRED.d`	×	√	√
`MEMORY_ALIGNED_STORES_RETIRED.d`	×	√	√
`MEMORY_ALIGNED_MEM_REFS.d`	×	√	√

Tabelle B.7: Ereignistypen der Klasse `MEMORY_EVENTS` und ihre Verfügbarkeit.

B.1.3 Das Interface `CYCLE_ACCOUNTING`

Die Ereignistypen der Klassen unter dem Interface `CYCLE_ACCOUNTING` des Abschnitts 3.1.1.14 sollen es ermöglichen, den Verbrauch von Taktzyklen den einzelnen Verursachern zuschreiben zu können.

B.1.3.1 Die Klasse `BRANCH_CYCLES`

Die Klasse `BRANCH_CYCLES` in Abschnitt 3.1.1.15 befasst sich mit dem Verbrauch an Taktzyklen, die der BPU zuzuschreiben sind. In Tabelle B.8 ist angegeben, ob der jeweilige Prozessor dazu in der Lage ist.

Ereignistypen der Klasse `BRANCH_CYCLES`

Ereignistyp	P6	Itanium	PowerPC 604e
`CYCLES_BPU_IDLE`	×	×	√
`CYCLES_BPU_MISPREDICTED`	×	√	×
`CYCLES_BPU_TAKEN_BRANCH`	×	√	×
`CYCLES_BPU_STALL.d`	×	√	×

Tabelle B.8: Ereignistypen der Klasse `BRANCH_CYCLES` und ihre Verfügbarkeit.

B.1.3.2 Die Klasse `INSTRUCTION_CYCLES`

In Abschnitt 3.1.1.16 werden Ereignistypen in der `INSTRUCTION_CYCLES` zusammengefasst, die den Verbrauch von Taktzyklen dem Laden von Instruktionen zuschreiben. Die Tabelle B.9 zeigt welche Prozessoren diese Typen zur Verfügung stellen.

Ereignistypen der Klasse `INSTRUCTION_CYCLES`

Ereignistyp	P6	Itanium	PowerPC 604e
`CYCLES_EXEC_LATENCY`	×	√	√
`CYCLES_FPU_STALL`	×	×	√
`CYCLES_INSTR_ACCESS`	√	√	×
`CYCLES_INSTR_ISSUE`	×	√	√
`CYCLES_SCOREBOARD_STALL.d`	×	×	√
`CYCLES_EXEC_STALL.d`	×	√	√

Tabelle B.9: Ereignistypen der Klasse `INSTRUCTION_CYCLES` und ihre Verfügbarkeit.

B.1.3.3 Die Klasse `MEMORY_CYCLES`

Der Ereignistyp dieser Klasse aus Abschnitt 3.1.1.17 schreibt die verbrauchten Wartezyklen den Speicherzugriffen zu. Tabelle B.10 zeigt, welcher Prozessor auf Ereignisse dieses Typs reagieren kann.

Ereignistypen der Klasse `MEMORY_CYCLES`

Ereignistyp	P6	Itanium	PowerPC 604e
`CYCLES_DATA_ACCESS`	×	√	√[1]

Tabelle B.10: Ereignistyp der Klasse `MEMORY_CYCLES` und seine Verfügbarkeit.

B.1.4 Das Interface `BASIC_EVENTS`

Das Interface `BASIC_EVENTS` in Abschnitt 3.1.1.18 umfasst in der implementierenden Klasse `INSTRUCTION_DECODE_AND_RETIREMENT` die Ereignistypen zur Beobachtung der In-

[1] Der Ereignistyp kann berechnet werden, wenn das Ereignis 9 und das Ereignis 10 über die Bits 0–4 und 5–9 im MMCR1–Konfigurationsregister ausgewählt wird. Das Ergebnis wird durch Addition der Werte der Zählerregister PMC3 und PMC4 berechnet.

struktionsverarbeitung.

B.1.4.1 Die Klasse INSTRUCTION_DECODE_AND_RETIREMENT

In Tabelle B.11 ist angegeben, welcher Prozessor Ereignisse der entsprechenden Typen der Klasse INSTRUCTION_DECODE_AND_RETIREMENT zählen kann.

Ereignistypen der Klasse INSTRUCTION_DECODE_AND_RETIREMENT

Ereignistyp	P6	Itanium	PowerPC 604e
CPU_CYCLES	√	√	√
INST_DECODED	√	×	√
INST_RETIRED	√	√	×
INST_DISPATCHED	×	√	√
INST_DECODED_PER_CYCLE.d	√	×	√
INST_RETIRED_PER_CYCLE.d	√	√	×
INST_RETIRED_DECODE_RATIO.d	√	√	×

Tabelle B.11: Ereignistypen der Klasse INSTRUCTION_DECODE_AND_RETIREMENT und ihre Verfügbarkeit.

B.1.5 Das Interface SYSTEM_EVENTS

In Abschnitt 3.1.1.20 bündelt das Interface die zwei Klassen PROCESSOR_SYSTEM_EVENTS und PROCESSOR_TLB_PERFORMANCE. Mit den Ereignistypen dieser Klassen sollen Performance–Aussagen über die Anzahl von System–Calls etc. gemacht werden können.

B.1.5.1 Die Klasse PROCESSOR_SYSTEM_EVENTS

Die Verfügbarkeit von Ereignistypen zur Zählung von Systemereignissen in den einzelnen Prozessoren ist in Tabelle B.12 angegeben (siehe Abschnitt 3.1.1.21).

B.1.5.2 Die Klasse TLB_PERFORMANCE

In Abschnitt 3.1.1.22 werden Ereignistypen vorgeschlagen, die eine Performance–Analyse des TLB ermöglichen sollen. In Tabelle B.13 ist angegeben, welcher Prozessor dazu in der Lage ist.

B.1.6 Das Interface MEMORY_HIERARCHY

Unter dem Interface in Abschnitt 3.1.1.23 sind die Ereignistypen der hierarchischen Cache–Struktur im Prozessor zusammengefasst. Im Folgenden werden die Klassen L1_CACHE_DATA, L1_CACHE_INSTR, L2_CACHE, L3_CACHE und CACHE_PERFORMANCE noch einmal aufgegriffen und angegeben, ob die Prozessoren die jeweiligen Ereignistypen unterstützen.

Ereignistypen der Klasse PROCESSOR_SYSTEM_EVENTS

Ereignistyp	P6	Itanium	PowerPC 604e
PROCESSOR_CPL_CHANGES	√[2]	√	√[3]
PROCESSOR_SYSTEM_CALLS	×	×	√
PROCESSOR_PIPELINE_FLUSHES_BRANCH	×	×	×
PROCESSOR_PIPELINE_FLUSHES_EXCEPTION	×	×	×
PROCESSOR_BUS_REQ_OUTSTANDING	√	×	×
PROCESSOR_BUS_TRAN_MEM	√	×	×
PROCESSOR_BUS_TRAN_IFETCH	√	×	×
PROCESSOR_BUS_TRAN_IO	√	×	×
PROCESSOR_BUS_DATA_RCV	√	×	×
PROCESSOR_BUS_LOAD	×	×	×
PROCESSOR_INTERRUPT_HW	×	×	×
PROCESSOR_PIPELINE_FLUSHES_ALL.d	×	√	√
PROCESSOR_RFI_CPL_CHANGES.d	×	√	×
PROCESSOR_INTERRUPTS_IN.d	√	×	×

Tabelle B.12: Ereignistypen der Klasse PROCESSOR_SYSTEM_EVENTS und ihre Verfügbarkeit.

B.1.6.1 Die Klassen L1_CACHE_DATA und L1_CACHE_INSTR

In Tabelle B.14 ist angegeben, welche Prozessoren Ereignisse der Ereignistypen dieser Klassen mit Zählerregistern erfassen können (siehe Abschnitt 3.1.1.25 und 3.1.1.26).

B.1.6.2 Die Klasse L2_CACHE

In Abschnitt 3.1.1.27 sind die Ereignistypen zur Performance–Analyse des L2–Caches beschrieben. Über die Möglichkeiten der einzelnen Prozessoren gibt Tabelle B.15 Auskunft.

B.1.6.3 Die Klasse L3_CACHE

In Abschnitt werden Ereignistypen zur Überwachung des L3–Caches vorgestellt. Dieser dritte Speicher in der Speicherhierarchie der Caches ist bisher nur bei den Prozessoren XEON MP und DP, sowie Itanium des Herstellers Intel verfügbar. Alle Ereignistypen sind im Itanium–Prozessor zu finden.

[2] Der Pentium P6 kann die Anzahl der empfangenen Hardware–Interrupts zählen. Der Ereignistyp HW_INT_RX ist eine Annäherung an den Wert des Ereignistyps PROCESSOR_CPL_CHANGES.

[3] Mit dem PowerPC 604e ist die Zählung der System–Calls über das Register PMC3 und Auswahl des Ereignistyps 11 mit Hilfe des Konfigurationsregisters MMCR1 möglich. Der gemessene Wert kann als Annährung an den Wert des Ereignistyps PROCESSOR_CPL_CHANGES betrachtet werden.

Ereignistypen der Klasse `TLB_PERFORMANCE`

Ereignistyp	P6	Itanium	PowerPC 604e
PROCESSOR_ITLB_REFERENCES	×	√	×
PROCESSOR_ITLB_MISSES	×	√	√
PROCESSOR_DTLB_REFERENCES	×	√	×
PROCESSOR_DTLB_MISSES	×	√	√
PROCESSOR_DTLB_MISS_RATIO.d	×	√	×
PROCESSOR_ITLB_MISS_RATIO.d	×	√	×
PROCESSOR_TOTAL_MISSES.d	×	√	√
PROCESSOR_ITLB_HITS.d	×	√	×
PROCESSOR_DTLB_HITS.d	×	√	×

Tabelle B.13: Ereignistypen der Klasse `TLB_PERFORMANCE` und ihre Verfügbarkeit.

Ereignistypen der Klassen `L1_CACHE_DATA` und `L1_CACHE_INSTR`

Ereignistyp	P6	Itanium	PowerPC 604e
L1_DATA/INSTR_REFERENCES	√/√	√/√	× / ×
L1_DATA/INSTR_MISSES	√/√	√/√	√/√
L1_DATA/INSTR_HITS.d	√/√	√/√	× / ×

Tabelle B.14: Ereignistypen der Klassen `L1_CACHE_DATA` und `L1_CACHE_INSTR` und ihre Verfügbarkeit.

B.1.6.4 Die Klasse `CACHE_PERFORMANCE`

In Abschnitt 3.1.1.29 sind abgeleitete Ereignistypen zusammengefasst, die die Leistung der Caches beschreiben. In Tabelle B.16 ist angegeben, welche dieser Metriken mit den betrachteten Prozessoren messbar sind.

Ereignistypen der Klasse `L2_CACHE`

Ereignistyp	P6	Itanium	PowerPC 604e
`L2_INSTR_PREFETCHES`	×	√	×
`L2_INSTR_FETCHES`	√	√	×
`L2_DATA_REFERENCES`	√	√	×
`L2_DATA_READS`	√	√	√
`L2_MISSES`	×	√	×
`L2_DATA_MISSES`	×	√	×
`L2_INSTR_REFERENCES.d`	√	√	√
`L2_TOTAL_REFERENCES.d`	√	√	×
`L2_DATA_WRITES.d`	√	√	√[4]
`L2_DATA_HITS.d`	×	√	×
`L2_INSTR_MISSES.d`	×	√	×
`L2_INSTR_HITS.d`	×	√	×

Tabelle B.15: Ereignistypen der Klasse `L2_CACHE` und ihre Verfügbarkeit.

[4]Ereignisse dieses Typs können beim PowerPC 604e direkt gemessen werden.

Ereignistypen der Klasse `CACHE_PERFORMANCE`

Ereignistyp	P6	Itanium	PowerPC 604e
`L1_INSTR_MISS_RATIO.d`	√	√	×
`L1_DATA_MISS_RATIO.d`	√	√	×
`L2_MISS_RATIO.d`	×	√	×
`L2_DATA_MISS_RATIO.d`	×	√	×
`L2_INSTR_MISS_RATIO.d`	×	√	×
`L2_DATA_READ_MISS_RATIO.d`	×	√	×
`L2_DATA_WRITE_MISS_RATIO.d`	×	√	×
`L2_INSTRUCTION_FETCH_RATIO.d`	√	√	×
`L2_DATA_RATIO.d`	√	√	×
`L3_MISS_RATIO.d`	×	√	×
`L3_DATA_MISS_RATIO.d`	×	√	×
`L3_INSTR_MISS_RATIO.d`	×	√	×
`L3_DATA_READ_RATIO.d`	×	√	×
`L3_DATA_RATIO.d`	×	√	×

Tabelle B.16: Ereignistypen der Klasse `CACHE_PERFORMANCE` und ihre Verfügbarkeit.

C Realisierung der Ereignistypklassen in der Software

Die folgenden Tabellen zeigen, ob die vorgeschlagenen Ereignistypen der Software aus Abschnitt 3.2 bei den betrachteten Betriebssystemen verfügbar sind oder nicht. Dazu werden die einzelnen Klassen beginnend mit dem Interface `PROCESS_MANAGEMENT` des Abschnitts 3.2.1 nochmals aufgegriffen.

C.1 Das Interface `PROCESS_MANAGEMENT`

Unter dem Interface `PROCESS_MANAGEMENT` sind in Abschnitt 3.2.1 die Klassen `PROCESSOR`, `PROCESS_THREAD`, `PROCESS` und `THREAD` zusammengefasst. Die Ereignistypen dieser Klassen beschreiben das Prozessmanagement des Betriebssystems.

C.1.1 Die Klasse `Processor`

Tabelle C.1 zeigt die Verfügbarkeit der Ereignistypen des Unterabschnitts 3.2.1.1.

Ereignistypen der Klasse `PROCESSOR`

Ereignistyp	Windows 2000	Linux
`PROCESSOR_INTERRUPTS`	√	√
`PROCESSOR_PRIVILEGED_MODE`	×	×
`PROCESSOR_PRIVILEGED_TIME`	√	√
`PROCESSOR_USER_MODE`	×	×
`PROCESSOR_USER_TIME`	√	√
`PROCESSOR_QUEUE_LENGTH`	√	√
`PROCESSOR_PROCESSOR_TIME.d`	√	√
`PROCESSOR_PROCESSOR_MODE.d`	×	×

Tabelle C.1: Ereignistypen der Klasse `PROCESSOR` und ihre Verfügbarkeit.

C.1.2 Die abstrakte Klasse `PROCESS_THREAD`

In Abschnitt 3.2.1.2 sind Ereignistypen beschrieben, die das Verhalten von Prozessen und Threads charakterisieren. Tabelle C.2 zeigt, welche der Ereignistypen innerhalb der verschiedenen Betriebssysteme gemessen werden können.

Ereignistypen der Klasse PROCESS_THREAD

Ereignistyp	Windows 2000	Linux
PROCESS_THREAD_CONTEXT_SWITCHES	√	√
PROCESS_THREAD_SYS_CALLS	√	×
PROCESS_THREAD_EXCEPTIONS	√	×
PROCESS_THREAD_PAGE_FAULTS	√	√
PROCESS_THREAD_IO_DATA_OPS	√	×
PROCESS_THREAD_IO_READS	√	×
PROCESS_THREAD_IO_OTHER_OPS	√	×
PROCESS_THREAD_IO_WRITES.d	√	×

Tabelle C.2: Ereignistypen der Klasse PROCESS_THREAD und ihre Verfügbarkeit.

C.1.3 Die Klasse PROCESS

Der zusätzliche Ereignistyp PROCESS_THREADS der Klasse PROCESS in Abschnitt 3.2.1.3 gibt die Anzahl der Threads innerhalb eines Prozesses an. Dieser Ereignistyp ist sowohl unter Windows 2000, als auch unter Linux verfügbar.

C.1.4 Die Klasse THREAD

Die in Abschnitt 3.2.1.4 beschriebene Klasse enthält zusätzlich zu den ererbten Ereignistypen zwei weitere Ereignistypen, welche das Verhalten von Threads genauer beschreiben. Tabelle C.3 zeigt, welcher Ereignistyp von welchem Betriebssystem unterstützt wird.

Ereignistypen der Klasse THREAD

Ereignistyp	Windows 2000	Linux
THREAD_SYNC	√	×
THREAD_MUTEXES	√	×

Tabelle C.3: Ereignistypen der Klasse THREAD und ihre Verfügbarkeit.

C.2 Das Interface STORAGE_DEVICE_MANAGEMENT

In Abschnitt 3.2.2 werden unter dem Interface die Klassen STORAGE_MANAGEMENT und DEVICE_MANAGEMENT zusammengefasst. Die Ereignistypen dieser Klassen beschreiben das Verhalten der Dateisysteme und der Speichergeräte. Die folgenden Unterabschnitte C.2.2 und C.2.1 beschreiben tabellarisch die Verfügbarkeit der Typen in den jeweiligen Betriebssystemen.

C.2.1 Die Klasse DEVICE_MANAGEMENT

Die Ereignistypen der Klasse DEVICE_MANAGEMENT in Abschnitt 3.2.2.1 beschreiben die Performance von Speichergeräten. Tabelle C.4 zeigt, ob der jeweilige Ereignistyp unterstützt

wird.

Ereignistypen der Klasse DEVICE_MANAGEMENT

Ereignistyp	Windows 2000	Linux
DEVICE_REQUESTS	√	×
DEVICE_REQ_OUTSTANDING	√	×
DEVICE_REQ_READ	√	×
DEVICE_SPLIT_IO	√	×
DEVICE_REQ_WRITE.d	√	×

Tabelle C.4: Ereignistypen der Klasse DEVICE_MANAGEMENT und ihre Verfügbarkeit.

C.2.2 Die Klasse STORAGE_MANAGEMENT

Die Ereignistypen der Klasse STORAGE_MANAGEMENT in Abschnitt 3.2.2.2 beschreiben Ereignisse des Dateisystems. In Tabelle C.5 wird angegeben, welches Betriebssystem die Zählung von Ereignissen der jeweiligen Ereignistypen unterstützt.

Ereignistypen der Klasse STORAGE_MANAGEMENT

Ereignistyp	Windows 2000	Linux
STORAGE_CONTROL_OPS	√	√[1]
STORAGE_DATA_OPS	√	√[2]
STORAGE_FILE_READ_OPS	√	√
STORAGE_WRITE_OPS.d	√	√

Tabelle C.5: Ereignistypen der Klasse STORAGE_MANAGEMENT und ihre Verfügbarkeit.

C.3 Das Interface MEMORY_MANAGEMENT

Das Interface in Abschnitt 3.2.3 umfasst mit der Klasse PAGED Ereignistypen, die das Verhalten des virtuellen Speichers beschreiben. Der Abschnitt C.3.1 zeigt die Verfügbarkeit bei den Betriebssystemen.

C.3.1 Die Klasse PAGED

Alle Ereignisse der Ereignistypen dieser Klasse sind mit den Performance Counter Classes von Windows 2000, beziehungsweise mit Hilfe des /proc–Dateisystems von Linux messbar.

[1] Im /proc–Dateisystem existiert das Verzeichnis stat/disk_io. Die Ausgabe des Verzeichnisses liefert unter anderem einen Wert für den Eintrag noinfo. Dieser Wert stellt die Kontrolloperationen dar.

[2] Im /proc–Dateisystem existieren Zähler für die Ereignistypen STORAGE_FILE_READ_OPS und STORAGE_WRITE_OPS.d. Der entsprechende Wert kann aus diesen beiden Werten berechnet werden.

C.3.2 Das Interface NETWORKING

In Abschnitt 3.2.4 sind unter dem Interface die abstrakte Klasse READS_WRITES, mit den Subklassen READS und WRITES und die Klasse CONNECTIONS zusammengefasst. Die Ereignistypen dieser Klassen sollen das Verhalten des Netzwerksubsystems beschreiben. In den folgenden Abschnitten wird untersucht, ob die Betriebssysteme die Messung von Ereignissen der jeweiligen Ereignistypen unterstützen.

C.3.2.1 Die abstrakte Klasse READS_WRITES

In Tabelle C.6 ist angegeben welche der Ereignistypen der Klasse READS_WRITES des Abschnitts 3.2.4.1 unterstützt werden. Da die Klasse die abstrakte Überklasse von READS und WRITES ist, stellen die Einträge der Tabelle die Verfügbarkeit der ererbten Ereignistypen der Subklassen dar.

Ereignistypen der Klasse READS_WRITES

Ereignistyp	Windows 2000	Linux
READS_WRITES_DENIED	√	×
READS_WRITES_EXCEEDING_BUF_SIZE	√	×
READS_WRITES_UNDERRUN_BUF_SIZE	√	×
READS_WRITES_BYTES_RECEIVED	√	√

Tabelle C.6: Ereignistypen der Klasse READS_WRITES und ihre Verfügbarkeit.

C.3.2.2 Die Klasse CONNECTIONS

In Abschnitt 3.2.4.4 werden Ereignistypen vorgeschlagen, die grundlegende Eigenschaften der Netzwerkverbindungen beschreiben. In Tabelle C.7 ist angegeben, welcher Ereignistyp im jeweiligen Betriebssystem unterstützt wird.

Ereignistypen der Klasse CONNECTIONS

Ereignistyp	Windows 2000	Linux
CONNECTIONS_REQUESTS	√	√
CONNECTIONS_PACKETS	√	√
CONNECTIONS_SERVER_RECONNECTS	√	×

Tabelle C.7: Ereignistypen der Klasse CONNECTIONS und ihre Verfügbarkeit.

D Beispielprogramm für den Vergleich der Heuristiken

```
#include <time.h>
double a(double w), b(double w), c(double w), d(double w);

  __inline__ double dowork(int count)
{
  double i = 0.0;
    while(count>0){
      i++;
      i = i *50;
      i = i/10;
      i = i -42;
      i = i+9;
      i = i / 40 + 23;
      count-;
    }
  return i;
}

double a(double w)
{
  double t1 = (double) clock();
  double t2 = 0.0;
  double i = dowork(1000000);

  t2 = (double) clock();
  b(6*w);
  printf("call a->b: %f\n", clock() - t2);
  t2 = (double) clock();
  c(w);
  printf("call a->c: %f\n", clock() - t2);
  printf("Inkl a: %f\n", clock() -t1);
  return i;
}

double b(double w)
{
  double t1 = (double) clock();
```

```
  double t2 = 0.0;
  double i = 0.0;

  i = dowork(1000000);

  t2 = (double) clock();
  d(w);
  printf("call b->d: %f\n", clock() - t2);
  printf("Inkl b: %f\n", clock() -t1);
  return i;
}

double c(double w)
{
  double t1 = (double) clock();
  double t2 = 0.0;
  double i = 0.0;

  i = dowork(1000000);
  t2 = (double) clock();
  d(w);
  printf("call c->d: %f\n", clock() - t2);
  t2 = (double) clock();
  d(w);
  printf("call c->d: %f\n", clock() - t2);
  t2 = (double) clock();
  d(w);
  printf("call c->d: %f\n", clock() - t2);
  printf("Inkl c: %f\n", clock() -t1);
  return i;
}

double d(double w)
{
  double t1 = (double) clock();
  double i = dowork(w*1000000);
  printf("Inkl d: %f\n", clock() -t1);
  return i;
}



int main(int argc, char *argv[])
{
  double t1 = (double) clock();
  double t2 = 0.0;
  double i = 0.0;
```

```
  i = dowork(1000000);
  t2 = (double) clock();
  i += a(2);
  printf("call main->a: %f\n", clock() - t2);
  t2 = (double) clock();
  i += c(2);
  printf("call main->c: %f\n", clock() - t2);
  printf("result: %f\n", i);
  printf("Inkl main: %f\n", clock() -t1);
  return 1;
}
```

E Glossar

ALAT Advanced Load Address Table: Struktur der HP/Intel IA–64 VLIW/EPIC–Prozessorarchitektur, welche zur Kontroll– und Datenspekulation verwendet wird.

ALU Arithmetic Logic Unit: Komponenten des Rechenwerks, die die Verknüpfung von Daten ermöglicht.

BIU Bus Interface Unit: Einheit des Prozessors, welche die Anbindung an den Bus regelt.

BPU Branch Processing Unit, *auch* Branch Prediction Unit: Einheit des Prozessors, welche Sprünge und die Vorhersage der Sprungziele bearbeitet.

BTB Branch Trace Buffer: Puffer, in welchem die letzten ausgeführten Verzweigungen, die Ursprünge und Ziele der Verzweigungen abgelegt sind.

CACHE Zwischenspeicher für den schnellen Zugriff auf Instruktionen und Daten. Solche Puffer sind vom Hauptspeicher getrennt, werden aber mit Instruktionen und Daten aus dem Hauptspeicher bestückt.

CISC Complex Instruction Set Computer: Hardwarearchitektur, welche viele Maschinenbefehle unterschiedlicher Länge für komplexe Aufgaben besitzt.

DTLB Data TLB: TLB ausschließlich für Daten, siehe TLB

EBNF Extended Backus-Naur Form: Formale Beschreibung einer (Programmier–)Sprache.

EPIC Explicitly Parallel Instruction Computing: Prozessordesign, bei dem der Compiler entscheidet, welche Instruktionen parallel auszuführen sind.

FLOPS Floating Point Operations: Die Anzahl der ausgeführten Fließkommaoperationen.

FPU Floating Point Unit: Einheit des Prozessors zur Berechnung von Fließkommaoperationen.

IC Instruction Cache: Puffer in welchem vorausgeladene Instruktionen abgelegt werden.

ILP Instruction Level Parallelism: ein Maß wie viele Instruktionen eines Programms gleichzeitig verarbeitet werden können.

IPC Instructions Per Clock Cycle: Maß für die Angabe der bearbeiteten Instruktionen pro Taktzyklus.

ITLB Instruction TLB: TLB ausschließlich für Instruktionen, siehe TLB

IU Instruction Unit: Einheit des Prozessors, welche die Maschinenbefehle verarbeitet.

LSU Load and Store Unit: Einheit des Prozessors, welche für das Laden und Speichern von Werten in Registern zuständig ist.

MCIU Multiple Cycle Integer Unit: Einheit zur Berechnung von Integeroperationen, die diese innerhalb mehrerer Taktzyklen abarbeitet.

MSR Machine Status Register *auch* Model Specific Register: Das Prozessorstatuswort, oder ein für den zugrunde liegenden Prozessor spezifische Register.

NOP No Operation: Instruktion, welche den Zustand der Maschine nicht verändert, jedoch Taktzyklen bis zur Abarbeitung verbraucht.

NOPS *siehe* NOP

OLTP On–Line Transaction Processing: Typ der Informationsverarbeitung, bei dem der Computer sofort auf die Anfragen des Benutzers reagiert. Verwendet zum Beispiel beim On–Line–Banking.

out–of–order execution : Maschinenbefehle werden in anderer Reihenfolge ausgeführt, als es das Programm vorsieht. Der ILP kann erhöht werden, wenn unabhängige Befehle gleichzeitig ausgeführt werden, auch wenn durch das Programm diese Befehle in einer Reihenfolge ausführt würden, *siehe* ILP und SCOREBOARD.

Overhead Zusätzlicher Rechen– oder Zeitaufwand, der durch eine Methode der Verarbeitung oder Messung entsteht.

PC Programm Counter: der Instruktionszeiger.

PEBS Precise Event–Based Sampling: Das Auftreten von Ereignissen wird mit Detailangaben über den Ort des Auftretens, die Registerinhalt etc. in den Attributen versehen.

PMI Performance Monitor Interrupt: spezielle Implementierung des IBM PowerPC, bei der ein Interrupt ausgelöst wird, wenn eine mit dem Auftreten von Ereignissen verknüpfte Bedingung erfüllt ist.

RAW Read After Write: Ein Datum wird gelesen, nachdem es verändert wurde. Diese Situation kann zu unerwünschten Ergebnissen führen.

RISC Reduced Instruction Set Computer: Hardwarearchitektur, welche nur wenige Maschinenbefehle konstanter Länge besitzt.

SCIU Single Cycle Integer Unit: Einheit zur Berechnung von Integeroperationen, die diese innerhalb eines Taktzyklus abarbeiten kann.

Scoreboard : Einheit des Prozessors, um Datenabhängigkeiten, die die out–of–order execution behindern aufzulösen, *siehe* OUT–OF–ORDER EXECUTION.

SDA Sampled Data Address: Adresse des zuletzt verwendeten Datums.

SIMD Single Instruction Multiple Data: Mit einer Instruktion können mehrere gleichzeitig Daten verarbeitet werden.

SIA Sampled Instruction Address: Adresse des aktuell abgearbeiteten Maschinenbefehls.

SLOC Source Lines Of Code: geschätzte Anzahl der Quellcodezeilen eines Programms ohne Kommentar und Leerzeilen.

TLB Translation Lookaside Buffer: Puffer, in welchem die den virtuellen Speicheradressen entsprechenden physikalischen Adressen abgelegt sind.

VLIW Very Long Instruction Word: Der Compiler setzt aus Maschinenbefehlen ein langes Befehlswort zusammen. Dieses Befehlswort definiert für jeden Startzeitpunkt von Befehlen genau einen Maschinenbefehl für jede vorhandene ALU.

Literaturverzeichnis

[1] *Das KCachegrind Handbuch*.

[2] *GNU gprof, The GNU Profiler*.

[3] *IA–32 Intel® Architecture, Software Developer's Manual, Volume 3: System Programming Guide*.

[4] *Intel® IA–64 Architecture Software Developer's Manual, Rev. 1.0, Volume 1: IA–64 Application Architecture*.

[5] *Intel® IA–64 Architecture Software Developer's Manual, Rev. 1.0, Volume 2: IA–64 System Architecture*.

[6] *Intel® IA–64 Architecture Software Developer's Manual, Rev. 1.0, Volume 4: Itanium™ Processor Programmer's Guide*.

[7] *Linux Programmer's Manual, proc – process information pseudo–filesystem*.

[8] *PAPI User's Guide, Version 2.3*.

[9] *PowerPC 604e™ RISC Microprocessor User's Manual with Supplement for PowerPC 604e™ Microprocessor*.

[10] *Computer Architecture – A quantitive approach*, Morgan Kaufmann Publishers, Inc., 1990, ch. 6 Pipelining.

[11] *Informatik–Handbuch*, Hanser Verlag München Wien, 1997.

[12] *Object–Oriented Software Engineering: Conquering Complex and Changing Systems*, Alan Apt, 2000.

[13] D. I. AUGUST, D. A. CONNORS, S. A. MAHLKE, J. W. SIAS, K. M. CROZIER, B. C. CHENG, P. R. EATON, Q. B. OLANIRAN, and W. M. W. HWU, *Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture*, IEEE, 1998.

[14] T. BOWDEN and B. BAUER, *T H E /proc F I L E S Y S T E M*. /usr/src/linux–x.y.z/Documentation/filesystems/proc.txt.

[15] T. CHILIMBI, R. JONES, and B. ZORN, *Designing a Trace Format for Heap Allocation Events*, in Proceedings of ISMM '00.

[16] E. CHOI, *Performance test and analysis for an adaptive load balancing mechanism on distributed server cluster systems*, Future Generation Computer Systems, 2004.

[17] ©2004 MICROSOFT CORPORATION, *Platform SDK: Windows Management Instrumentation – Performance Counter Classes*. msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/win32_perfformatteddata_perfos_system.asp.

[18] D. A. CORPORATION, *CAS Latency, What Is It?* www.dewassoc.com/performance/memory/cas_latency.htm.

[19] R. S. CORPORATION, *Using Rational PureCoverage®*. bmrc.berkeley.edu/purify/docs/html/installing_and_gettingstarted/3–pureCov.html#997695.

[20] ESKICIOGLU and MARSLAND, *Overview, Introduction, What is an Operating System?* www.cs.ualberta.ca/∼tony/C379/Notes/PDF/01.4.pdf, pages 22–23.

[21] F. GABBAY and A. MENDELSON, *Improving achievable ILP through value prediction and program profiling*, Microprocessors and Microsystems, 1998.

[22] HITACHI®, *Hard disk drive specifications Hitachi Deskstar 120 GXP, Revision 4.1*. D120GXP_sp41H.pdf.

[23] INFINEON TECHNOLOGIES® NORTH, *Intel Dual–Channel DDR Memory Architecture White Paper*. MKF_520DDRwhitepaper.pdf.

[24] INTEL®, *VTune™ Peformance Analyzer 2.0*. http://www.intel.com/software/products/vtune/.

[25] B. KING, *Moore's Law no more?* http://www.silicon.com/comment/0,39024711,11028346,00.htm.

[26] J. LEVON and P. ELIE, *OProfile – A System–wide Profiler for Linux Systems*. http://oprofile.sourceforge.net/.

[27] J. LIN, T. CHEN, W. HSU, and P. YEW, *Speculative Register Promotion Using Advanced Load Address Table (ALAT)*, tech. rep., Department of Computer Science And Engineering University of Minnesota, ???

[28] D. MACRI, *Processor & Platform Analysis Tools*. optimizations.org/optimizations/Siggraph2001–Macri1.ppt.

[29] RED HAT, *The Official Red Hat Linux Reference Guide*. www.redhat.com/docs/manuals/linux/RHL–7.3–Manual/ref–guide/ch–proc.html.

[30] SAMSUNG ELECTRONICS®, *DEVICE OPERATIONS DDR SDRAM*. ddr–device_operation_timing_2003_04_17.pdf.

[31] SEAGATE®, *Cheetah 10K.6 SCSI Product Manual, Rev. D*. 100195486a.pdf.

[32] J. SEWARD and N. NETHERCOTE, *The design and implementation of Valgrind*. http://developer.kde.org/ sewardj/docs–2.1.2/manual.html.

[33] S. SIEMEN, *Top Speed*, Linux Magazin, 2003.

[34] M. STONER, *Reducing the Impact of Misaligned Memory Accesses*. www.devx.com/Intel/Article/16850.

[35] UNBEKANNT, *ISO/IEC 14977:1996(E)*, in ISO/IEC.

[36] UNBEKANNT, *Robust Programming + Testing, Profiling & Instrumentation*. www.cs.princeton.edu/courses/archive/ spring03/cs217/lectures/Robust.pdf.

[37] UNBEKANNT, *Source lines of code*. http://en.wikipedia.org/wiki/Source_lines_of_code.

[38] UNBEKANNT, *What is the TLB?* www.tc.cornell.edu/Services/Edu/Topics/Performance/SingleProcPerf/tlb.html.

[39] J. S. VETTER and P. H. WORLEY, *Asserting Performance Expectations*, IEEE, 2002.